



DEVOPS ON GOOGLE CLOUD PLATFORM

Covering Docker, Jenkins, Kubernetes

SACHIN SRIVASTAVA

DevOps on GoogleCloud Platform

Covering Docker, Jenkins, Kubernetes

Sachin Srivastava

DevOps on Google Cloud Platform: Covering Docker, Jenkins, Kubernetes

Copyright © 2021 by Sachin Srivastava

This work is subject to copyright. All rights are reserved by the Author, 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 author nor the publisher can accept any legal responsibility for any errors or omissions that may be made.

Printed on acid-free paper

To my parents Vijay and Shobha.

Table of Contents

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
Chapter 1: Introduction to DevOps	1
What Is DevOps?	1
The DevOps Engineer	2
Adopting DevOps.....	3
The Manager Must Promote the Change	3
The Developer Must Be Responsible for the Software	4
The Operational People Must Be Treated As "First-Class Citizens"	4
Continuous Integration and Continuous Delivery Policies Must Be Built.....	5
Barriers to the IT Department Must Be Removed	6
The Release Process Must Be Automated	6
Agile Practices Must Be Promoted Across the Entire Company	7
Reasons for Adopting DevOps.....	7
What and Who Are Involved in DevOps?	8
Changing the Coordination	9
The DevOps Chain	11
Defining the Development Pipeline	13
Centralizing the Building Server.....	15
Monitoring Best Practices	16
Best Practices for Operations	17
Conclusion	18
Chapter 2: Introduction to GCP	19
Introduction to Cloud Computing	19
Cloud Computing Service Model	20
The Deployment Models	22
Why Use the Cloud?.....	23
Introduction to Google Cloud Platform	23
Starting with GCP	24
Understanding Billing in GCP.....	28
GCP Resources	30
Google SDK	32
Conclusion	35
Chapter 3: Introduction to Continuous Integration and Delivery	37
Definition of Continuous Integration	37
What Is Build in a Continuous Integration Scenario?	38
The Code Repository Server	40
The Continuous Integration Server	40
Continuous Delivery	42
Differences Between Continuous Integration and Continuous Delivery	44
Strategies for Continuous Delivery	44
Benefits of Continuous Integration and Continuous Delivery.....	47
Designing a Continuous Integration and Continuous Delivery System	49
Building Continuous Integration and Continuous Delivery Pipelines.....	51
Continuous Database Integration	53
Continuous Testing and Inspection	54

Preparing the Build for Release	56
Identifying the Code in the Repository	57
Creating Build Reports.....	57
Putting the Build in a Shared Location	58
Releasing the Build.....	58
Conclusion	62
Chapter 4: Containerization with Docker and Kubernetes.....	63
Introduction to Docker	63
Why Use Docker?.....	65
Using Docker in Google Cloud Platform	66
Introduction to Google Compute Engine	66
Creating a Compute Engine Instance	67
Instance Group	74
Container Application in Google Cloud Platform	75
What Is Kubernetes?.....	79
Using Kubernetes Engine to Deploy an Application	82
Deploying Our First Kubernetes App.....	85
Conclusion	96
Chapter 5: Continuous Delivery with GCP and Jenkins	99
An Introduction to Jenkins.....	99
Continuous Integration and Delivery with Jenkins.....	100
Code	101
Unit Test.....	101
Code Integration	101
System Testing	102
Stage Release.....	102
User Acceptance.....	102
Production Release.....	102
Designing a Good Branching Strategy	103
Using Jenkins in GCP	104
Configuring Jenkins on Kubernetes	105
Designing the Jenkins Architecture.....	110
Namespaces, Pods, Services, Quotas, and Deployments in Kubernetes.....	112
Create Jenkins Service.....	117
Deploying Jenkins on Kubernetes	124

Creating a Continuous Delivery Project.....	130
Creating the Repository.....	135
Creating Jenkins Pipelines.....	138
Creating the Jenkinsfile.....	142
Conclusion.....	146
Chapter 6: Microservices Architecture with GCP.....	147
Introduction to Microservices Architecture.....	147
Implementing a Microservices Architecture.....	149
Microservices Architecture with Docker and Kubernetes.....	150
Creating the Microservices Architecture in GCP.....	154
Creating the Services.....	155
Publishing the Image in the Registry.....	157
Creating the Kubernetes Cluster.....	161
Conclusion.....	163
Chapter 7: Monitoring in GCP.....	165
What Is a Monitoring System?.....	165
Factors Involved in the Monitoring System.....	167
Why Monitoring Is Important.....	169
White-Box and Black-Box Monitoring.....	171
Latency.....	172
Traffic.....	172
Error.....	172
Saturation.....	172
Building a Monitoring System.....	173
Configuring Stackdriver on GCP.....	174
Creating the Application.....	178
Log Analysis with Stackdriver.....	181
Alerts in Stackdriver.....	183
Policy Alert Configuration.....	185

Creating a Dashboard.....	187
Testing the Dashboard.....	189
Conclusion	190
Chapter 8: Creating and Managing Infrastructure in GCP	191
Infrastructure As Code	191
Infrastructure as Code Principles	192
Architecture for Infrastructure As Code	194
Infrastructure As Code in Google Cloud Platform	195
Configuration.....	196
Templates.....	197
Resource.....	201
Types	201
Manifest.....	201
Deployment	204
Starting with Google Cloud Deployment Manager	205
Upgrading Our Infrastructure	209
Expanding and Customizing Our Deployment.....	212
Creating a Template for Our Deployment	216
Defining the Template with the Environment Variable.....	219
Conclusion	222
Chapter 9: Identity and Access Management with Google Cloud Platform	223
What Is Identity and Access Management?	223
Defining a Digital Entity	224
The Importance of Digital Identity	225
IAM and Digital Entity.....	226
Authentication	228
Authorization	228
Roles.....	228
IAM in Google Cloud Platform	228
Google IAM Policy	232

Creating and Managing the IAM Policy	234
Creating a JSON File.....	235
Using the API via REST	239
Creating the API via Java Code.....	240
Using the Console.....	242
Conclusion	244
Chapter 10: Network Configuration and Management in GCP	245
Fundamentals of Networking in GCP	245
Introduction to the Virtual Private Cloud	247
Network and Subnets	249
Defining the IP Range for the Subnet	250
Routes and Firewalls.....	252
Tagging Network	254
Implementing a VPC Network.....	255
Creating and Maintaining Firewall Rules.....	262
Creating and Maintaining Routes in GCP	265
Conclusion	269
Index.....	271

x

About the Author

Sachin Srivastava has more than 15 years of extensive experience in the design and development of different scale applications, particularly in the telecommunications and manufacturing industries. He is also a freelance consultant, with specialization in DevOps, cloud, and security..



About the Technical Reviewer



Gerald Black is a DevOps engineer who has written three books about DevOps and micro-services. He works as a consultant, helping large companies to advance their systems development, by tweaking related software processes and tools. Gerald is also a Google Developer Expert in Kubernetes (Google Container Engine) and a member of the Node.js foundation.

xiii

Acknowledgment

My thanks go to Gerald Black, for the amazing feedback he provided during the technical review of the text. Another big thank you to Vikas Singh, and Auchitya, for their support and help in completing this book.

xv

Introduction

Cloud technology is increasingly present in our daily lives. The intent of this book is to introduce Google Cloud Platform, of cloud computing services, with an eye to practical DevOps practices.

DevOps is currently a hot commodity in the information technology (IT) field, and every day, a new company begins to adopt its practice. The cloud is a natural environment for DevOps, because in the cloud environment,

such practices as infrastructure as code and continuous delivery can be adopted more efficiently. Thus, the cloud is the natural environment in which DevOps can grow and be increasingly effective.

In 2003, Google created Site Reliability Engineering (SRE). This represented a new approach to the continuous large-scale release of product features. SRE shares many concepts with DevOps. I use Google Cloud Platform, because it is one of the most highly recommended by Gartner, particularly its use in infrastructure development. As with the Gartner Magic Quadrant, Google offers strong network management, and with the integration of Kubernetes, it is ideal for developing container and infrastructure solutions. By adding code, Google Cloud Platform offers a complete solution for creating the deployment management for release of infrastructure. In addition, Google Cloud Platform has one of the highest levels of application programming interface (API) available in the cloud environment.

All the codes in this book can be accessed from the related GitHub site. Please feel free to read that code and alter it for your personal use.

xvii

CHAPTER 1

Introduction to DevOps

DevOps, DevOps, DevOps, there is hardly a day in our professional lives that we don't hear that mantra. The reason is simple: by adopting DevOps practices, a company can reduce "time to market," the time necessary to identify a new software or other requirement and to put it in place.

In this chapter, I introduce some of the advantages of DevOps and the changes that must be made to accommodate and promote their use most profitably by a company.

What Is DevOps?

The term *DevOps* is derived from the combination of two words: *developer* and *operations*. *DevOps* is used to define a movement born of the need for reducing barriers between the development and operations teams of a company. The goal of DevOps is to reduce time to market. This means adopting DevOps practices, to reduce the time needed, from the identification of a new requirement to the time it goes live for customers. The DevOps journey introduces such practices as continuous integration and continuous delivery, which help to reduce time to market and produce better quality software.

The most significant and possibly most expensive failures related to a new release occurred in 2012, when Knight Capital Group, a global financial services firm, lost \$440 million, owing to a glitch in its servers during the release process, and, in 2013, when an upgrade failure at Goldman Sachs sent orders accidentally, resulting in the loss of what was thought to have been millions of dollars. DevOps allows a set of practices that can reduce potentially expensive errors.

When we think of DevOps, we must think of a movement formed to change how an entire company works together. The goal is to build a set of practices that can be used to reduce impediments to communication across different departments in the company. To be successful, DevOps should be promoted at the highest level of the company and accepted by each of its departments.

The philosophy behind DevOps was born in 2008, at the Agile conference in Toronto, Canada. During this conference, Patrick Debois presented his talk “Infrastructure and Operations.” In it, Debois expounded on the application of Agile methodologies for building an infrastructure. He offered three case studies and, at the end, proposed a better method of communication and other improvements allowing developers to gain more IT skills, in the sense of the system knowledge necessary to ensure a smoother release process.

The movement grew, until the 2014 publication of the book *The Phoenix Project*. In this book, authors Gene Kim, Kevin Behr, and George Spafford describe how to use DevOps to create a successful business. With this book, the start of the DevOps movement became official, and after, a new IT figure was born with the advent of the DevOps engineer.

The DevOps Engineer

The role of the DevOps engineer is a very recent one. The DevOps engineer represents a kind of bridge between the developer and the operations manager. In most cases, the role the DevOps engineer assumes is a mix of operations manager and developer, and this is fine, because these engineers must have the necessary knowledge for advising and managing a problem from both disciplines.

In some cases, the responsibility of a DevOps engineer is connected to continuous integration and delivery. Another responsibility associated with this position is infrastructure management, usually, infrastructure as code (IaC), and to help put in place optimal DevOps practices across the company.

Some companies view the DevOps engineer as an evolution of the site reliability engineer (SRE), whose main responsibility is to maintain the software in production and automate all the steps to solve an issue when one arises and to take the appropriate steps to ensure the normal administration of tasks necessary for maintaining the system. The role of the DevOps engineer is varied and can change from one company to another, but all have at their core ensuring the changes required to adopt the DevOps practices initiated by the company.

Adopting DevOps

Adopting DevOps in a company is like starting on a new journey. During this journey, management must be effective for change to be successful. The following bulleted list highlights the essential signposts of the journey.

- The manager must promote the change.
- The developer must be responsible for the software.
- The operational people must be treated as “first-class citizens.”
- Continuous integration and continuous delivery policies must be built.

- Barriers to the IT department must be removed.
- The release process must be automated.
- Agile practices must be promoted across the entire company.

To achieve the desired result requires that DevOps changes be initiated by management and integrated into the company culture.

This step is very important, to ensure a successful DevOps journey, and, of course, it involves some technical changes, to be really effective. Let's examine in detail what this means.

The Manager Must Promote the Change

To be successful, the changes required by DevOps must be pushed and accepted by management first. For change to be effective, it must have strong company approval.

Imagine, for example, at the outset of our journey, that we start a new development. We want to design a continuous integration/continuous delivery (CI/CD) system. To do that, we decide to adopt Scrum as our methodology, instead of Waterfall, which was previously used.

One day, the chief technology officer (CTO) proposes a new feature that he/she absolutely wants to release in spring, only months away, but to do this, we must cancel and delay some other features. The Scrum Master tries to tell the CTO not to introduce the new feature so soon, because it will delay other features and cause some issues. The CTO insists, and using his/her power, pushes the feature to spring. To hit this date, the team must postpone some other jobs and work faster on the new feature, thereby creating some software quality issues. The requirements are not completely clear, and during the CI/CD, cycle issues are identified, and this makes the software of very poor quality and essentially not ready for release.

In the end, the team loses confidence in the DevOps practice, and little by little, everyone goes back to the usual way of doing things.

The Developer Must Be Responsible for the Software

In the normal development cycle, the responsibility of the development team ends when the software is released to live production. After that, the developer works on new features and is involved only when operations find a bug to fix. But this means a new feature must be released, and the operations team must find a way to mitigate the bug.

If we want to have a successful DevOps journey, we must empower the developer. This means that when the operations team finds a bug in the software, the developer working on the function must be involved in the fix. This has two major advantages:

- The developer can more easily identify the problem and find a fix for the issue. Because he/she knows the software, it is easier for him/her to find the root cause of the problem.
- Because the developer can identify a problem's root cause, it is easier for him/her to work on a permanent solution. This, with a CI/CD practice in place, reduces the time to market for release and improves the quality and stability of the software.

This requires a big change in company culture: to lead the way to another important change for DevOps, which, of course, must have complete management approval to be really effective. The big advantage is that this assures improvement in the quality of the software.

The Operational People Must Be Treated As“First-Class Citizens”

When we design a new feature, the development and the architecture teams must be involved with the operations team. This is because those responsible for the correct functioning of the live software make up the operations team.

The role of the operations staff during architectural decision making is particularly important in the release of a new feature. Imagine, for example, that we must design a new feature for our system. The developer proposes a fancy new component for the web interface and offers a mini demo. Of course, on the developer's laptop, no issues occur. Problems can arise, however, when the component is tested on the actual server. The only people

who can respond effectively to such problems are the operations technicians, those who know the server on which the software should be installed and run and who know all the policies related to security, software version, etc. In this case, the operations team can reject the component, because, for example, it does not meet company standards, or the team can start a process to test the server and ready it for the new component.

This means, for this Sprint of the next n-Sprint phase, the component cannot be used, but the operations team can advise the development team when the server is ready. Another important reason for including the operations team in the design of software is log level. For a developer, the message can be clear, but this is because he/ she knows the software and understands what's happening. Operations personnel must be able to understand an issue primarily by reading the log. If the log is too chatty or otherwise unclear, this will impede a correct analysis of the error and cause a delay in finding a resolution and identifying a root cause of the problem.

Continuous Integration and Continuous Delivery Policies Must Be Built

Using CI/CD policies helps the development and operations teams to identify faster potential issues with the software. When we establish a practice for CI/CD, we receive constant feedback. An important part of every CI/CD system is the code review. When a developer completes the code, it must be fully tested. First, to ensure successful integration, the developer must ask other software engineers to review the code and raise any issues found in it. This is a very important procedure for the CI/CD system. For every simple update in the software, the developer must begin to adopt test-driven development (TDD) practices. In this way, every commit can be fully tested by the CI software, for example, Jenkins, and promoted to the next step, for example, creation and delivery to the quality assurance (QA) environment for testing.

Having a CI/CD system in place helps to improve the quality and stability of the system. The reason for this is connected to the nature of the system. The entire life cycle of the software is extended every time we commit a new file to it. This means, in the event of a new bug, that we can determine in real

time the cause of the error, and we can easily roll back the software delivery. At the same time, we gain the ability to review the software constantly. This helps the operations team to identify security risks and take all the necessary measures needed to eliminate them.

But to achieve true success and avoid creating problems and destabilizing the system, the software engineer must invest more time in unit testing and automation testing. In addition, we must invest time in code review. If we don't have a good test coverage of the software, or don't really review the code, we can release software that is potentially dangerous and compromise the security of the application.

Barriers to the IT Department Must Be Removed

Normally, development and operations teams use different software for their daily work. This can create barriers that must be removed, if we want to ensure an effective DevOps journey.

DevOps can unify communications tools and promote communication across different IT-related departments. In this way, we can coordinate the time frame for the software release and can better plan the work involved. If, for example, a new feature introduced to the CI system creates a security bug, the security team can use the common channel to communicate to the development team, and this can put in place a fix for solving the problem. At the same time, the operations team can be advised of every step the developer has taken with the software and can be ready for the release time.

The Release Process Must Be Automated

Analyzing the error rate, we can positively identify humans as the main cause of failure. The main focus of DevOps is to reduce human and other errors and reduce the time to market. To achieve this, one of the important changes we must make is to automate the release process.

With an automatic process for releasing the software, we reduce the human interaction with the system. Less human interaction reduces the number of failures, because we have a predictable and repeatable process in place.

The other advantage of having an automatic process in place is the possibility of defining the IaC. With the IaC, we can define, via code, what type of structure we want for our software. In addition, defining the IaC makes the infrastructure predictable and allows for the faster release of upgrades. Automating the release process and defining the infrastructure reduces or removes, in the best-case scenario, human interaction and, for this reason, effectively reduces the error rate.

Agile Practices Must Be Promoted Across the Entire Company

DevOps was born during an Agile conference, and to be effective, a company must begin to implement Agile across all its departments. Normally, the Agile practice is mostly used by the development team, but for a good DevOps journey, we must spread this practice as well to the infrastructure and operations teams. If the entire team works in Sprint and, if possible, shares the same backlog, this can help to improve communication. During the Sprint planning, the other team can adjust the work in order for it to be more effective.

At the same time, with Agile in place, we can improve communication and make the work more visible across the team. At the end of the Sprint phase, we can see a demo of the work of the other teams. This helps to see how effective the work is and how to improve the iteration from one team to another.

Reasons for Adopting DevOps

There are different reasons why a company decides to adopt DevOps. Normally, the adoption of the DevOps philosophy is related to improvement in the quality of the software and a better way of managing its release.

When a company adopts DevOps, the first step is to improve communication across teams. This characteristic of DevOps is shared by the Agile methodologies and can be put in place only with a harmonization of the tools used across the company.

This change is not always easily accepted by all IT employees. The initial resistance is usually to the change of culture necessary to adopt DevOps. In general, the life cycle for designing and implementing infrastructure is managed using the ITIL. This means the procedure follows Waterfall methodologies, because it is essentially impossible to configure a server without the server being physically in your hands.

Adopting DevOps means changing the way we think of infrastructure: where possible, migrating it to the cloud, adopting infrastructure as code, and adopting the compatibility of the case, using Sprint to manage the work. This demands that all teams use common project methodologies and create a common product backlog that is shared with the development team, in particular, when the project involves new infrastructure.

Another reason for adopting DevOps practices is the improvement in the quality of the software released. With DevOps, we can adopt some procedure for improving the quality of the software. For this we must have in place continuous integration and continuous delivery. With these, it is easy to identify errors when we push the code on the repository. In addition, because we have continuous delivery, we can release the software directly on the QA more times per day. This ensures a continuous check of the software and continuous feedback for the software engineer.

These are just some common reasons that drive a DevOps journey. Whatever the reason, it is important to understand what actors are involved in DevOps. To do that, we must clarify some misunderstandings commonly connected with the use of DevOps. We must try at this point to identify the common mistakes associated with DevOps and to clarify its role and who is involved.

What and Who Are Involved in DevOps?

In talking about DevOps, we can encounter some misunderstanding of what it is and who is involved in it. The first myth regarding the adoption of DevOps is associated with the professionals who deal with it. DevOps, for many people, involves only software engineers, system engineers, and system administrators.

This assumption is incorrect. When a company decides to adopt DevOps, the first change required is to improve communication across the various teams. This means not only development and operations but other teams as well, such as QA, security, and business. To be effective and successful, a DevOps journey requires that all team members work together. The goal of DevOps is to reduce time to market. This means that when a new feature is designed, every team must communicate, to reach the goal. The QA engineer must respond quickly to the software engineering team and communicate any glitch found in the software. At the same time, the software engineer must communicate with the security team, to describe what the software does and what libraries are used, and to allow the security team to marshal the necessary assets to ensure the safety of the software. The business analyst must be aligned with the software architect, and the software engineer with what the customer wants.

As you can see, to undertake a successful DevOps journey, the whole organization should be involved. Every team must take responsibility for a small part of the business, but in tandem with other teams. DevOps seeks to remove communication barriers across teams, making it easier to identify and correct errors during development and not after release. This ensures better software, a more timely release to market, and better alignment with what the customer needs and wants.

All these actors must work together like musicians in an orchestra. If all respect the symphony, everything runs smoothly, but if one team starts to make problems or doesn't practice good communication, the intended goal will be compromised. For this reason, the most important job when adopting DevOps is to improve the coordination of the internal and external teams.

Changing the Coordination

By adopting DevOps, one of the goals we want to achieve is the reduction of coordination. This means ensuring that those responsible for managing the team invest less time coordinating the different operations. This becomes very important when moving the software from the development server to the stage server. If a CI/CD practice is in place, the software is automatically promoted.

When more automatic processes are introduced, human interaction is reduced and, thereby, the requirement of coordination. This is necessary to reduce time to market. Fewer humans require approval; therefore, fewer delays occur. This requires a change in the classic coordination processes. When we adopt a nonautomatic process, normally, when we finish the software development, the team responsible for the development communicates the completion of the development and then coordinates with the other teams responsible, to move the software onstage. This coordination essentially delegates to humans different ways in which to communicate, for example, via e-mail. DevOps tries to change this way of coordination, reducing the human interaction and, of course, changing the way coordination is actualized.

The coordination has different means of being actualized. These change depending on the context—whether the team is remote, on-site, or partially remote. The normal attributes necessary for good coordination are to be

- Direct
- Indirect
- Persistent

These three attributes define how we manage coordination across the team. Every style has its strengths and weaknesses, so we must be sure to use the correct type of coordination for our purposes. The wrong type can result in an unnecessary consumption of resources and poor coordination. I will now discuss the different styles and when to use one instead of another. I will describe how to use Agile, to improve coordination and to split it across roles and artifacts of the Agile methodologies.

The goal for all the kinds of coordination is to improve communication and, with that, reduce time to market. Remember: The ultimate goal of DevOps is to reduce time to market.

Direct Coordination

With direct coordination, those responsible for the coordination know each other. This means the coordinator directly coordinates the job of every team member. This kind of coordination requires a lot of work from those

responsible for the coordination. Normally, this effort can be mitigated when the team is managed using Scrum. In this way, during the stand-up, the staff responsible for the coordination can receive direct feedback regarding the status of the team and take decisions about that.

Indirect Coordination

By this type of coordination, we don't just coordinate people, we coordinate a team, for example, system administration, software engineering, etc. This kind of coordination requires greater coordination, because we don't really go deeply into the details of the task but approach it from a higher level. Imagine, for example, that we must manage new software being put in place, a new piece of infrastructure, and new software functionality. The coordination we want to have is not about detailed tasks but a general view of the status of some specific task. This view gives to the coordinator the capacity to have a plan and start to move on the other activities to calculate an estimated time for the release.

This kind of planning is normally delegated to the product owner. Keeping still to the Scrum style of management, the product owner doesn't really go deep inside a single functionality, but he/she takes an overall view. The production owner is responsible for the entire project and, of course, can help the team reach the best result, reducing unnecessary effort.

Persistent Coordination

This is not really a kind of coordination but essentially an artifact. Persistent coordination refers to all the reports and e-mails sent when a decision is reached about a project.

Persistence gives the team all the instruments for keeping a daily record of the production story and allows teams to make new decisions, based on the history of the project and prevents any misunderstandings about the project itself.

The DevOps Chain

Until now, I have discussed only the kinds of coordination available and what kinds can be used to improve communication. However, the most important question we want to answer is why coordination is so important in DevOps.

The reason is simple. The DevOps movement progresses according to a “toolchain.” Essentially, this toolchain is used to define every step of the production process.

Figure 1-1 shows the phases for the DevOps of a software release. Every phase can be managed by a different team. For this reason, strong and clear coordination and communication are important.



Figure 1-1. Porter’s value chain for DevOps

To better understand the importance of coordination and communication, we must understand how every phase is connected to the other, creating a “chain” of production for the software.

The first phase is the *code* . During this phase, code for the software is created. Every developer puts the code in a common repository, for example, Git, and this leads to the next link in the chain.

The second phase is *build* . This phase is directly connected with the continuous integration practice. The code previously committed is downloaded in the build server and then built in an automatic way. At the same time, a test is performed for the first time. If all elements of the test are successful, the next phase begins.

The third phase is the *test* phase. The software previously built is tested by some automatic process, but this time, the software is tested altogether. In the build phase, only the unit test connected with the specific functionality we release is executed. If the system doesn’t find any issue, the software is promoted to the next phase. In case of failure, the software will be rejected, and an automatic system will advise the developer of that.

The fourth phase is the *configure* . This phase requires a clear distinction. When we have good and tested DevOps practices in place, we can have continuous release. This means, obviously, the continuous release of the software in production. However, for software that is mission critical, this phase is normally split into two different parts. The first release is intended for a restricted number of servers called *canary servers* .

Note The term *canary server* designates an intentionally restricted number of servers that are used to test the new software. The purpose of the canary server is to allow a real user to use the new software and provide real-time feedback about potential bugs and the quality of the server. A canary server is very important when we want to be sure not to release software that can be potentially destructive to the company. At the same time, canary software can be used for pen testing and to improve system security.

The fifth phase of the chain is *release* . In this phase, the server is configured, as well as the infrastructure for the new software. This phase defines IaC. The server is created and managed using the code. Such software as Chef, Puppet, Ansible, and CloudFormation are examples of software for creating IaC.

Note Infrastructure as code (IaC) is at the core of DevOps. IaC provides the ability to create the infrastructure for a new server. This guarantees the integrity of every new release, because human interaction is reduced and the integrity of the release is improved. In addition, IaC allows DevOps to create a different environment as a request. This makes it possible for the developer to create a different environment and different test environments directly on request. With IaC in place, we can create and orchestrate an *immutable infrastructure* , that is, an infrastructure composed of some immutable components that are replaced every time we release the infrastructure. Instead of updating a component for our infrastructure, we can simply deploy a new immutable component with the necessary update. This guarantees the stability of the entire infrastructure and ensures that an infrastructure always yields the same result with every release.

The sixth phase of the chain is *monitor* . This is extremely important for providing continuous feedback about our software and infrastructure. Monitoring is very important in DevOps, because it allows the developer to

gain feedback about the software, including an average of the failure, the kind of failure, etc., and, at the same time, can be used to check the metrics of the server and provide feedback for autoscaling it.

Coordination and communication are crucial for putting the complete DevOps chain in place. This is because every phase requires a good coordination at every step. We must ensure reliable feedback at every step, because we must react quickly to errors and adjust the system, to prevent new errors.

Defining the Development Pipeline

To ensure a successful DevOps journey, one of the most important jobs is to define the development pipeline. Building this pipeline is essentially the core of the changes required by DevOps.

The first of the changes in the development life cycle is to put in place continuous integration. This requires some changes to our development practice, which can be summarized as follows:

- Define the unit test.
- Define a branch policy.
- Have in place a continuous integration system.

These three practices are the backbone of the development pipeline. The first, the *unit test* occurs every time the developer commits code to a central repository.

When the code is committed to the software for continuous integration—for example, Jenkins—this compiles the code and executes the unit test associated with the software. In the case of failure, an e-mail, with the test results, is sent to the developer.

Because we don't want to break the main branch, we adopt the second practice, the *branch policy*. This is important for maintaining a clean master branch. When a development team adopts this policy, every developer creates a specific branch when developing a feature. This policy is strictly connected to a code review. For every merge with the master, a code review occurs, after the build has been completed and correctly tested. Essentially,

for every commit, only the branch is built. In this way, in case of error, the master is not broken and is always ready for a release.

In the case of a positive build, we can ask for a code review, and when the code review is complete, merge the branch with the master, and, of course, restart a complete system for *continuous integration*. With continuous integration in place, we build and test every time we commit into the master or a branch.

Continuous integration must be paired with a good communication system. In particular, we must have a good mail system, to send e-mail to the developer to brake the continuity in the pipeline.

With this pipeline in place, we have continuous software production. What closes the pipeline are the release and monitoring.

During the development life cycle, the release does not happen during production but in QA and testing the server. This release happens automatically. Essentially, it is a promotion of the software built by the continuous integration system. This release is used for testing purposes by the QA engineer, to test the software, provide faster feedback to the developer, and put in place fixes to any bugs faster.

Having a release in QA is important, not only to fix bugs, but to start the monitoring phase. Monitoring is very important in DevOps, to reduce and prevent errors for occurring in the system.

Monitoring is very important for checking and maintaining the stability of the system. A good monitoring system must check not only the availability of the system, for example, if the network is available or the software is working, but can be used for preventing future errors.

There is a lot of software for monitoring, for example, Nagios, Prometheus, Zabbix, or the ELK combination, Elasticsearch, Logstash, and Kibana. All this software has its specific strengths and can be used in combination for achieving the best results.

One of the crucial reasons for effective monitoring is the log. With a good log, it is easy to initiate some log analysis policy. This policy is intended to isolate common error conditions and define some practice of mitigating the error and, at the same time, give to the developer the critical space to fix the software.

Centralizing the Building Server

Centralizing the building server is crucial for building the correct pipeline. When we design a DevOps architecture, we must think of reducing points of failure.

When we adopt a build server, we centralize everything in one server. This means that we use a different software to release our new software. Having only one server, or cluster, for building new software means that there is only one point of failure. Any problem is centralized at only one point. This reduces the cost of maintaining the software and speeds up the release process.

The building server is normally connected with an artifact repository server. The repository server is where the build is stored and saved. It is associated with continuous release. Essentially, with this practice, we build the software every time and release it to a server. This server essentially maintains the different versions of the software we build on the server. Normally, we establish a naming policy to maintain the different software versions. This is because we want to identify every version of the software uniquely.

With the artifact server, we can easily centralize one point for release of the software. In this way, we can have different versions of the same software, and, if we use Docker, we can have different versions on the same server at the same time. We can also start them at the same time, with some minor adjustments. This allows the QA engineer, for example, to undertake some regression test, and in case of new errors, identify exactly what version has the bug. This allows the developer to understand exactly what change to the code introduced the error.

Monitoring Best Practices

To be effective, monitoring must be combined with some other practice. A log analysis is the most important practice for preventing errors and understanding how the system functions. Some software is required for analyzing the log and making related predictions.

The software most commonly used is ELK (Elasticsearch, Logstash, and Kibana). This ecosystem is helpful because it gives a complete log analysis

system, not only providing alerts, but also a graphical representation of the error and the log.

Log analysis is very important for improving the quality of software. One important practice we can put in place is to have some software that not only identifies the number of errors but graphs these as well.

Having a graphical representation of the errors is important for providing visible feedback about the software, without the necessity of reading a log, in order to understand the status of the software.

Monitoring is the backbone for every DevOps practice, and if we want a very successful journey, we must be sure to have a good monitoring system. At the same time, we must start to monitor not only the production but possibly the canary server. This is because it can reveal an error, and we can solve it before release to production. Monitoring can take two forms. *Black-box monitoring* tests a piece of code as if it were in a black box. It reveals only the status of the system, to determine whether it is alive. It doesn't really indicate what is happening inside, because the monitoring is external. An example of black-box software monitoring is Nagios.

The opposite of this is *white-box monitoring*. This type of monitoring provides a clear picture of the inside of the system, such as, for example, the number of HTTP connections open, the number of errors, etc. Prometheus is an example of white-box monitoring software.

Best Practices for Operations

In DevOps, the operations team has a big influence on achieving the best results. The importance of the operations team is strictly connected to the quality of the software and what the customer thinks about the company.

In case of an error, the operations team is the first face of the company. This team is normally delegated to maintain the software in the production environment.

The only point of contact with the software is the log. For this reason, some member of the operations team must be included when software is designed, and, more important is the feedback they can provide when software is

released for testing. This is because if the log is insufficient, the operations team can't really identify the error, which means more time will be required to fix the issue.

At the same time, the operations team can help to identify common issues and provide documentation to solve them faster. This documentation is actually a step toward resolving the issue. It is used essentially by first-line operations engineers. It is “live,” meaning that it is never closed and must be carefully managed, so that it aligns with the most recent software updates.

The documentation must indicate common errors in the log and show how to solve the root cause(s) of the problem. This documentation should be written with people who don't know the system in mind and, based on that, must provide specific details about the appropriate steps to be taken.

Another operations practice we can put in place is *developer on-call*. This practice introduces a new figure to the operations world. The developer on-call is essentially a software engineer, working with the operations professionals to resolve errors in production. This has two principal advantages. The first is a reduction in the time it takes to identify and fix an issue. Because one of the developers works on the issue, he/she can easily identify what's gone wrong and what part of the code creates the issue. This can drive the operationsteam's efforts to fix it.

The second advantage is improving the level of responsibility. Because the developer works to fix a live issue, he/she understands better what's wrong with the software and thus can improve the way he/she writes the software and the log, because a bad log can result in more work for him/her in future.

Conclusion

In this chapter, I have offered a brief introduction to DevOps —what it is and how a movement was born. DevOps is very important in relation to the cloud. Cloud development requires software that is always live and designed to be released faster and of higher quality.

DevOps puts the accent on quality and on time to market. It allows for a simple design microservice architecture, because of the practices connected with continuous integration and delivery, which help to deliver faster service on a system.

DevOps is very important to modern software development, and more companies are starting to adopt it, because it promotes some best practices necessary for improving the quality of software. DevOps requires a change, not only in how we think of infrastructure, but in how well we design and organize the internal company infrastructure.

DevOps essentially represents a change in corporate culture. To ensure its optimal practice, it is important to change the organization, so that its priorities align with the requirements of cultural change, practices for success, and change that is driven by management and, of course, approved by the engineers.

CHAPTER 2

Introduction to GCP

The cloud is one of the technologies most frequently adopted today. More and more companies have started to use this technology for their projects. This change is driven by different factors, for example, a reduction in the cost of spinning up new projects. Of course, this is only partially true. Normally, the cloud can be more expensive in the long run, but, on the other hand, it can reduce resources. In the cloud, it is easier to spin up a new operating system with all the environments necessary for its application.

At the same time, a cloud solution offers flexibility in terms of cost savings not available in a traditional infrastructure. With the cloud, we generally pay for what we use. This means that if we don't need, or simply don't use, an instance, we don't pay for it. This is a big saving for a small company that needs to spin up its business. With a traditional infrastructure, we must pay the electricity bill for the server, in any case. The major competitors in the cloud are essentially three: Amazon Web Services, Microsoft Azure, and Google Cloud Platform (GCP). In 2017, Google was cited as the most visionary cloud, by Gartner Magic Quadrant, based on its leading position in the field of cloud computing.

In this chapter, I provide a brief introduction to cloud and, following, an introduction to Google Cloud Platform. In addition, I review the similarities between Amazon Web Services (AWS) and GCP.

Introduction to Cloud Computing

Cloud computing is a new IT paradigm. The definition of cloud computing derives directly from the National Institute of Standard and Technology (NIST), as follows:

Cloud computing is a model for enabling ubiquitous, convenient, ondemand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

From the preceding definition, we can see that the cloud is essentially a shared resource within a network, another important distinguishing point. The cloud also has some other characteristics, which are essential for defining cloud computing.

- *Self-service and on-demand* : A customer can unilaterally define the required resource and pay for it only when used.
- *Network access* : As stated in the NIST definition, the cloud is a set of services offered via networks. This means that customers can access different resources via networks and, possibly, across different platforms, such as, for example, mobile phones, tablets, and thin client computers.
- *Pooling of the resources* : The cloud allows for the sharing of resources with different people at the same time. In addition, different users can have a different configuration. Cloud computing can manage this necessity and share resources with different users.
- *Faster response to change* : This capability of the cloud allows for faster releases, most of the time, in an automatic way. This means the cloud must be faster to adapt, in order to share new resources.

- *Measurable service* : Every cloud system must allow the user to manage and check the chosen resource. The cloud must have a system to control and manage the resource used, including the ability to stop its use. Normally, the cloud uses a pay-per-use paradigm, meaning the user pays only for the resource consumed.

Another important characteristic of cloud computing is its different service and deployment models. These various services and models, de facto, define how we can use cloud.

Cloud Computing Service Model

The most important differentiation of the cloud is the service model. The service model defines how to offer cloud to the customer. Every model has different characteristics. Service models include

- *SaaS (software as a service)* : The SaaS model allows users to design

and promote their software. Based on the cloud's infrastructure, this means consumers essentially share some resource on the network and run their software on that resource. Consumers are not able to manage the underlying infrastructure, such as the network, operating system, etc., and they are able only to use the software. The software is accessible on different platforms, such as mobile phone, tablet, or computer. Good examples of SaaS include Google Docs, Office 365, and Zendesk, with which the consumer can access software from different platforms with, in most cases, only the requirement of an Internet connection.

- *PaaS (platform as a service)* : The PaaS model offers the user the ability to deploy his/her own software, library, database, and everything else required for the software, in a cloud environment, without thinking about the underlying infrastructure. With PaaS, the consumer can develop an application, using the language provided, and essentially not care about the resource. An example of PaaS is Heroku, in which a consumer develops an application using his or her own preferred language, and the only work required is to create the software infrastructure, such as the database, and deploy the software for allowing the user to access his/her cloud resource.

- *IaaS (infrastructure as a service)* : The IaaS model is the most manageable service model. With IaaS, a user can run a preferred software. The software is run in the infrastructure the user creates. This includes the operating system, applications, or anything else that is required. At the same time, the user is able to manage some underlying network features, such as, for example, the firewall and load balancer. Good examples of IaaS include GCP, AWS, and Azure.

All these service models are important for defining the kind of cloud we want to use. Every service model affects how the customer approaches and uses the cloud, whether that be only to use software over the network across a different kind of platform or to define the operating system and, starting from that, build the application.

The Deployment Models

In cloud computing, we can identify four different types of deployment models. The difference in the models is defined only by how we release on the cloud, but not by what kind of cloud we will deploy. This means that each of the service models can be defined according to the four deployment models.

These models and their differences are listed following:

- *Private* : The private model of cloud computing is intended for internal use only. This kind of cloud is normally built to be shares in a private company, across the different business units. This cloud is completely owned and managed by the provider, normally, another business unit of the company.
- *Community* : This kind of cloud model is intended to be shared across a specific community, for example, a security community or charity. In this type of deployment, the community owns and manages the cloud infrastructure.
- *Public* : This cloud model can essentially be used by anyone. Usually, it is rented. Examples of this kind of model are Amazon Web Services, GCP, and Microsoft Azure.

- *Hybrid* : This type of cloud model combines different models. It is normally used when there is a need for part of the data to remain private, for example, customer data, for security reasons. The purpose of having a hybrid cloud model is to combine characteristics of different kinds of cloud models, for example, a need for privacy but, at the same time, a need to share some information with a community.

The different deployment models are intended to respond to the different needs of the potential user. For example, a hybrid solution can be adopted when we want to have an SaaS cloud deployment but have concerns about the security of the data, because of, for example, some regulation that makes it necessary to secure how our data is managed. In this case, we can have the SaaS in the public cloud, for example, GCP, but store the data in our internal private cloud. This can essentially meet all the legal requirements and, at the same time, provide public software to our customer.

Why Use the Cloud?

As with every technology, the cloud has some benefits and some disadvantages. The cloud can be cost-effective. In particular, if we want to start a new business, adopting the cloud can be cheaper than buying and maintaining an internal server.

In addition, the public cloud allows for service-level agreements. About 99% of the time, this means our site and our business are essentially always online. A public cloud also can help a company to scale with the business. We can easily add more power to our actual infrastructure in minutes and scale down when we don't require this power anymore.

This helps to save the company money and to improve profitability. Another important consideration is the total cost of operation (TCO). TCO is generally calculated based on all the costs of hardware and maintenance. The TCO of the cloud is very low, because we essentially rent the hardware on a monthly fee, and we can always reduce the cost by reducing the resource used, according to business necessity.

Because the cloud infrastructure can essentially grow with our business's needs, this creates a big return on investment (ROI). This is because we pay

only for what we really use. With a traditional infrastructure, we reduce the initial ROI, because we must repay the initial hardware investment, and this cannot be supported by our business.

Introduction to Google Cloud Platform

GCP is the public cloud offered by Google. It is made up of a suite of services run on the same infrastructure with which Google runs the software for the customer, such as YouTube and Gmail.

GCP was first offered to the public in October 2011. Since that time, it has continued to grow in popularity and is now the third most popular cloud platform, after AWS and Azure.

GCP offers a wide range of services, which can be divided into the following areas: computing and hosting, storage, networking, big data, and machine learning.

For every area, GCP offers a complete set of products that can be used to build our own cloud application. The most popular services are

- *Google Compute Engine* : This provides the capacity to create a virtual machine to run an operating system. It allows for the creation of different “computers” inside the cloud.
- *Google App Engine* : A PaaS component for building applications. With App Engine, it is possible to create applications with different types of languages and frameworks. At the time of writing, App Engine is supported by Go, PHP, Java, .NET, Ruby, Python, and Node.js.
- *Google Kubernetes Engine* : A managed orchestrator for the container, this is used to deploy, scale, and release containers.
- *Google Cloud Bigtable* : A compressed high-performance and proprietary data storage feature developed by Google.
- *Google BigQuery* : This is a RESTful web service, used to analyze a massively large number of datasets.
- *Google Cloud Function* : An event-driven serverless cloud platform. With Function, it is possible to create infrastructure as code, that is, infrastructure designed and implemented by the code.

- *Google Cloud Datastore* : A highly scalable fully managed NoSQL document database, built on Bigtable and Megastore technology
- *Google Storage* : This is a RESTful service for storing data on GCP. It is comparable to the Amazon S3 service.

This is only a short list of services, but it is enough to start our journey. The next step is to set up and configure a GCP instance.

Starting with GCP

We can create a new GCP account by following some simple steps. First, we must connect to the site: <https://cloud.google.com/> .

This is the initial page for connecting to GCP. To create a new account, simply click the Try GCP Free button. Insert all the parameters, to create the free account. To complete the process, we must provide credit card information, but don't worry, no fee is charged. Use remains free for one year or, alternatively, until we don't reach the \$300 free trial limit. After that, the account is automatically converted to a pay-for-use one.

When the account is initiated, we will see a page such as that shown in Figure 2-1 .

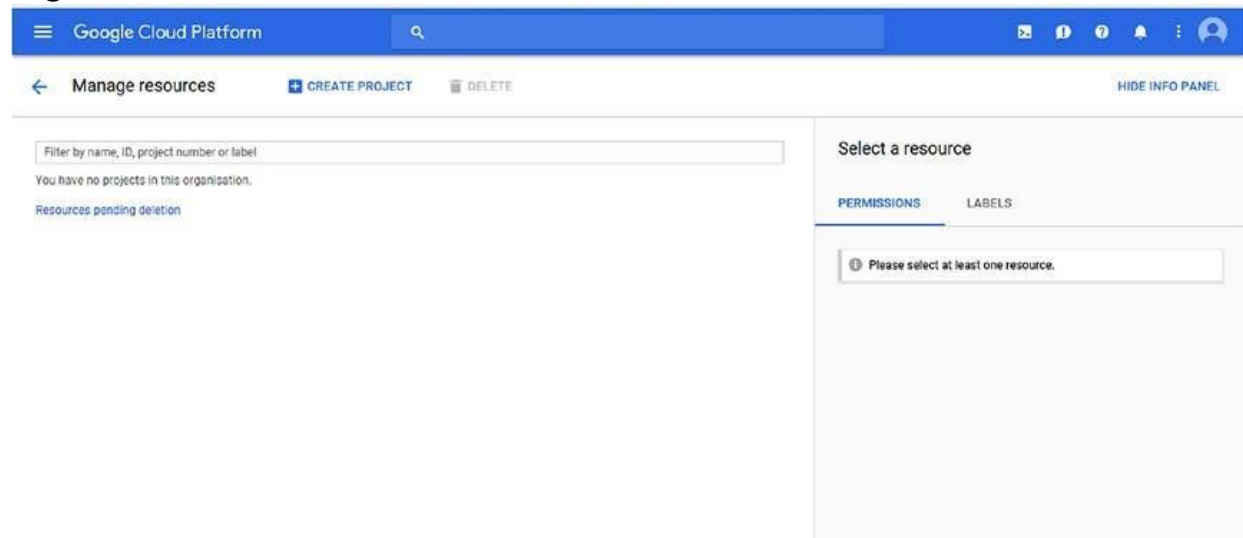


Figure 2-1. The Google Cloud Platform screen

The initial page of GCP shows the resource used and the project we have in the cloud. Of course, because we've just started, the page is blank.

The first action we must take is to create a new *project* . A project is

essentially a container for regrouping all IT and non-IT resources connected with a specific cloud project. Every project is identified by some specific parameter, as follows:

- *Name* : This is a string used to identify and describe the project. The name is only for user reference and can be changed at any stage. Using the free tier, we can create 24 projects.
- *Project ID* : This is a globally unique string for identifying the project. It is created starting with the project name. It is possible to edit and change the project ID. To create the project ID, we can use any lowercase letter, number, and hyphens. The only requirement is the unicity of the name. After this is entered, it is no longer possible to change it.
- *Project Number* : This is a parameter that is autogenerated by GCP. We can't manage or change this number.

To create a new project, just click the Create a new project button. This starts the process of creating a new project (Figure 2-2).

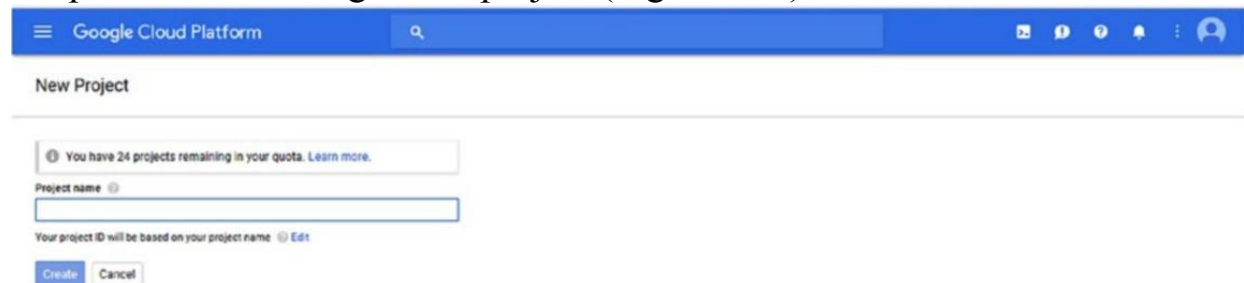


Figure 2-2. Creating a new project with GCP

Now we can see that to create a new project, we must add a new name. In this case, we assign the name *PracticalDevOpsGCP* , leave the project ID with the default value, and click Create.

Clicking the Create button creates the project for us. This takes a few seconds. When the project is ready, we can click the right side to return to the main page. In this case, we move back to the dashboard.

The home dashboard shows the command to create a new project or select another one (Figure 2-3).

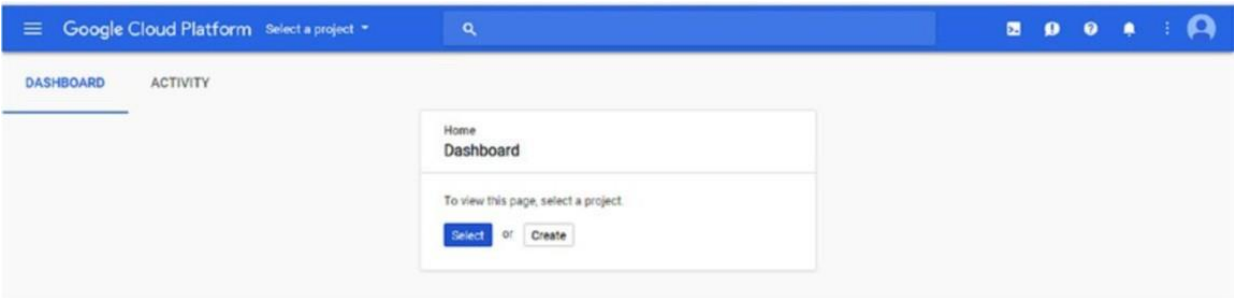


Figure 2-3. *Entering the Google dashboard*

Because we have created the project previously, we click the Select button. This takes us to another page from which to select the project we want to open (Figure 2-4).



Figure 2-4. *Selecting the project*

Select our project and then click Open, to open the project. When we open the project, we see the complete dashboard from GCP (Figure 2-5). This dashboard can be configured to adapt to our needs.

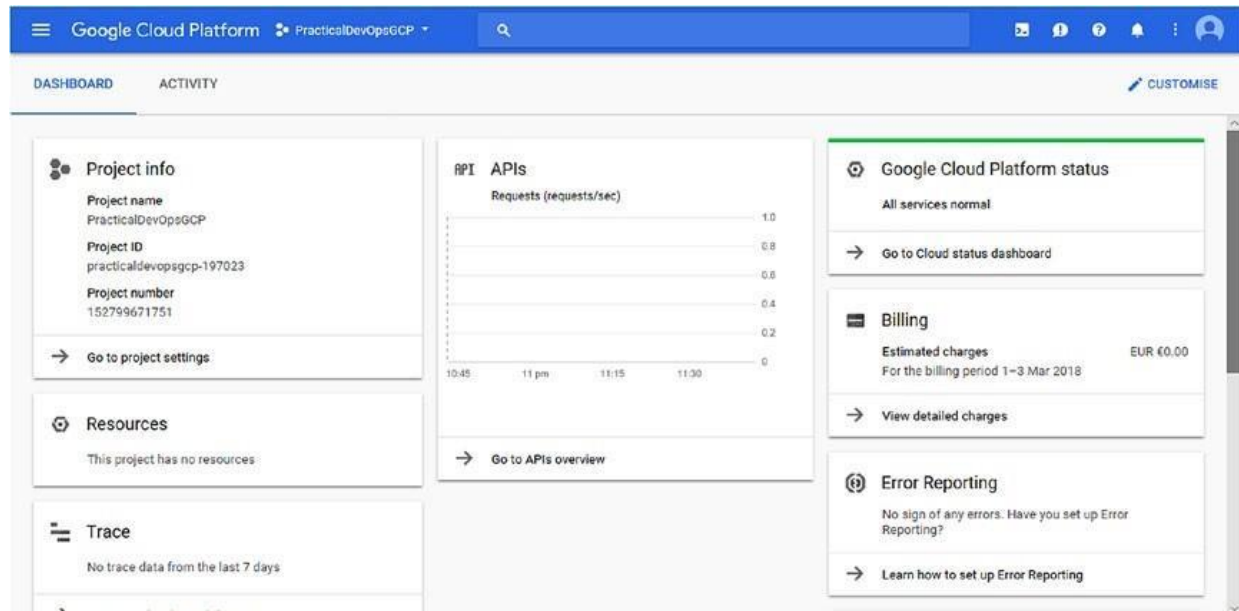


Figure 2-5. *The Google dashboard for our project*

On the left, we can see “Project info.” This shows the general information about our project. At the center of the screen is APIs Requests (requests/sec). This is an important parameter that shows the number of requests to our API per second. For most services, with GPC, it is possible to enable the API interface at runtime. This means that we can programmatically use this service in production. When we are in production, we can easily identify a problem by simply taking a look at this panel. If, for example, we see a significant drop in the number of requests per second, this can point to a problem in our infrastructure.

On the right, we see the Google Cloud Platform status, which indicates the general status of the platform. It is useful for identifying the general problem on the platform.

In addition to the technical information, the dashboard shows some nontechnical information, such as billing. To access the billing information, click the Billing section on the dashboard.

Understanding Billing in GCP

Billing is an important part of our process, because it is essentially the return on our business. GCP offers a very good platform for understanding billing

and how a bill is composed and read.

GCP identifies billing accounts for the following items:

- *Billing account ID* : This is the unique ID generated by Google associates with our account.
- *Billing account name* : This is the name associated with the billing account ID. We can change this name to be more user friendly.
- *Status* : This indicates the status of the billing, whether closed or open.
- *Number of project* : When we create a billing account, we can link it to more than one project. This is used to have one point of payment but different bills for different customers or projects.

The Billing dashboard also shows information about budgets and alerts (Figure 2-6). We can create a monthly budget and raise an alert when we reach the budget ceiling. This is very useful when we must take control of a project's expenses.

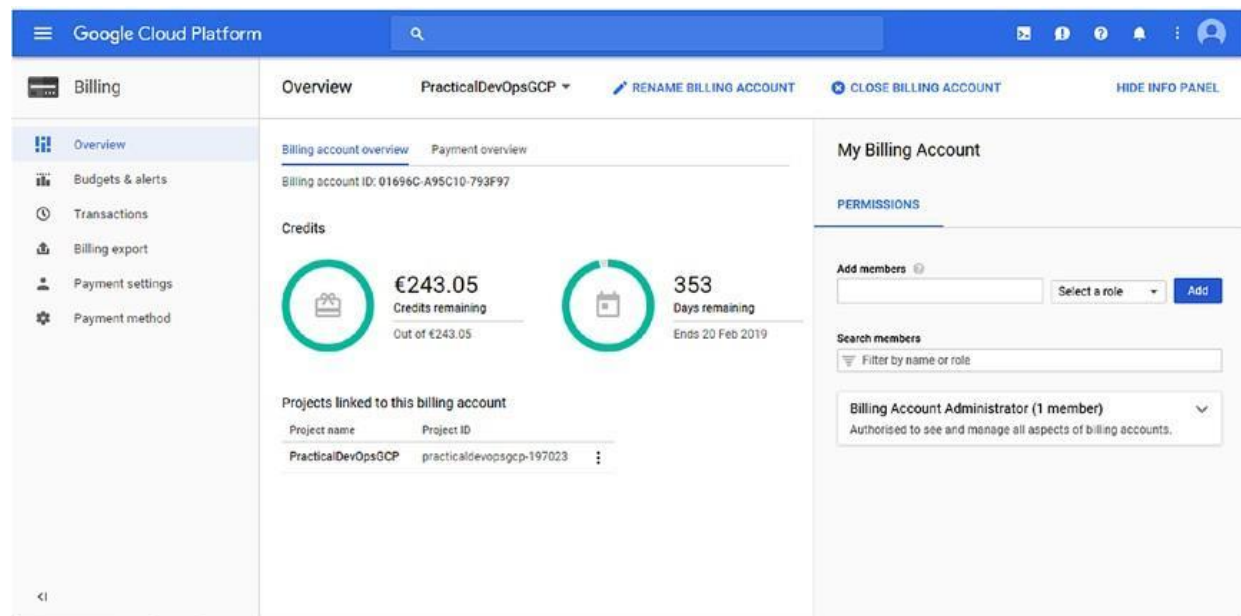


Figure 2-6. The GCP Billing dashboard

By default, only the owner of the Google Cloud account is responsible for managing the billing. We can easily add another user just by clicking the Permissions section on the right side of the dashboard. When we add users, we can select roles for them. All the roles can be managed by the identity and access management (IAM) sections.

The GCP billing dashboard is very clear and manageable. We can easily identify how many projects are associated with the billing account, as well as the payment overview, which shows all the information about payment details. We can easily change this parameter, according to the needs of the company.

GCP Resources

When we talk about the cloud, we are also essentially talking about hardware. In the case of GCP, we “rent” the Google infrastructure.

Google hosts resources in different geographical areas, because this reduces the risk of the system going down. The possibility of a natural disaster or some other problem occurring at the same time in two different locations is minimal. Another significant advantage of having different resource locations is the reduction in latency.

Each of these locations is known as a *region*. A region is essentially a Google data center. In a data center, we find all the resources required to build a Google cloud application. These resources include a physical server, network components, and a virtual machine. The actual region is located in the central United States, western Europe, or East Asia.

Every region is essentially a collection of *zones*. A zone is a deployment area for a cloud platform. A zone should be considered as a single point of failure in our infrastructure.

Because a zone can have a downtime, to ensure fault tolerance and high availability, we must consider deploying our application across different zones and, possibly, different regions. The different regions and zones help us to design for complete fault tolerance and high availability in the cloud.

The zone has a specific name. This is created using the name of the region and a number, which identifies the number of the zone, for example, europe-west2.

Note When we design a cloud application, it is important to think about its availability. This is essentially related to the decision we make about our

deployment model. If we want our service to have high availability, we must use either a regional application, such as App Engine, or a managed multiregional application, such as Cloud Storage. If we want to build disaster recovery for the data that follows this strategy, we use some multiregional-based service, such as Google Cloud Storage or Google Cloud Datastore. If we use a zonal or regional service, we snapshot the data in a multiregional resource. The data should be replicated in a different region or zone. This way, if one zone fails, we will have another zone available. For computing, use the zonal or regional resource, such as Google App Engine, but in case of failure, have a mechanism for spinning up the application in another zone or region. Of course, for total high availability, we must have a load balancer, to balance the resource across the zone or region and connect the data with a multiregional service.

Following are some different types of GCP resources:

- *Regional* : A regional resource is one that can be redundantly deployed across all zones of a specific region. This provides high availability to a zonal resource.
- *Zonal* : A zonal resource operates in a single zone. If the zone becomes unavailable, the resource itself becomes unavailable.
- *Multiregional* : This cloud service is directly managed by Google to be redundant and distributed across different zones and regions. The data stored in a multiregional area is stored across different regions and not in only one region or zone
- *Global resource* : This kind of resource can be accessed by another resource, independent of the zone or region. The global resource is normally a preconfigured disk, snapshot, or network.

Knowing the type of resource is important when we decide to put our cloud architecture in place. This is because it essentially drives the design of the architecture.

When we plan our architecture, it is very important to understand the scope of the operation, based on the kind of resource we choose. For example, if we must create a network, we create a global resource, because it can be

shared across different zones and regions. However, when we assign an IP, this is essentially a zone operation, because the IP address changes, depending on the zone.

When thinking about the cloud, we must make choices based on the efficiency of the architecture. This means that we never use a hard-drive resource from a different region, because the latency is too high. Planning the resource correctly is the key difference between a good and bad cloud project.

Google SDK

GCP offers a good command line interface that we can use to manage our cloud. This software development kit (SDK or devkit) is called Google Cloud SDK. It is available for different operating systems and can be downloaded from the following link: <https://cloud.google.com/sdk/docs/> .

Select the correct SDK for your operating system and install. This is generally an automatic process. When the SDK is installed, we can initialize our cloud environment. When the SDK is installed, we can open the command line and digit the following command:

```
gcloud init
```

This command starts to configure the Google Cloud SDK. When asked to log in, simply type“Y.” This opens the browser. Now log in to GCP (Figure 2-7).

```
Welcome! This command will take you through the configuration of gcloud.

Your current configuration has been set to: [default]

You can skip diagnostics next time by using the following flag:
gcloud init --skip-diagnostics

Network diagnostic detects and fixes local network connection issues.
Checking network connection...done.
Reachability Check passed.
Network diagnostic (1/1 checks) passed.

You must log in to continue. Would you like to log in (Y/n)? Y

Your browser has been opened to visit:

https://accounts.google.com/o/oauth2/auth?redirect_uri=http%3A%2F%2Flocalhost%3A8085%2F&prompt=select_account&response_type=code&client_id=32555940559.apps.googleusercontent.com&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcloud-platform+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fappengine.admin+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fcompute+https%3A%2F%2Fwww.googleapis.com%2Fauth%2Faccounts.reauth&access_type=offline

You are logged in as: [ * * * * * ].

Pick cloud project to use:
[1] practicaldevopsgcp-197023
[2] Create a new project
Please enter numeric choice or text value (must exactly match list item):
```

Figure 2-7. Google SDK initialization

Now that we have seen that we can select to create a new project or use the project we just created, press 1 and complete the Cloud SDK configuration. gcloud is now configured. We can use this tool to access and manage the different resources in our cloud environment, via the command line. It is possible to use the command line from the console as well. To launch the command line, just click the cloud shell button, on the right of the search bar. This command opens a shell directly on the console (Figure 2-8).

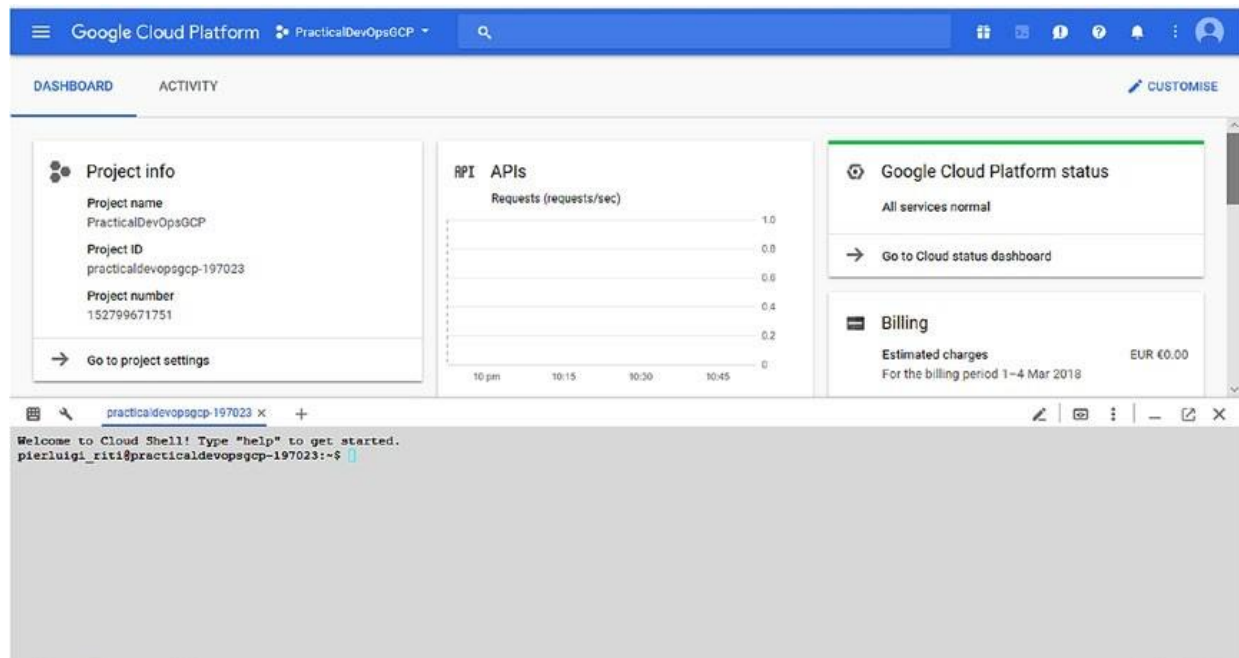


Figure 2-8. Google SDK opened on the console

gcloud has a very comprehensive help system. For example, we can see all the operations we can execute on the computer, using the command `gcloud compute -h`

This produces an output like that following:

```
C:\Users\user\AppData\Local\Google\Cloud SDK>gcloud compute -h
```

Usage: `gcloud compute [optional flags] <group | command>`

group may be `accelerator-types` | `addresses` | `backend-buckets` | `backend-services` | `commitments` | `disk-types` | `disks` | `firewall-rules` | `forwarding-rules` | `health-checks` | `http-health-checks` | `https-health-checks` | `images` | `instance-groups` | `instance-templates` | `instances` | `interconnects` | `machine-types` | `networks` | `operations` | `os-login` | `project-info` | `regions` | `routers` | `routes` | `shared-vpc` | `snapshots` | `ssl-certificates` | `target-http-proxies` | `target-https-proxies` | `target-instances` | `target-pools` | `target-sslproxies` | `target-tcp-proxies` | `target-vpn-gateways` | `url-maps` | `vpn-tunnels` | `xpn` | `zones`

command may be `config-ssh` | `connect-to-serial-port` | `copy-files` | `reset-windows-password` | `scp` | `ssh`

We can obtain more information about the command, by using the command line `gcloud compute --help`. This command essentially produces a man page for the specific command. On this man page, it is possible to find more information about the command and the operations we can perform with it.

`gcloud` can be used in another scripting language, for automating operations. For example, we can use the command to have all the live instances in our project and manage them. This can be part of a Jenkins script for indicating the status of an instance during the deployment and, for example, stop the monitoring for that specific instance. Another important feature on GCP is the REST API, called Cloud API. This is actually the most powerful API on the cloud market. With the GCP REST API, we can essentially execute any operation in the cloud. For example, we can create a virtual private cloud (VPC) or firewall, using the API. The languages supported are

- Java
- JavaScript
- .NET
- Object-C
- Python
- PHP (Beta)
- Dart (Beta)
- Go (Alpha)
- Node.js(Alpha)
- Ruby (Alpha)

With the API, it is possible to integrate any component of GCP.

Conclusion

This chapter presented a short introduction to GCP. Different services were discussed, and we configured the project we will work on for the rest of the book.

Different products related to cloud computing were covered, in addition to how to install and use the basic SDK tool from Google. You saw how

Google Cloud is configured and what a region and zone are. These are important when deciding how to build our cloud project.

GCP offers a range of services. In this book, I will concentrate on the services necessary for DevOps. You will see how to use App Engine and create all the services required to continue our DevOps journey.

CHAPTER 3

Introduction to Continuous Integration and Delivery

Continuous integration (CI) and continuous delivery (CD) grow in popularity every day. This is because they are crucial to reducing time to market and improving the quality of software.

With the practice of CI and CD, every time we release software from a central repository, it is built and released to test. This represents hundreds of deliveries every day. CI and CD are strictly connected, and one is an extension of the other. Both practices have some associated costs and savings. In this chapter, I introduce CI and CD and try to show how they are important to our DevOps journey.

Definition of Continuous Integration

The definition of CI is quite simple. It is a development practice that requires the developer to integrate code in a central shared repository. Every time the developer commits the code, it is integrated with other code and verified by the execution of a test.

CI starts every time we commit code to the centrally shared repository. This means that every time we change something, for example, a label on an HTML page, or a variable, we test the entire solution, because we test the solution at every single commit. We can find errors more quickly and easily

in the build and fix them. What we do at every commit is essentially build the entire solution.

Adopting CI is cheap. Essentially, we only need a server with Jenkins, and we can start to use it. CI can be summarized in three simple phases (see Figure 3-1).

Continuous Integration

Develop Test Deploy

Figure 3-1. *The continuous integration chain*

1. Develop
2. Test
3. Deploy

When we have CI in place, we execute this cycle every time we commit the code in our repository. When we use CI, we create a build every time. This is the essence of CI: we have an entire software life cycle every time we commit the code. There are two schools of thought about that. Normally, CI releases to QA.

What Is Build in a Continuous Integration Scenario?

In a CI scenario, a build is more than compiling software. A build is made up of all the operations required to release the software. A build is essentially a process that puts all the code together and verifies that all work fine.

If we consider a typical project, we can see that different people are involved in different areas. The developer creates the feature and, if necessary, changes the script for the table. The database administrator (DBA) puts in place the script and advises the developer when the database is ready. Starting from that, the developer continues the development.

At the end of development, the software is integrated and tested altogether. This can take weeks of work to complete, and in the case of errors, the software may return to the developer for a fix. This costs time and can

reduce the quality of the software. This is because, in classic project management professional (PMP), a.k.a. Waterfall, methodologies, quality is a result of three variables:

- Scope
- Time
- Cost

If we skimp on any one of the three variables, we can reduce the quality of the software. In a CI scenario, all CI processes start with the commit of the source code in the repository.

A CI scenario can be designed with these simple steps:

1. The developer commits the code in the repository.
2. The CI server pools the repository, downloads the last code, and starts to test. If all tests are passed, the server compiles it.
3. The CI server sends a notification, via e-mail, slack, etc., with feedback about the integration.
4. The CI server continues to pool the repository, to check the new change.

Figure 3-2 shows a sample CI system. Here, we can see that we have a mail server that is used to send the feedback to the developer. Feedback is crucial to a good CI system, because it provides an immediate critique of the build, and the developer can use it to resolve any issues faster.

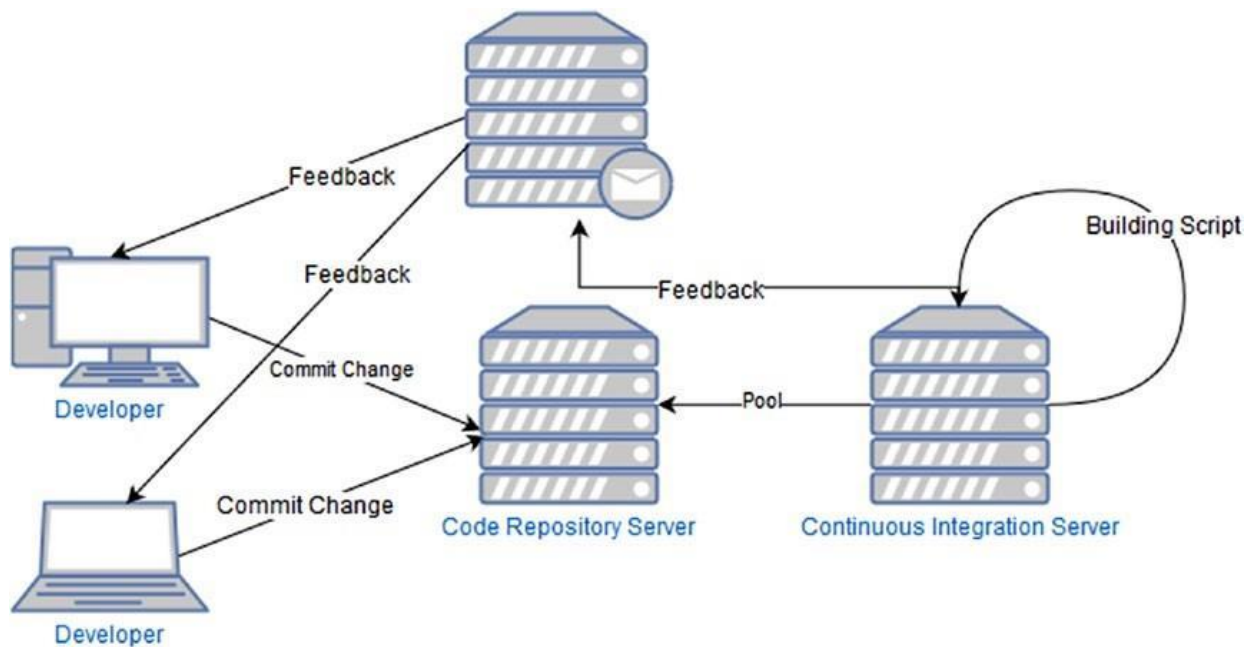


Figure 3-2. *A CI system*

This cycle starts every time the developer commits the code in the repository, which means it can start a hundred times per day.

The Code Repository Server

The code repository server is where we store our software. This is essentially a software for the repo, like Git or SVN. The server can be in-house, meaning that we have an internal server, or external, in which case, we don't manage the server directly, for example, when we put the code in Bitbucket.

A good CI system must have a repository server. This is essentially the starting point of our process. Every time the developer commits, we start the process. We can have many branches in our repo, but only one master branch, which is essentially where we integrate the other branches every time.

The Continuous Integration Server

The continuous integration server is responsible for running the integration script every time we commit the code. We can use different software for doing that, for example, Jenkins, Travis CI, TeamCity, etc.

A CI server executes some specific operations.

1. Retrieves the code from the repository server
2. Merges the last commit with the old software
3. Executes the test on the software
4. Builds the software
5. Sends a feedback with the result

It is not necessary to have a CI server. We can perform this operation with a simple script, such as Bash, Ant, Maven, or Makefile. We can write a simple script to merge and build the software, such as the following:

```
#!/bin/bash
function integrate_code() {
SOURCE=$1
DEST=$2

git checkout $DEST

git pull --ff-only origin $DEST
result=$?
if [ $result -ne 0 ] then

echo "Error in pull"
exit 1
fi

git merge --ff-only $SOURCE --no-edit result=$?
if [ $result -ne 0 ]
then

echo "Error in merge" exit 1
fi

git push origin $DEST result=$?
if [ $result -ne 0 ]
then

echo "Error in a push" exit 1
fi
```

```
return 0
}
```

This script merges the code from a branch with another. It is a very simple script and just a piece of a more complex building system.

When we use a CI server, we can reduce the number of scripts we need to maintain, because a CI server starts a build in an automatic way. For example, we can configure Jenkins to start a build in different ways (Figure 3-3).

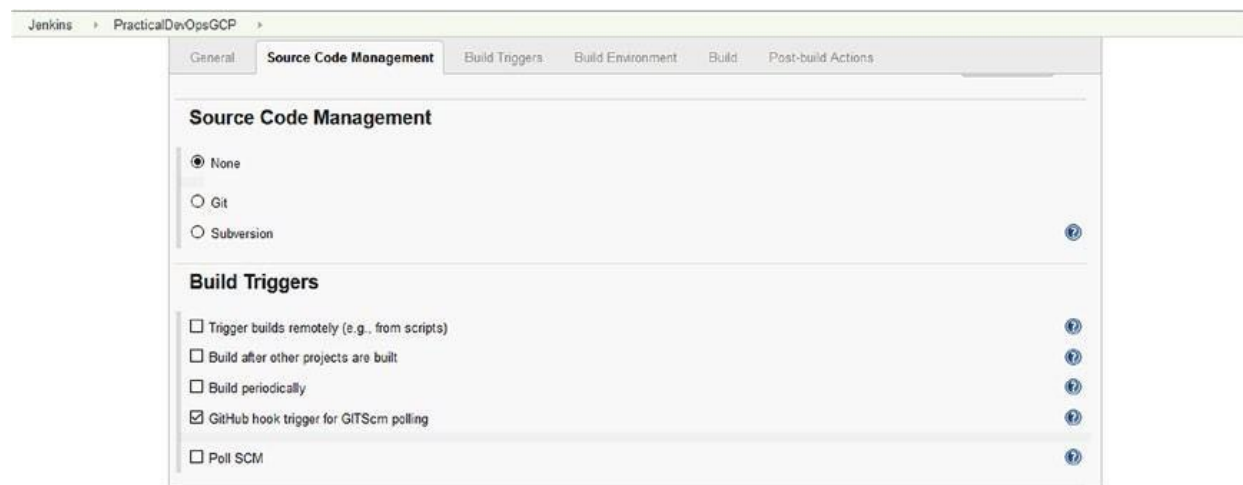


Figure 3-3. The Jenkins build trigger configuration

You can see from the preceding figure that we can connect Jenkins with most of the source control management systems, such as Git, Mercurial, etc., and we can trigger the build with a different option, for example, GitHub hook. In this way, when we commit the software in Git, Jenkins automatically starts a build. By adding to the automatic build, we can build at a certain time or use an external script to start the build.

Note When we use a periodic build, we are not really using a continuous integration approach, because the build does not start when the software is committed to the code. This kind of build policy can be good, for example, when we want to have a daily build that can be integrated with a CI policy.

In addition, with a CI server, we have a dashboard from which we can see what builds are good and what builds failed. This can offer an immediate visual status report on our software.

Continuous Delivery

CD is a software engineering practice used to release software within a short cycle. This means that with every build, we create a new build of the entire software. This does not mean that we release the software to production, but if we want, we can release it. This is the difference between continuous delivery and continuous deployment.

Note We must understand the differences between *continuous delivery* and *continuous deployment*. The concepts are similar, but there is a substantial difference between the two. By continuous delivery, we are referring to a pipeline for creating a build but not necessarily one we intend to release to production. With continuous deployment, we release the build to production every time. This relatively small difference between continuous delivery and deployment makes a big difference to a company's business.

With CD practices, we always have a build ready to use. This allows the QA team to start testing immediately, with a restricted number of features, and give immediate feedback to the development team. This reduces the time to fix the problem and improve the quality of the software itself. Of course, this depends more on the environment. In most of the environments with a CI/CD system in place, we don't really need a QA team to execute the test.

This type of approach helps to reduce costs. Maintaining the software or resolving an issue during the development life cycle is certainly more efficient than fixing a problem when the software has been produced. In addition, with CD, we always test a small part of the software, because CD takes place with every commit, thus reducing the risk of releasing software with a destructive bug, for example.

The idea behind CD is similar to that informing DevOps, but they are two different practices. DevOps concentrates more on changing an entire company culture. Instead, CD concentrates on producing a new software build. However, because DevOps essentially represents a change of culture, CD and CI practices fall within its sphere. CD (see Figure 3-4) is an extension of CI, because CD adds another step to CI. For this reason, if we want to have good CD in place, we must have a strong foundation in CI.

Develop Commit Test Build Deliver *Figure 3-4. The continuous d chain*

Differences Between Continuous Integration and Continuous Delivery

CI and CD are similar, but there are some differences between these practices. CI concentrates on integrating the software with every commit. This occurs after unit testing.

CD extends CI, because it adds another layer after integration and testing of the software are complete. CD builds the software and prepares it for potential release.

CI places a big emphasis on the testing phase. This is very important for CI, in particular when code is merged with the main branch. The goal of CI is not to lose functionality after the merge.

On the other hand, CD places great emphasis on building software. With CD in place, we can decide to release new software on a daily basis. In 2011, Amazon had an average release of new software every 11.6 seconds. This is a huge number of releases per day. With continuous release, we automate any step and process required for achieving this result.

Strategies for Continuous Delivery

To ensure good CD, we must have the following:

- Good branching strategies
- A strong unit test policy
- An automatic testing phase
- Automatic code promotion

All of the preceding practices are strictly connected and help to produce good and strong CD. Some of them are connected to CI, such as branching strategies and unit testing; others are more connected to CD.

Good Branching Strategies

In CI, the goal is to integrate the software with the main branch. With that in mind, we can develop our branching strategies.

The most common branching strategies (Figure 3-5) are to create a branch for every feature/bug we work on. In this way, we can merge single features with the master branch.

Branch Feature 1 Branch Bug Fix 1 Branch Bug Fix 2
Master Branch

Branch Feature 2 Branch Feature 3 **Figure 3-5. Branching strategies in place**

Because we have a different branch for every feature of a bug, we can play with the code without breaking the master line of code. This means that during development, we always have a buildable and potentially releasable line of code.

When we release the software in our branch, we execute the unit test against our branch. We don't test only the feature we develop, but the entire system. In this way, we can have immediate feedback about any error we introduce in the code.

If all tests are passed and the feature is green-lighted, we can start to integrate our branch with the master line of code. When we merge, we start another set of tests. We can also start some code analysis, and if it is green-lighted, we can release a new build with the new feature.

A Strong Unit Test Policy

To be effective, good CD must have strong CI in place, and for strong CI, we must have strong unit testing strategies in place. Unit testing is essential if we want to build a good CI system, because testing can identify an error in what we intend to release.

Unit testing is important not only for identifying the error but because it can be used to validate business requirements. A unit test must be written before development. This means that we must write the code for passing the unit test. This technique is called test driven development (TDD). With TDD, we write the test based on the business requirements and then start to write the code. This ensures a correct correlation between the requirements and the code we release.

TDD is normally connected with a code coverage value. This means that we make sure that a certain percentage of code is covered by the test. A good percentage of code coverage is about 85%. This essentially covers all the code, and we can be quite sure of the quality of our code with this percentage tested.

Another important practice for tests is the *test pyramid*. This phrase is a metaphor used to describe the different granularities of tests in a bucket. The concept was defined by Mike Cohn in the book *Succeeding with Agile*. When we think of a test pyramid, we must include three types of tests.

- Unit tests (the base of the pyramid)
- Service tests (the middle)
- UI tests (the top of the pyramid)

This pyramid helps to test all the important aspects of the software. It is important to have a test pyramid in place. This is because it helps to catch most of the errors and the design of a more reliable system.

An Automatic Testing Phase

Testing is very important to guarantee the quality of the code. In addition to a unit test, we can conduct another type of test. Usually, we have an *integration test*, for check if all the software is correctly integrated with the other components of the system. We can add an *acceptance test*. This kind of test is designed to be executed on the entire system.

When we execute the integration test, we essentially remove all the mock parts of the test and use the real system instead. Normally, we create mock parts in the unit test phase, because we don't yet have any piece of the system ready for testing. For this reason, we create a fake response for that.

The integration test is important for testing the entire system and to validate our integration. In case of any errors in the integration test, we must revert the integration of the code.

The acceptance test is important for reducing the risk of accidentally removing features and having a build that does not align with business

requirements. Usually, the acceptance test is designed by the QA engineering team and is conceived to test any integration with the system. This test normally tests the UI/UX of the system, although it is not intended to test the software itself but, more generally, the system and the features connected with it.

Automatic Code Promotion

Code promotion is the basis of continuous release, because it is used to define what version of the software is ready to be released. A code promotion occurs when the test phase is correctly passed and the code builds without any issue.

Normally, a CI server like Jenkins has the ability to promote the code itself. In general, this is done by tagging the code in a specific way or creating a new branch for release. When we promote a release, what we essentially do is release the code in a different kind of server. For example, we move the code from the development server to the staging server. The different server can be used, for example, by the QA engineering team, for executing some additional manual test.

When we have a CI system in place, usually we have a file to define the artifact. This file describes all the libraries and the relation of every piece of the software. This is described by the term “artifact immutability” and is exemplified by Maven, with which we can define the system and all its dependencies, to install and build the software.

Code Inspection

Another important practice connected to CI and CD is *code inspection*, a.k.a. *linting*. This practice is very important for maintaining a good architect level of code.

This technique is used to explore the code and create a clear picture of what it looks like. We can, for example, identify if the method is too long or complex.

We can use CI to produce a quality code check. With Ruby, for example, we can use RuboCop. This tool analyzes the code and shows all errors identified

in it. In Python, we can use PEP8 to enforce some rules. The use of these rules enhances the quality of the software, because all the development follows some specific rules.

Another important check on the software is the *cyclomatic complexity*. This is a measure used to determine the complexity of a program. It measures the number of independent linear paths through the method. These are determined by the number and complexity of conditional branches. When we have a low cyclomatic complexity, this means the method is easy to read, understand, and test.

Benefits of Continuous Integration and Continuous Delivery

Until now, I have discussed and presented the differences between CI and CD. Both practices have some costs and benefits (see Table 3-1) that we must consider when we adopt their practice.

Table 3-1. *Costs and Benefits of CI and CD*

Practice Continuous Integration

Cost/Change

The developer must write code using the TDD practice. The code must be associated with the unit test.

We need to put in place a new CI server, which must be used to monitor the repo and start the build on every commit.

The developer must integrate the software at least one time per day, which doesn't allow for much difference in integration.

Feedback for every build we make must be in place.

Continuous Delivery For CD, we must have strong CI in place, because CD is an extension of CI.

We must automate all processes for deploying the software and remove the human interaction.

Benefits

Because we test every release, we can reduce the number of bugs we release in production.

A bug is identified soon, which means it can be fixed soon. This accelerates the fix and, of course, saves money

The software is integrated at least one time per day, which means we can have at least one release of the software daily.

With a CI server in place, we can reduce testing time, because we can execute more tests in parallel and then reduce the time to complete the tests.

The QA team can reduce the amount of tests needed to be executed to test a single functionality, which means we can spend more time improving the quality of the software testing real-world scenarios.

CD reduces the time to deliver software, because delivery occurs every time the process ends.

We can improve the number of releases, potentially to more than one release a day, instead of a big release every six months. This accelerates the feedback provided by customers, which can drive our development.

We can see from the preceding table that both practices require some changes. These changes are connected to the way the developer writes the code and how it connects to the infrastructure put in place.

The major cost is essentially creating and maintaining the CI server, because we must configure it for every new feature we add. We can reduce this cost by creating a Jenkinsfile. This is a Jenkins feature that allows us to create a pipeline for CD. Using Jenkinsfile, we can automate and store in the repository our process for the pipeline.

Designing a Continuous Integration and Continuous Delivery System

For putting in place a complete CI and CD system, we must make some changes to our infrastructure and in our architecture. The architecture changes are not directly connected to the software itself but more in the way we produce the software and release it.

The first change we must make is in how we write the code. The developer must start to write the unit test for every single class or function we release, but to be really effective, we must use the TDD technique. This is because, otherwise, we risk writing a test to pass the code, and not to test the requirement we want to implement, which can reduce the benefit of CI.

Another change we must initiate is to force the developer to integrate the software as soon as possible—at least one time per day. Otherwise, we can spend more time integrating the software and the test phase, and fixing a bug can take a very long time.

We must also put in place some rules about the code. We can implement a codeinspection system, using a tool such as *infer*, developed by Facebook, which can check multiple languages, such as C, C++, and Java, and produce a report indicating a line of code with an error. This can help to improve the quality of the software and reduce potential bugs. Other tools, such as PEP8 or RuboCop, work with a specific language and are often used to force some rules regarding the complexity of the method, number of operations executed in the method, number of lines of the method or the class, and length of the lines of code. These rules help to have a readable and maintainable code. This does not directly improve the quality of the code, but it helps to reduce the maintenance time required by the code.

The most important change we must put in place for a CI system and, later, a CD system is an automatic script for building the software. This script must be used to produce some automatic operation to compile and build the software, and, more important, must be able to start from the command line.

To create the script, we can use software like Maven, Ant, or MSBuild, or we can use the simple command-line scripting in Bash or PowerShell. The language is not important, but we must have something that we can start every time we build under the same conditions that always produces the same result.

The most important change in the CI server is the building block of the CI and CD practice. There are a lot of applications we can use for that. Some are free, like Jenkins, others require a license for professional use, such as Travis CI.

To create a good system for CI and CD, we must adhere to some principles.

- Commit the code frequently. Every minor update to the code must be committed and tested.
- Don't break the code with the commit. With the first commit, execute a local build and test, because the code commit doesn't stop the cycle for the CI.
- Develop unit tests. Every commit must be associated with a strong unit test, because we must test to validate the code.
- Create a script for building the software automatically. We must reduce human interaction, which means we have to create a script for building the software and ensure this works every time, to give us the consistency we need for our system.
- Build the software for different environments. With a CD system, software development has different stages. Normally, we have a development server, a staging/test server, and one or more production servers. Every environment has different characteristics and, of course, connection parameters. We must create a system to build software in each of these environments.
- Design pipelines for the software release. To improve quality, we must create a script that automatically promotes the software at every stage.
- Design a strategy to release the software at every stage. Because different stages are involved, we must design a strategy and an architecture for the software release. For example, when the software is built in development and the test is passed, this must be promoted at this stage. By doing so, we can easily create a Docker image, release in the registry, and, by software for orchestration, release it in the stage.

These principles are the foundation for a good CI/CD system. Of course, they must be adapted to specific company needs, but, in general, if we follow these principles, we are sure to reduce human interaction and put in place a good CI/CD system.

Building Continuous Integration and Continuous Delivery Pipelines

To build a good system for CI and CD, we must create a pipeline. With a pipeline, we can define the steps necessary for building the software and eventually release it in production.

When we build software, we can identify different stages. Every stage is responsible for a specific validation of the software. The basic pipeline is composed of three stages.

1. Development
2. Staging
3. Production

When we define a pipeline, we essentially create a system for promoting the software from one state to another, when a certain condition is in place. This process must be managed programmatically, so that it can be easily changed/updated and reduce human interaction. Today, there is a lot of software that we can use for that, for example, GoCD, Travis CI, GitBucket, Circle CI, and Jenkins.

All this software can be used to visually create a pipeline with the different stages we want for our software. Much of it supports some type of scripting language. Having a script for the pipelines is important, because we can save the script in a software repository. If we have to create another environment, we have only to download the script.

For creating such a script in Jenkins, we use a Jenkinsfile. With this file, we can define all the steps we want for our pipeline and, of course, use it to promote the software from one stage to another. An example of a Jenkinsfile (Listing [3-1](#)) follows:

Listing 3-1. A Basic Example of a Jenkinsfile pipeline {

```
agent any
stages {
stage('Build') {
steps {

echo 'Building..' }
}
stage('Test') {

steps {
echo 'Testing..'
}
}
stage('Deploy') {
steps {
echo 'Deploying. .. '
}
}
}
```

From the file, we can define different stages and different agents, which is important for our CI/CD system. The different stages can have different parameters for work. For example, DB connection of system access passwords. In the stage section, we can prepare our system to get this value and change the behavior for response at different stages. Not every CI system has only three phases, but they are a good starting point.

Another important piece of our CI/CD system is the feedback system. This is essentially an e-mail sent out to advise the user about the status of the build. This message is very important, because the developer can react and fix any broken part of the build. The e-mail can be very simple. It must include only the number of the build, the error that has occurred, and the tests failed. This information helps the developer to identify issues and fix them faster. The system must escalate the e-mail if the n-build failed to CC, for example, the team leader.

Continuous Database Integration

When we release software, usually we have a database in which we store the data when we change the software. This can be associated with a change in the data structure, for example, a new table. In this context, it is important to establish a continuous database integration commonly known as *database migration* .

Continuous database integration is the process of releasing and rebuilding all the database and populating it with new data, every time we release the software.

Following this process ensures that we have a database that is always aligned with the last code and, of course, provides a fresh set of test data. Another benefit of this process is that it identifies any issue with the database every time we release the software. To take advantage of this process, we can create and maintain the script for the data manipulation language (DML) and data definition language (DDL). This script must be stored in the software repo, as with every other piece of code in the system, and then reviewed by the DBA team for approval or rejection.

To adopt this process, we can follow these simple steps during our build:

1. Drop the entire database. This way, every time we build the software, we have a fresh new database.
2. Maintain the DML and DDL script in the code repo. The script for creating the database must be in the code repo and integrated every time we release the software.
3. Have a stage for re-creating the database. Because we continuously integrate the database, our pipeline must have a stage for creating and maintaining the database.
4. Have a code review for the DML and DDL script. The DBA team must be aware of any change we put in place in the database, so that we can easily identify any relevant issue. In addition, we must have a code review, to ensure that the update doesn't break the database.

5. Ensure that the test data is always aligned. Because we can change the database structure, we must align the test data to reflect the changes we have made in the database.

When we have automated this process, we can easily fix any issue with the data simply by calling the process for re-creating the database. This guarantees a set of data that is always correct and removes the chance of having an issue connected to wrong data being in the system. Of course, as with every procedure in CI/CD, this must be adapted to your system. Not all systems can have a complete database release every time. This must be a decision based mostly on the system you are working on.

Note Database migration is not always simple. In some cases, this procedure can be very dangerous. For example, if you work in the financial sector, you don't want to destroy and re-create the database every time you release the software.

With modern development frameworks, such as Rails, .NET, etc., we probably use object-relational mapping (ORM). This frequently includes a procedure known as code first, which means we create the code and, based on that, produce the database. In production, this is not recommended. In this case, adopting continuous database integration can be useful for maintaining ongoing control of the database.

Continuous Testing and Inspection

A principal aim a company wants to achieve by building a CI and CD system is to improve the quality of the software released. To ensure this, a good CI and CD system must have continuous testing and continuous inspection in place.

The scope of continuous testing is to create reliable software every time it is released. To achieve this, we create different types of tests that can be executed in an automatic way every time we build the software.

The first type of test we automate is the unit test. This test must be executed every time we compile and build the software. This test is the first point for testing the release and the quality of the software. The unit test can also

check the code coverage of our software. Code coverage is important for understanding what percent of lines of code of our software is covered and then tested.

There is no specific optimal number for the percentage of code coverage, but good code coverage is considered to be between 80% and 90%. What is clear is that to write a good unit test, write to effectively test the functionality and not only to achieve code coverage. The reasons for having code coverage in place are essentially two.

- It improves the quality of the unit test, ensuring that more code is

covered and more bugs are intercepted during the testing phase.

- It allows us to be confident that when we develop a new feature, we are not releasing a new bug into the production environment.

The unit test is only the first step in our testing system. Another test we must include in the system is the *integration test*. This type of test is designed to test the software with the real components. This phase of testing occurs after the unit testing and uses real data to execute it. During the unit test, we can have the ability to mock some data. For example, when we must communicate with an external web service, the integration test combines the different components and tests all the software together.

Note Integration testing is an important phase of continuous integration, because it is responsible for testing the entire system and not only the code we develop. When we start the integration test, we essentially bring a different piece of software together and remove any mock library we use in the test. In this phase, we essentially use real data, to test the system and see if it works well with the new software.

The last phase of testing is made up of the *acceptance test* or *verification test*. This phase of the test is designed by the QA engineering team to test the system from a user's point of view. This means that in the event of an interface, the test is essentially designed on the interface. The goal of this phase of testing is to verify the user requirements and validate them. At this stage, we have the test pyramid build and can easily use it for testing.

In addition to the different testing phases, another important phase is the *code inspection*. This phase is a check of the code, using a set of rules to produce a report on the software itself. Code inspection can be split into two different phases.

- *Code review* : This phase is in place before the final integration.
- *Static code analysis* : This phase occurs when we integrate the software.

The code review is the first phase of the code inspection. During the code review, the code must be approved by another developer first, to be integrated into the main branch. The other members of the team review the code and leave feedback about it. The developer of the code takes the notes and adjusts the code, based on the comments and asks for another review. When all comments are addressed, the code can be approved and finally merged in the main branch.

The static code analysis is made up of two different phases. The first occurs when the developer executes the local commit. During this phase, the code can be validated by some rules. These rules check, for example, the following:

- The complexity of the method
- Lines for every method
- Number of characters per lines
- Comments on the method or class

There are more tools available for making this analysis, for example, PEP8 in Python, and different languages can have different types of rules applied to them.

The other type of analysis we make of the code is the static code analysis. This analysis has the goal of highlighting issues connected directly with the code. There are different tools for doing that, but it is normally executed by automated tools. This is important for identifying potential runtime bugs that can appear in the code and to fix them before release to production. When all the analysis and tests are executed, the code can finally be built and prepared for release.

Preparing the Build for Release

The last step in a CI/CD system is to prepare the release for the build. This follows some simple rules.

- Identify the code in the repository.
- Create a build report.
- Put the build in a shared location. For most modern software, we can have a nexus for our artifacts that allows us to rebuild the software in every system.

These basic rules can be used to identify the software ready for the build. A ready build can be released faster into production, or in our QA environment.

Identifying the Code in the Repository

Identifying the code in the repository is important for understanding when we have a production-ready build. We can identify the code in different ways.

- *Create a label in the repo* : The fastest way to identify the last build code in the repo is simply to identify the code.
- *Tag the code* : A more complex way of identifying the code is tagging. This means creating a tag in the repo with the value for identifying the version.
- *Create a branch* : Another way to identify the code is to create a new branch. This is similar to the tagging technique, only we use the branch instead.

To identify the code, we must create software with a unique name. To create a name, we can use a naming convention such as the following:

PracticalDevOpsGP.1.1.0

The naming convention we create uses this syntax: *<feature>.<major release>.<minor release>.<build number>* . When we build, we essentially change only the last version of the number, for example,

PracticalDevOpsGP.1.1.1, 1.1.2, etc. As with every other feature, this must be created by the CI/CD system in an automatic way.

Creating Build Reports

Build reports include important information we disseminate after the build. This report must consider staff other than technical personnel, so that what is in the report can be universally understood.

For build reports to be effective, they must include the following information:

- Name of the release
- Feature released
- Where to get the release

The report can take the form of an auto-generated message, produced by the CI/ CD system and sent to a specific list of users every time the release is ready. The field can easily be included during the build. The trickiest part can be the feature released, but data related to this can be gathered simply by connecting the CI system with the system for maintaining and designing the software. For example, by connecting Jenkins with Jira, we can determine what task we are still working to connect. In this way, when Jenkins receives the code for the build, it can include a description of the feature, which can be added to our report.

The report can be used to identify the feature and, of course, by the QA team, to identify any discrepancy in the feature planned to be released and the one effectively released.

Putting the Build in a Shared Location

When we finish creating the build, we must share it with other teams. Where we put the build depends on the policy we use to release the software.

If, for example, we release a WAR or an MSI file, we can put the software directly in a shared server. If, for example, we want to create a Docker image, this image must be published in an internal registry used to retrieve the last image to build.

What we must keep in mind is a very simple concept: the immutability of the build. When QA tests a specific build version and validates it, we must release exactly the build used in the QA. The system doesn't have to make another build; it just uses the file passed in QA for release.

Releasing the Build

Releasing the build is the last phase in our CI/CD system. The build release is not intended to be solely for production but can, for example, be used to restrict the number of servers, specular to the production server, on which some customers can try out new features. This type of server is known as a *canary server*.

Note Canary servers restrict the number of servers that allow customers to use a feature. The use of canary servers is to reduce potential bugs in production and to obtain real feedback about the release. Because the canary server is the same as that used in the production release, we can use it to gain feedback about the quality of the software. For example, we can see how the software works in a realtime environment and intercept memory leaks and other bugs raised in production.

Another important consideration is how we release when we release in a cloud SaaS. We don't want customers to have any interruption in the usage of the software. To do that, we must identify a specific way of releasing the software and ensure the reliability of the software itself.

In other cases, we can schedule software maintenance windows. At a specific time, we essentially stop the functionality of our software and release a new version of the software itself.

Note With CD, we don't really have to release the software in production every time. This feature is part of CD, which is important to know. To release the software without interrupting functionality, we can release it with some specific procedures. The most commonly used include

- Blue/green deployment
- Canary deployment
- Incremental deployment

The main goal of these techniques is not to interrupt the functionality of the software and intercept potential problems with the infrastructure and the software as soon as possible.

Blue/Green Deployment

Blue/green deployment is a technique for releasing software that reduces the risk of downtime. It is called “blue/green” because we release two production environments: one called blue and another called green.

With blue/green deployment, we have only one live environment. The system for CD releases the new version of the software in the environment that is not live. When the software is ready and tested, it is installed in the other environment and then switched in production. With blue/green deployment, we essentially have two similar environments, and we just switch between the two (see Figure 3-6).

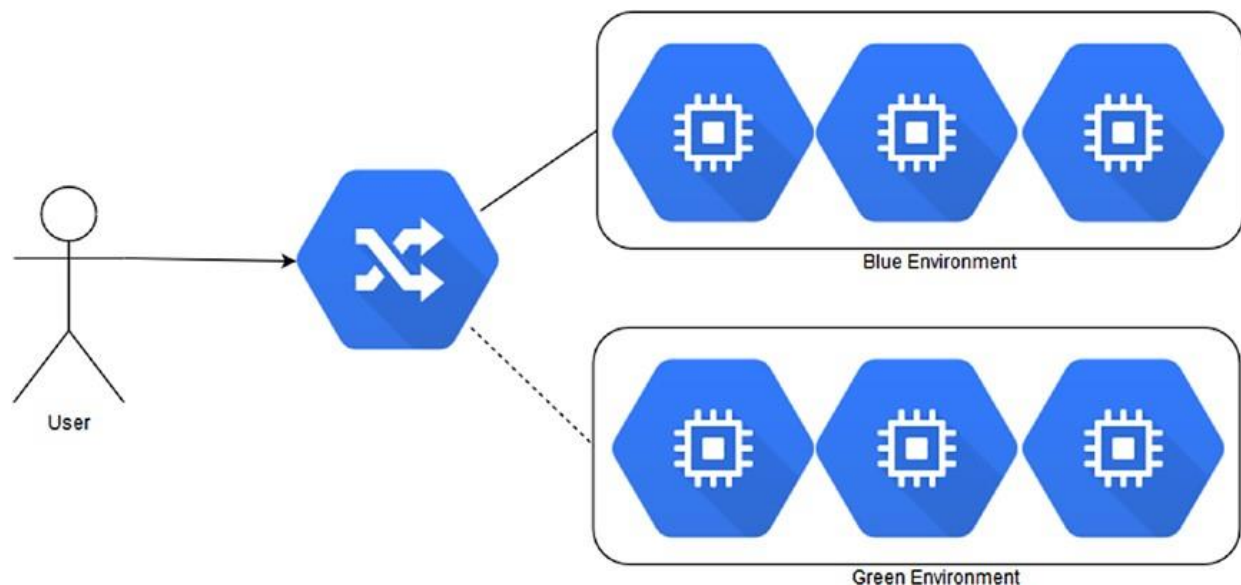


Figure 3-6. Blue/green deployment

Blue/green deployment has some benefits and some costs. With blue/green deployment, we can easily roll back the environment, in case of error, because we always have an environment ready for production.

The cost is connected with some architectural design that we must bear in mind. The first concerns the database. When we release the software, we

may have to modify the table, before proceeding with blue/green deployment. First, we must release the database. When we have released the database, we can then switch the environment.

Another important point we must keep in mind is the user session and other data that can be used by the software. We must have a cache common to the environment, in order not to lose this information and allow its use without any issue.

Canary Deployment

Canary deployment is intended to reduce the risk connected with the release. We release the software in a small part of the infrastructure, which means only a small percent of customers is touched by the release. In case of failure, we can easily roll back the release. The release is intended to be incremental in terms of users. We increment the number of users after a certain time, so that we don't reach 100% (see Figure 3-7).

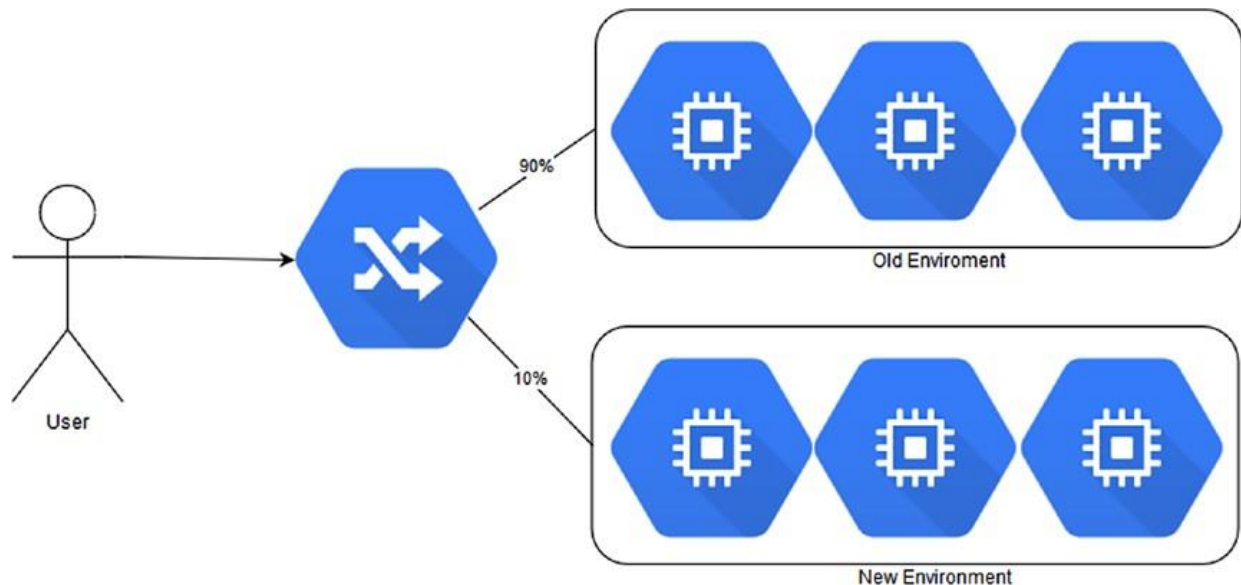


Figure 3-7. *Canary deployment*

This type of deployment can be associated with blue/green deployment. The difference is in how we switch the infrastructure. We create our new environment, and when we are satisfied, we start to release a subset of the users in the new server.

Canary deployment is used to provide immediate feedback about the deployment from a restricted number of users. This can help to identify and solve an issue without a complete rollback, because we release only to a restricted number of users. In the event of a rollback owing to any issue, we can just release to a small number of servers.

Another benefit of canary deployment is connected to the slow ramp-up in the number of users. When we release new functionality, a slow ramp-up of users is preferred for analyzing the use of the memory and other issues connected with the functionality. At the same time, the allows the chance to create specific monitoring values for the software.

Incremental Deployment

Incremental deployment (Figure 3-8) is used when we want to have only one hardware in production. By this technique, we release to only a percentage of users at a time, for example, to 5% of users. When we are satisfied with the first release, we move to another set of users.

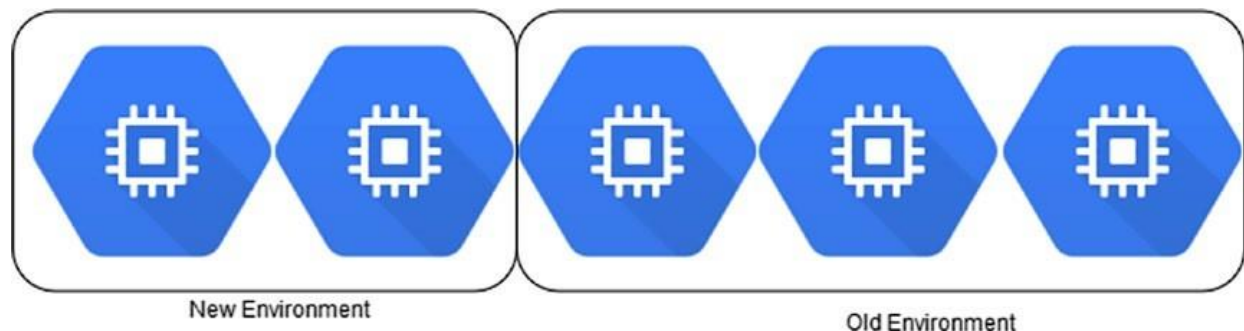


Figure 3-8. *The incremental deployment process*

The incremental deployment process is used with only one line of hardware. This is because only a small part of the software is used at a time. The benefit of this type of deployment is connected with the small amount of servers we release to.

Because of this, we can monitor the feature better, immediately identify any issue with the software, and adapt the infrastructure or the code to fix the issue.

Conclusion

In this chapter, I discussed CI and CD. Both techniques are at the core of DevOps and are the basis of the cloud deployment.

CI and CD are two closely connected practices. This is because one is the evolution of the other. It is important to spend the time to fully understand these practices to correctly put them in place.

Both practices require a complete change in the way we develop software and how we design the software. We must exert some pressure on the developer to follow the guidelines for CI and CD.

On the other hand, a good system for CI and CD improves the quality of our software and helps to increase release to more than one or two times a year.

CHAPTER 4

Containerization with Docker and Kubernetes

Today, when we think of the cloud, we think of containerization. Containerization can be seen as an evolution of virtualization. With virtualization, we usually re-create an entire operating system (OS) and host it on a host machine.

Using a software for the container, such as Docker, it is possible to create a complete image of our applications and release it via the common registry. To manage and release this image, we can use software known as a container orchestrator, such as Kubernetes.

When we adopt the practice of CI and CD, using such containers as Docker and orchestrators such as Kubernetes helps to speed up the automatic process to release, and at the same time, we can have a strong policy for rollback.

Introduction to Docker

Docker is probably the most famous software for containerization. It offers a level of virtualization called *operating-system-level virtualization*. This is known as containerization.

This type of isolation allows us to run more than one OS inside another. For example, it is possible to create a container inside an Ubuntu Linux with Red Hat. There is an important distinction between a container and a virtual machine (VM), which is that a container doesn't require a full operating system for run. When we create a VM, we essentially re-create an entire OS. When we create a container, we get only a part of the operating system. This reduces the size of the image. When we talk about virtualization, we can identify two types: *hypervisor-based virtualization* and *operatingsystem virtualization*. In hypervisor-based virtualization, the system emulates the hardware. This means that we can re-create the network, the hard disk drive (HDD), etc. In operating system virtualization, the virtualization is made at the operating-system level. The host isolates each container from the other, in particular, the host isolates the filesystem of each container, but they run in a single host. Because the container has a filesystem isolated OS, a container loses flexibility. You can run containers only with the same host. It is not possible to run Windows on a Linux host, for example, because the OS virtualization is run by the systems. Windows and Linux have two different OS kernels and filesystem structures, which does not allow Windows to run on Linux.

Containers suffer from less security compared to hypervisor virtualization. This is because when we create a Linux container, we use libcontainers. These access five basic namespaces—Network, Process, Mount, Hostname, and Shared Memory—but, for example, don't use SELinux, cgroups, and other libraries used to enhance OS security. This translates to a greater possibility of malicious software preventing execution and operation. Conversely, a container is an isolated environment. This means that in the case of a breach on the container, it is not likely to create an issue on the host system, because a container doesn't share anything with the hosting OS.

Docker containers are complex to create and difficult to maintain and automate. Docker is essentially software designed to create other software, based on system operation virtualization. Docker adds an application deployment on the top of the virtualized container. With Docker, it is easy to create a complete runtime environment similar to the production one, and this can speed up the development process.

Docker helps developers to eliminate the discrepancy that can result when software is developed and released in production. This is because when we release the software in production, a Docker container works in the same OS configuration used to develop the software.

With Docker, we can easily create our process for CI and CD, because when we commit the code, we can create and compile the Docker image directly and release it in the test.

From an architectural point of view, with Docker, it is easy to implement a microservice architecture. This is because any container can be a single piece of the application, and we can manage this single piece independently of another.

Docker is composed of the following different components:

- The *Docker engine* is a client-server application. The client talks with the server application, called a daemon. The daemon is responsible for executing the container. Client and daemon can be on the same machine or different ones.
- *Images* are the basis of our Docker architecture. Images are used to launch our container, and we can create personal images, starting with another basic image.
- The *registry* is where the image is stored. We can identify two types of registries: private and public. Public registries can be reached via the address <https://hub.docker.com>. This is the official Docker Hub registry. Here, we can create an account and start to store our images in the registry.
- *Containers* are basically an image that is executed. A container can have more than one running process inside, depending on how the images are designed and created. A container is essentially a lightweight stand-alone

application. We can combine more than one container to execute a complex application.

Why Use Docker?

Docker becomes more popular with the cloud, because it is very useful when we have to build a PaaS cloud. The reason is simple. The concept of containers allows us to create a layer of different isolated containers with a single application inside. We can easily build a PaaS specific to a single customer. This allows great flexibility in all aspects associated with the release of the design of our PaaS.

Docker also can be used to

- Create an environment for the developer like that for production. Docker can have the same software stack as that used in production.
- Create a building block for service-oriented or microservice architecture. With Docker, we can easily isolate the application and use that to build our microservice or service-oriented architecture (SOA).
- Create a system for CI and CD. We can use Docker to isolate the application and easily deploy in a test environment or even in production.
- Create a stand-alone light application environment for testing and experimentation.

These are only a few examples of adopting Docker. More companies adopt Docker to accelerate the release or deployment processes. Modern integrated development environments, such as Visual Studio or IntelliJ, now have a plugin to “dockerize” our application. This makes it easy for developers to create Docker images every time they build the software.

Using Docker in Google Cloud Platform

The best way to learn something is to get your hands dirty, so the best way to learn Docker in Google Cloud Platform (GCP) is to use it.

With Docker, we can create our SaaS and our IaaS, because we can use it to provide our infrastructure. This is because we can create a container, for example, for our database, and our application, for example, our Java- or Ruby-based web site. In GCP, we can create Docker containers using Google Compute Engine. First, to understand how to create and use Docker, it is important to understand what it is and how to create an instance in Compute Engine.

Introduction to Google Compute Engine

Google Compute Engine is the IaaS component offered in GCP. The Google Compute Engine is composed of three basic components:

- Virtual Machine
- Network component
- Persistent Disk

With Google Compute Engine, we can create a workflow for scaling from a single instance to a globally distributed instance. When we create a Google Compute Engine, we can choose any configuration in terms of CPU, memory, and space. Google Compute Engine also makes it possible to use predefined instances. These are

- *Standard* : This kind of machine has 3.75GB of memory per CPU. We can choose up to 8 kinds of processor configurations, from 1 to 96. The maximum number of persistent disks we can use is 16.

This configuration is ideal for tasks requiring a good balance between CPUs and memory.

- *High-memory* : This kind of configuration has 6.5GB of memory per CPU. We can choose a minimum of 2 CPUs, up to 96. The maximum number of persistent disks we can use is 16. This configuration is ideal when more memory than CPUs is required.

- *High-CPU* : This kind of configuration has 0.90GB of memory per CPU. We can choose a minimum of 2 CPUs, up to 96. The maximum number of

persistent disks we can use is 16. This configuration is ideal when more CPUs than memory is required.

Note There is some limitation when we create a new persistent disk in Compute Engine. The maximum size is 64TB for persistent disk. If we have to create a larger disk, we must create a cluster of persistent disks. Every instance has a limit of 16 disks that can be connected. There is a beta feature, for example, in which the limit of the persistent disk can be connected to every instance. For a shared core, it is possible to attach only 16 instances. For 1 CPU, it is possible to attach up to 32 disks. For between 2 and 4 CPUs, it is possible to attach up to 64 disks. With 8 or more CPUs, it is possible to attach up to 128 disks.

The core component for Google Compute Engine is an *instance* . This is a VM that is hosted on the Google infrastructure.

It is possible to create an instance provided by Google, based on Linux, on a Windows server, or it is possible to create or import a custom image, by running a Docker image. In this case, Google offers an optimized OS for running the Docker image. This operating system is based on the Chromium OS.

Creating a Compute Engine Instance

To create a new Compute Engine, we can use Google SDK with the `gcloud compute`, or we can use the console. To create the instance, we must follow these two steps:

1. `gcloud`.
2. Select the region and zone.

The first step, configuring `gcloud`, creates the OAuth2 to authenticate and access to the resource. To use `gcloud compute`, first we must configure Google SDK for creating the token necessary for the authorization.

For configuring Google SDK, we must open the command line and use the `gcloud init`. This uses the default configuration and is good for most of the configuration. Open Google SDK and run the command `gcloud init`, which gives the following output:

Welcome! This command will take you through the configuration of gcloud.

Settings from your current configuration [default] are:

core:

account: pierluigi.riti@gmail.com

disable_usage_reporting: 'False'

project: practicaldevopsgcp-197023

Pick configuration to use:

[1] Re-initialize this configuration [default] with new settings [2] Create a new configuration

Please enter your numeric choice:

In this case, select the number 1, then press Enter. We require this additional output:

Your current configuration has been set to [default]

You can skip diagnostics next time by using the following flag: gcloud init --skip-diagnostics

Network diagnostic detects and fixes local network connection issues.

Checking network connection...done.

Reachability Check passed.

Network diagnostic (1/1 checks) passed.

Choose the account you would like to use to perform operations for this configuration:

[1] pierluigi.riti@gmail.com

[2] Log in with a new account

Please enter your numeric choice:

Select the mail we want to use for configuring Google SDK. We can use the user we are logged in as or log in a new account.

You are logged in as: [pierluigi.riti@gmail.com].

Pick cloud project to use:

[1] practicaldevopsgcp-197023

[2] Create a new project

Please enter numeric choice or text value (must exactly match list item):

Select the project we want to use or create a new one. In this case, create a new one, called `practicaldevopsgcpcli`. To create this, select option 2, which will give an output like the following:

Please enter numeric choice or text value (must exactly match list item): 2

Enter a Project ID. Note that a Project ID CANNOT be changed later. Project IDs must be 6-30 characters (lowercase ASCII, digits, or hyphens) in length and start with a lowercase letter.

Insert the name of the instance and press Enter.

Your current project has been set to: `[practicaldevopsgcpcli]`.

Not setting default zone/region (this feature makes it easier to use `[gcloud compute]` by setting an appropriate default value for the `--zone` and `--region` flag).

See <https://cloud.google.com/compute/docs/gcloud-compute> section on how to set

default compute region and zone manually. If you would like `[gcloud init]` to be

able to do this for you the next time you run it, make sure the Compute Engine API is enabled for your project on the <https://console.developers.google.com/apis> page.

Your Google Cloud SDK is configured and ready to use!

- * Commands that require authentication will use `pierluigi.riti@gmail.com` by default

- * Commands will reference project ``practicaldevopsgcpcli`` by default

Run ``gcloud help config`` to learn how to change individual settings

This `gcloud` configuration is called `[default]`. You can create additional configurations if you work with multiple accounts and/or projects. Run ``gcloud topic configurations`` to learn more.

Some things to try next:

- * Run ``gcloud --help`` to see the Cloud Platform services you can interact with. And run ``gcloud help COMMAND`` to get help on any `gcloud`

command.

* Run ``gcloud topic -h`` to learn about advanced features of the SDK like arg files and output formatting

In the end, we can log in to the console, and in the section for selecting the project, we can see the new project created (Figure 4-1).

Select



Figure 4-1. The new project created with the command line

This command is used to create a new project. Now that we have set the project, the next step is to create the new instance. Because we have two projects, we must set the default project.

```
gcloud config set project practicaldevopsgcpcli
Updated property [core/project].
```

This configures the main project to the new project we have just created. To use the command line, we must enable the API for the project. For the link, just create the following command on the console:

```
gcloud compute instances create example-instance --zone us-central1-f
```

Because the API is not enabled the output is

```
ERROR: (gcloud.compute.instances.create) Could not fetch resource:
- Project 377223342623 is not found and cannot be used for API calls.
If it is recently created, enable Compute Engine API by visiting
https://console.developers.google.com/apis/api/compute.googleapis.com/
overview?project=377223342623 then retry. If you enabled this API
recently,
wait a few minutes for the action to propagate to our systems and retry.
```

For enabling the API, just get the link and put this in the browser, then enable the API. When the API is enabled, the console shows an option such as Figure 4-2 .

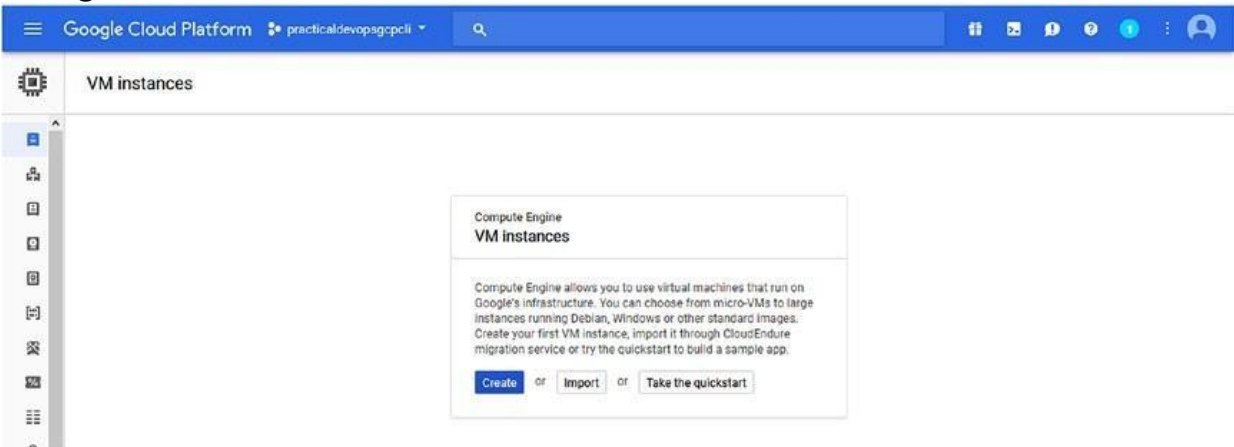


Figure 4-2. The console used to create a new Compute Engine

We can finally create the instance, using the `gcloud` command.

```
gcloud compute instances create practicaldevopsgcp --zone us-east1-b
```

The output is

```
Created [https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcpcli/zones/us-east1-b/instances/practicaldevopsgcp].
NAME    ZONE    MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP  STATUS
practicaldevopsgcp  us-east1-b  n1-standard-1  10.142.0.2   35.227.53.98  RUNNING
```

The instance is finally live. The result shows the information for access on the instance with the internal and the external IP. It is possible to check the instance created in the console. Simply connect to the dashboard and see the new instance (see Figure 4-3).

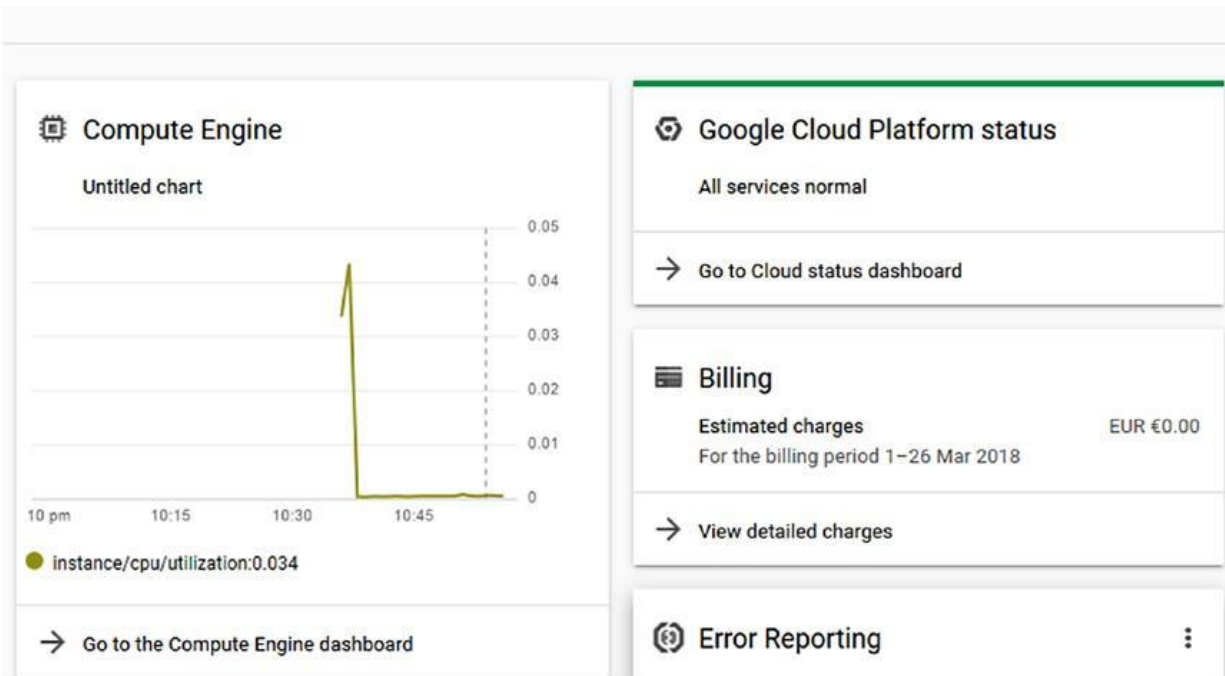


Figure 4-3. The dashboard where it is possible see the compute engine use

For connecting to the instance we have just created, we must go to the Compute Engine dashboard. Under the Connect column, select to connect to the new browser. This opens the connection in the new page (Figure 4-4).



Figure 4-4.

Selections to open the connection on the instance

Select the Open in browser window, to open the connection in another browser window (Figure 4-5).



Figure 4-5. The connection on our instance, open in another browser

We just completed the first instance. The basic instance is created with Debian Linux by default. We can install Docker in the system, or simply create a different instance using a specific image for Docker. Google has designed four kinds of pre-built images for Docker.

- *Containers-Optimized OS from Google* : This is a Google image created directly, based on the project Chromium OS. This image includes Docker and Kubernetes. The image project is `iscos-cloud`, and the image family is `iscos-stable`.
- *CoreOS* : This is a Linux-based OS designed for containerization. It includes Docker, rkt, and Kubernetes. The image project is called `coreos-cloud`, and the image family is `iscoreos-stable`.
- *Ubuntu* : This is an image based on the Ubuntu version 16.04 LTS that includes the LXD. The image project is `isubuntu-os-cloud`, and the image family is `isubuntu-1604-lts`.
- *Windows* : This is Windows' core version designed per the containerization, which includes Docker. The image project is `windows-cloud`, and the image family is `iswindows-1709-core-forcontainers`.

Because we want to create a Docker image for our system, we now create a new instance, based on one of the Docker images.

Instance Group

In Compute Engine, we can create a named group of VM. When we create a group of instances, we can manage all the instances at the same time. We can create two types of instance groups: *managed* and *unmanaged*.

When we create a managed instance group, we use an *instance template* . This is used to create a group of identical instances. With a managed instance, we can manage all the instances as one, which means when we modify the instance template, we modify all the instances at the same time. A managed instance group has several benefits.

- *Autoscaling* : When the application requires more resources, we can scale the instance to fit the new requirements.
- *Load balanced* : Because all the instances are managed as one, the resource is shared and balanced among the instances. Of course, we must create the load balancer on top of that to ensure the functionality.

- *Management of the unhealthy instances* : In the event an instance in the group is stopped or crashed, it is automatically re-created with the same name as the previous instance.

There are two types of managed instance:

- *A zonal managed group* : All the instances are based in the same zone.
- *A regional managed group* : All the instances are based in the same region.

Google suggests using a regional managed group instead of a zonal one, to have the application spread across the different geographical areas. Spreading the application across different geographical locations reduces the risk of faults, as very different geographical zones can experience faults caused by natural disasters or human error at the same time. Thus, regional management can enhance availability of our software.

The unmanaged instance is not recommended by Google, and it doesn't have any of the benefits of a managed instance. This group is made up of dissimilar instances and can only use the load balancer. This can be used only when there is some old instance, and we want to use a load balancer on it.

Container Application in Google Cloud Platform

A container is probably the best way to release and put a microservice into production. To create a container in GCP, we can construct a new Compute Engine with one of the container-optimized OSs. This is a family of OSs optimized for the run container in Compute Engine. For example, CoreOS is one of the operating systems.

Note CoreOS is a lightweight operating system, based on Linux Kernel, that is designed to provide an infrastructure for the cluster. CoreOS has a minimal set of commands and automates all the jobs related to the container application. With CoreOS, it is easy to deploy and maintain containerized infrastructure programmatically, and because it is cluster-based, we can be sure our application is always reachable.

We now construct a new Compute Engine with a CoreOS operating system and then create a container in that. The following is one of the ways to create a container application in GCP.

1. Connect to our Google Cloud Platform instance and select the `projectpracticaldevopsgcpcli`.
2. Click the board Compute Engine, which opens the page to manage our Compute Engine instance (Figure 4-6).

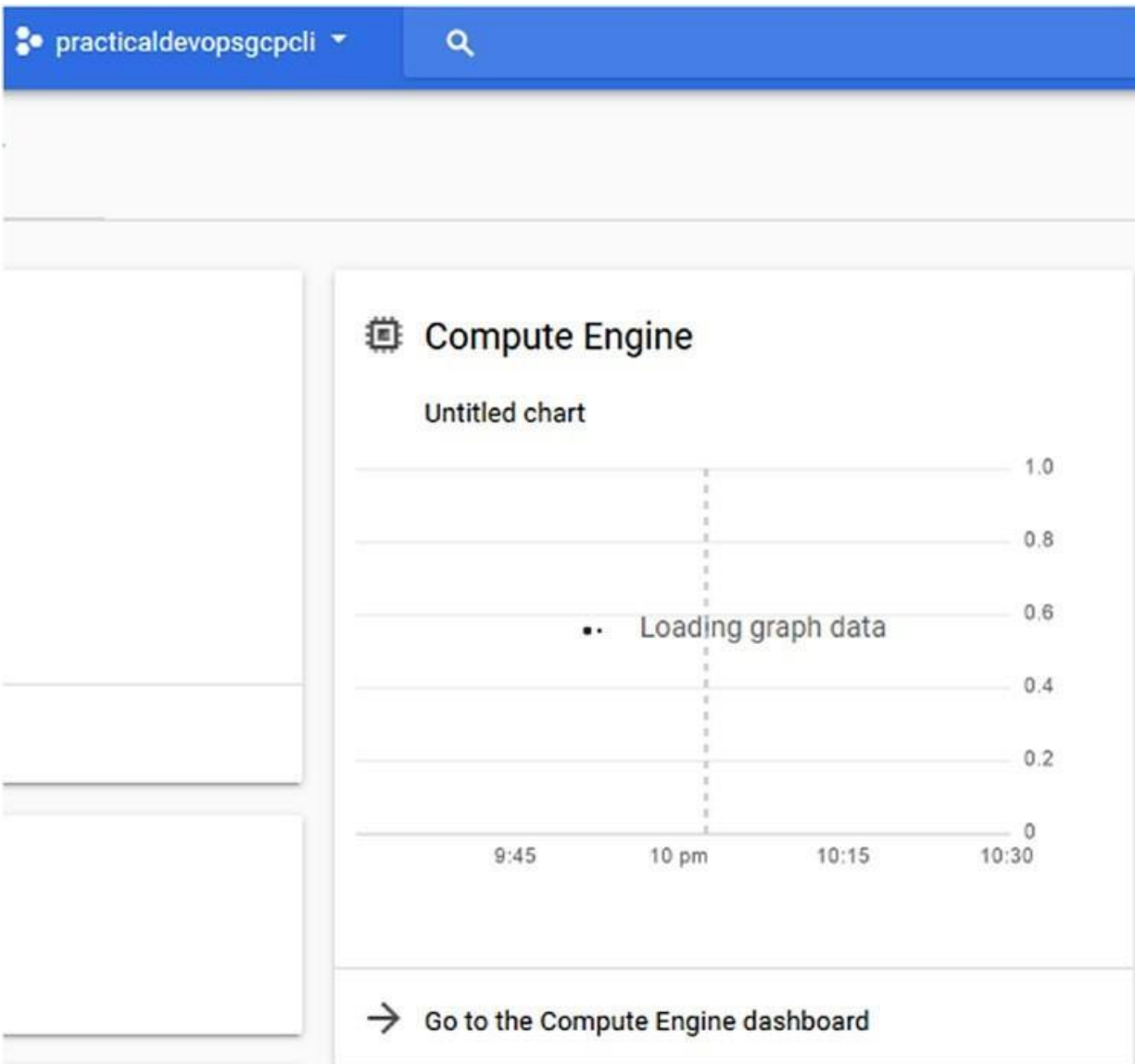


Figure 4-6. The Compute Engine board, an extract from the Google Console dashboard

3. This opens the page of our Compute Engine. We must now create a new instance, based on one of the container-optimized OSs. Select the Create Instance button from the toolbar.

4. In the page for creating the instance, type the name *practicaldevopscontainerinstance* and check the Container box (Figure 4-7). This changes the default OS from Debian to one that is container-optimized.

[←](#) Create an instance

name

practicaldevopscontainerinstance

Zone

us-east1-b

Machine type

Customise to select cores, memory and GPUs.

1 vCPU

3.75 GB memory

Customise

Upgrade your account to create instances with up to 96 cores

Container


☒ Deploy a container image to this VM instance. [Learn more](#)

Container image

For example, gcr.io/google-containers/busybox

Advanced container options

Boot disk



New 10 GB standard persistent disk

Image

Container-Optimized OS 65-10323.64....

Change

Figure 4-7. *Creating a new container instance*

5. In the container image text box, we can identify the Docker image we want to use. For our test, we can use a busybox image. In the text box, we must insert the following string: [gcr.io/googlecontainers](https://gcr.io/google-containers/busybox) /busybox.

Note Google provides a private registry for the Docker container. This registry can be used to store our private Docker images and for creating a new Instance. It is possible to see all the images in the registry via this link: <https://console.cloud.google.com/gcr/images/google-containers/GLOBAL> .

6. Click the Create button and then create the new Compute Engine instance with the Docker image (Figure 4-8).

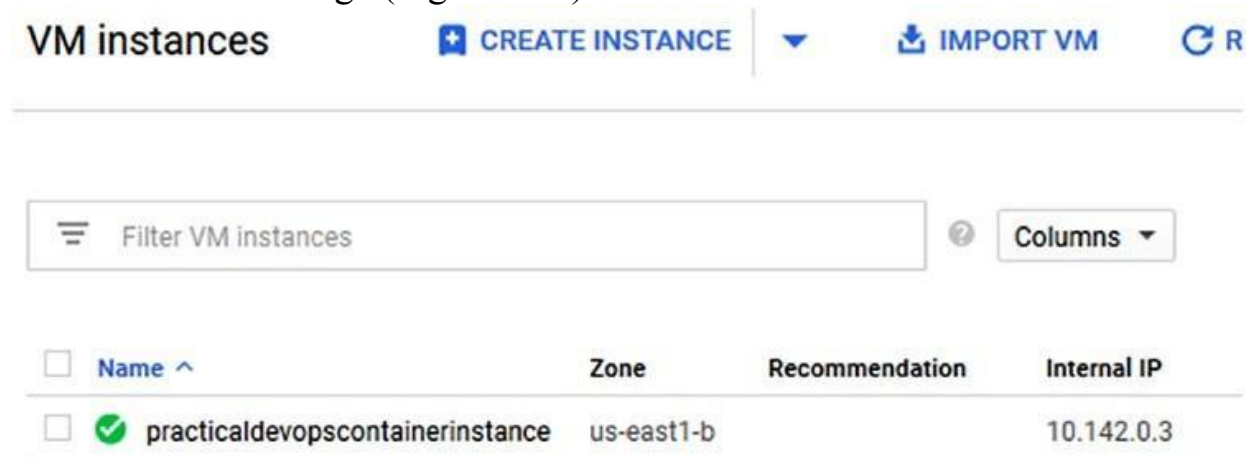


Figure 4-8. The new instance is created with Docker inside

We can access the new instance just by clicking the name. This opens the details of the instance. Then scroll down to Remote access and click SSH (Figure 4-9).

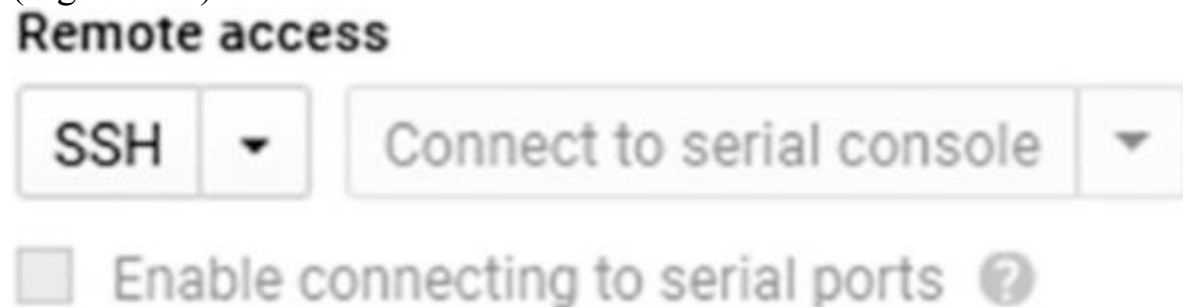


Figure 4-9. The Remote access selection

This opens new browser windows for the instance with Docker and our image. We can see the image actually installed, with the commanddocker images. When we execute the command into the instance, we get the output shown in Figure 4-10 .

```

Connected, host fingerprint: ssh-rsa 2048 3B:74:79:60:09:AB:2E:59:7B:13:93:2F:13:5C:2C:2E:33:4D:34:54:39:DD:11:3
9:C6:B9:F7:20:4E:7E:6B:AF
pierluigi_riti@practicaldevopscontainerinstance ~ $ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
gcr.io/gce-containers/konlet    v.alpha.0.7-latest  4480f8a9c9b6       4 months ago       105 MB
pierluigi_riti@practicaldevopscontainerinstance ~ $

```

Figure 4-10. The Docker images command output

Creating a Compute Engine instance in which we create our container is a good starting point. Google offers a private registry, and we can upload our images to the registry and use that in our instance. Using the Google registry, we can put in place our CI/CD basic systems. For example, we can use Jenkins to create the image and put it in the registry.

Using a Compute Engine instance to create our Docker image does have some limitations.

- It is possible to deploy only one container in the VM.
- It is possible only to use an optimized-container OS to create the instance.

In the Compute Engine, the image from the Docker Hub registry can be used as well; however, there is a limit of one container for VM. If we want to design a microservice application, we probably need to use more than one container per VM. To do so, Google offers another service called *Kubernetes Engine*.

Kubernetes Engine is a managed environment to deploy and manage containerized applications. The engine runs Kubernetes, an open source platform designed to manage and scale containers. Using the Kubernetes Engine allows the use of more than one container. Before proceeding to the creation, we must understand what Kubernetes is.

What Is Kubernetes?

Kubernetes, sometimes referred to as K8s, is an open source platform developed by Google and managed by the Cloud Native Computing Foundation. It is used to manage and scale containerized applications in a cluster, such as Docker.

Note Kubernetes is a very complex application. I could devote an entire book to Kubernetes, which is not within the scope of this chapter. I just want to provide a brief introduction to the Kubernetes platform.

Kubernetes defines some basic building blocks for creating and managing a resource. All the components are designed to be loosely coupled. The first component of the Kubernetes building block is called a *pod* . A pod is a collection of Docker containers, all with the same IP address. Every pod can be composed of one or more containers, hosted in the same host and sharing the resource. Each pod shares a unique IP address with the cluster and can be managed manually, by the API or by the controller.

The other resource defined in Kubernetes is called *labels* . These are used for creating a key-pair value for identifying the resource, for example, a pod. The labels don't provide any uniqueness. In Kubernetes, it is possible to have more than one object with the same name and to create what is called a *label selector* , or *selector* .

With a selector, it is possible to identify a set of objects. The selector is the core primitive for grouping objects in Kubernetes. Actually, we can use two types of selectors, *equality-based* and *set-based* , when we want to identify a resource. A label selector can be made with multiple requirements, separated by commas. In this case, all requirements must be met.

A set-based label selector is used to allow the filtering, using a key according a set of values. The selector supports three types of operators: in, notin, and exist. The selector can be used, for example, to choose only one type of resource, such as environment=production or tier (at the production, stage).

The *controller* is used to manage the state of the cluster. To change the state of the cluster, the controller manages the pod. There are three types of controllers.

- *Replication controller* : This is used to replicate the service across the Kubernetes cluster. Another important use of the Replication controller is to spin up new pods when one fails. It is used to

maintain the minimum number of nodes run and ensure high availability of the cluster.

- *DaemonSet controller* : This controller is responsible for ensuring that all the nodes run a copy of a pod when we add a node in the cluster. Pods are added to the new node by this service.
- *Job controller* : This is used to run a pod until some or all of the run is completed successfully. It checks the status of the pod, and when a certain number is complete, the job terminates itself.

The last piece of the building block is *services* . In Kubernetes, a service is a set of pods working together. For example, we can create a pod with a complete piece of our microservice architecture and define a service with that. The service is defined and identified by a label. This helps to identify the service itself. With Kubernetes, it is possible to search a service with the service discovery and use the IP address or the DNS name for finding the service we need to manage.

Note A loosely coupled system is one designed and implemented so that each service has no, or little, knowledge of the definition of the other services. The services recognize one another only for data exchanged among them and do not know how the service is implemented and if another service is used.

Kubernetes is a container orchestrator for automating deployment, management, and scaling of the containerized application, such as, for example, Docker. This container is defined in pods. This offers a high level of abstraction, grouping the containerized applications. A pod is the essential building block of every Kubernetes application. Pods help us to release our container. Instead of releasing a single container at a time, we can release the pods. A pod can have more than one container in the definition, which helps in releasing more than one container at a time.

With Kubernetes, we can create a *cluster* to manage the system. A cluster is a collection node. A Kubernetes cluster has one master node and zero or more worker nodes. This means that we can have a Kubernetes cluster with just one master node. In Kubernetes, a node is a machine in which it is possible to run and schedule a pod. The machine can be physical or virtual. Because the node is essentially the basic block of the cluster, every node can

be a master node or a worker node. It depends only on the configuration we choose for our cluster.

One of the key components of Kubernetes is *etcd*. This is a lightweight distributed and persistent key-store developed by CoreOS. This component is used to store the state of the cluster at any time. etcd is used to define a high-availability cluster.

The reason for using Kubernetes can be found in the nature of DevOps itself. When a company decides to adopt the DevOps practice, with Kubernetes we can faster release and deploy our application using Docker.

At the same time, Kubernetes can grow with business necessity. This is because of the scalability of the service itself. With Kubernetes we can add or remove a node to the cluster, and Kubernetes takes care to manage the pods with our application.

Using Kubernetes Engine to Deploy an Application

Kubernetes Engine offers the flexibility and power of K8s in the GCP. With Kubernetes Engine it is possible to create a cluster of Docker containers to use to design our microservice architecture. You will now see how to use Kubernetes Engine to create and manage a Docker container and cluster.

To access Kubernetes Engine, we must install another command line: the Kubernetes command line called *kubectl*. This command line gives us the commands needed to manage the Kubernetes cluster. This command line is built into the Google Cloud Shell. We can open the command shell directly from the Google Cloud Console. We can add this console to our Google SDK, as follows:

1. Open the Google Cloud SDK.
2. Run the command `gcloud components install kubectl`. This starts to download and install the Kubernetes command line tool.
3. When the installation is complete, we can verify if all has been correctly installed, using the command `kubectl version`.

This command shows the version installed on the Kubernetes command line (Figure 4-11).

```
C:\Users\user\AppData\Local\Google\Cloud SDK>kubectl version  
Client Version: version.Info{Major:"1", Minor:"8", GitVersion:"v1.8.6", GitCommit:"6260bb08c46c31eea6cb538b34a9ceb3e406689c", GitTreeState:"clean", BuildDate:"2017-12-21T06:34:11Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"windows/amd64"}
```

Figure 4-11. The Kubernetes version command shows the version of the command line

It is possible to use the command line we have in the Google console. All we must do is open the GCP console and click the Console button. This opens the shell, at the bottom of the console (Figure 4-12). We can use the same command and see the same output

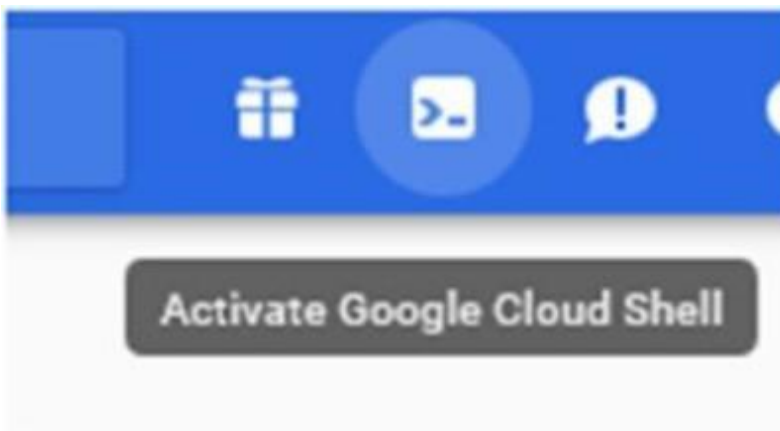


Figure 4-12. The Google

Cloud Shell button

First, to proceed, we must be sure gcloud has the correct project id and zone configured. To set the project id, in our case practicaldevopsgcpcli, use this command:

```
gcloud config set project practicaldevopsgcpcli
```

The output for this command is a simple line: Updated property [core/project]. This indicates that we have correctly updated our project. The next property we need to update is the default compute zone. This way, we are sure to create our cluster exactly where we want. To do this, we can use the following command line:

```
gcloud config set compute/zone us-east1-b
```

The output of this command is a simple line: Updated property [compute/zone]. This indicates that we have correctly updated the compute zone.

Now that we have set up the zone and the code project, we can start to create our Kubernetes cluster. In Kubernetes, a cluster is at least one server. The node is essentially the Compute Engine VM on which we run the Kubernetes process for being part of the cluster. The first step is to create the cluster `gcloud`.

```
gcloud container clusters create practicaldevopsgcpcluster
```

Note The first time we execute this command we can get an error, because we don't have the Kubernetes API configured. We must access it from the following link: <https://console.cloud.google.com/apis/api/container.googleapis.com/overview?project=practicaldevopsgcpcli> .

The command starts to create the Kubernetes cluster, which can take several times to be complete. At the end, we can see the cluster in our console. The result of the operation is a line telling us that the cluster is ready to use. We can navigate to the page. The default cluster has a default of three nodes. The image used in the node is the container-optimized OS. The basic image is created by Google for the container. The size of the machine is an n1-standard, which translates to 1 CPU, with 3.75GB of RAM. This is essentially enough for a basic use of Kubernetes.

https://console.cloud.google.com/kubernetes/workload_/gcloud/us-east1b/practicaldevopsgcpcluster?project=practicaldevopsgcpcli

To see the instance of our cluster, if all the completed operations are good, you will see something like Figure 4-13 .

Name	Status	Type	Pods	Namespace	Cluster
event-exporter-v0.1.7	OK	Deployment	1/1	kube-system	practicaldevopsgcpcli
fluentd-gcp-v2.0.9	OK	Daemon Set	3/3	kube-system	practicaldevopsgcpcli
heapster-v1.4.3	OK	Deployment	1/1	kube-system	practicaldevopsgcpcli
kube-dns	OK	Deployment	2/2	kube-system	practicaldevopsgcpcli
kube-dns-autoscaler	OK	Deployment	1/1	kube-system	practicaldevopsgcpcli
kube-proxy-gke-practicaldevopsgcpcli-default-pool-d02feab7-7vp8	Running	Pod	1/1	kube-system	practicaldevopsgcpcli
kube-proxy-gke-practicaldevopsgcpcli-default-pool-d02feab7-jmq5	Running	Pod	1/1	kube-system	practicaldevopsgcpcli
kube-proxy-gke-practicaldevopsgcpcli-default-pool-d02feab7-nwtm	Running	Pod	1/1	kube-system	practicaldevopsgcpcli
kubernetes-dashboard	OK	Deployment	1/1	kube-system	practicaldevopsgcpcli
i7-default-backend	OK	Deployment	1/1	kube-system	practicaldevopsgcpcli

Figure 4-13. The Kubernetes Engine cluster created

With the cluster ready, what we need to do is get the credentials for accessing the cluster. To get the credentials, we must use the following command:

```
gcloud container clusters get-credentials practicaldevopsgcpcli.
```

The result of this command follows:

Fetching cluster endpoint and auth data.

kubeconfig entry generated for practicaldevopsgcpcli.

We are now ready to deploy our cluster. For our example, we use a Google-ready application. The goal, for now, is to deploy in Kubernetes. The application is called `helloserver`, a simple Go server application.

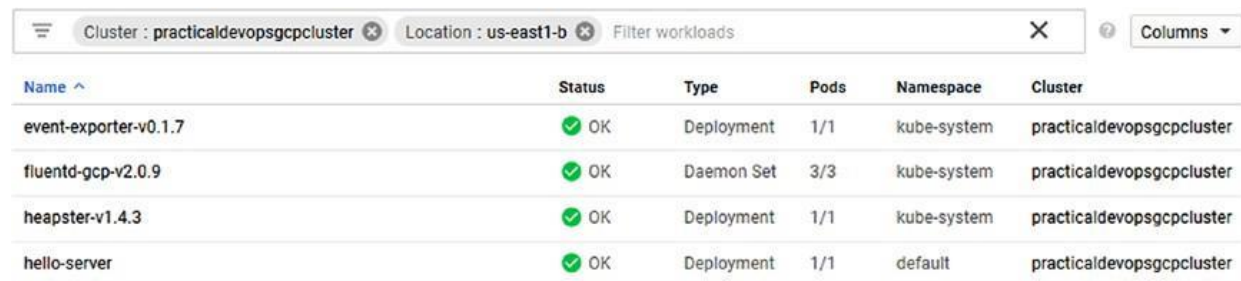
Deploying Our First Kubernetes App

As you saw, Kubernetes creates a cluster and allows us to manage it from one single point. When we create a cluster in Kubernetes, we deploy on the master node, and which the master deploys in every single node.

This behavior is managed by etcd. etcd is the key-value used to remember the status of the cluster when the master sees the different statuses on the node. The daemon present in the node installs the application and aligns every node. Service maintenance is normally done one node at a time. This means that when one node is installing or updating, the other is reachable. For installing the application in the cluster, we can use the `kubectl` command line to launch the following command:

```
kubectl run hello-server --image gcr.io/google-samples/hello-app:1.0 --port 8080
```

This command is very simple and takes the name of the server, with the image parameters and the port to send to the Docker for the parameter. The result of the operation is a simple line: deployment "hello-server" created. We can see the new server in our cluster (Figure 4-14).



Name ^	Status	Type	Pods	Namespace	Cluster
event-exporter-v0.1.7	OK	Deployment	1/1	kube-system	practicaldevopsgcpcluster
fluentd-gcp-v2.0.9	OK	Daemon Set	3/3	kube-system	practicaldevopsgcpcluster
heapster-v1.4.3	OK	Deployment	1/1	kube-system	practicaldevopsgcpcluster
hello-server	OK	Deployment	1/1	default	practicaldevopsgcpcluster

Figure 4-14. The Kubernetes cluster with the hello-server application

It is useful to spend some time analyzing the parameter on the command line. The parameter, `--image`, is used to indicate the image we want to deploy in our cluster. The image can be a public get from the Docker Hub or one from the Google Repository. What's important is the version of the image. In Docker, we can identify the version of an image by adding the value after the `:`, for example, `hello-app:1.0`, indicates version 1.0 for the image `hello-app`. When we fix a version for the image, we must be sure always to deploy the same image version, if we want to put our CI/CD system in place. For example, we can name the image *stable*, because we want to use the last build of the image.

Using a specific name for tagging our Docker image helps to always deploy in production the last image with that name. When we release the image to the registry, Kubernetes checks that the image is the latest and downloads for

installation the image with the same tag. This means that if we put a new stable in the registry, this is automatically deployed in our system.

We have now deployed our application, but for it to be useful, it must be reachable from other users. To expose our application on the Internet, we must configure a load balancer.

A load balancer is used to balance the resources across the cluster. This is important when we have to manage the traffic coming from the Internet. Because Kubernetes creates a cluster of the resource, it is normal to use a cluster to manage the traffic. In Kubernetes, we can expose the cluster, using the command `kubectl expose` from the `kubectl` command line.

```
kubectl expose deployment hello-server --type "LoadBalancer"
```

We have defined the load balancer when we add the parameter `--type "LoadBalancer"`. This exposes our service to the Internet and configures a load balancer to manage the request. The `LoadBalancer` command is important, if we want to expose the service to the outside world. `LoadBalancer` is created in GCP, and after the IP is assigned to the external IP of the cluster. In this way, it is possible to connect to the cluster and show the service run.

The application is exposed on the Internet using an external IP. To see the IP, we must see the information about our cluster. We can get the information by using the command `kubectl get service`. This shows the information about the service we indicate in Kubernetes. To get the information about our application, we use the following command:

```
kubectl get service hello-server
```

The result is a table showing the information about the service.

```
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE hello-server
LoadBalancer 10.35.247.216 35.196.27.26 8080:32104/TCP 20m
```

Note The external IP is used to identify our application. This will be different, depending on your Google Cloud configuration.

We can see our application working (Figure 4-15). Open the browser and follow this link:

<http://35.196.27.26:8080>

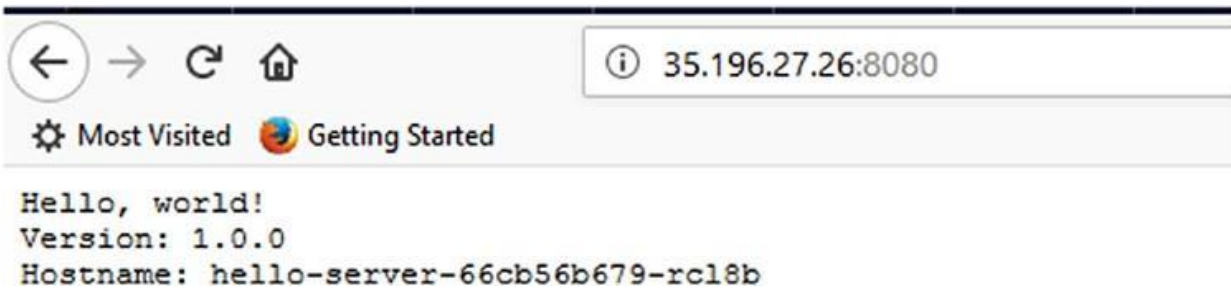


Figure 4-15. The live hello-server application

Congratulations! We have now deployed our first Kubernetes application.

Configuring the Kubernetes Dashboard

Until now, we have seen how to use Kubernetes via command line, but Kubernetes has an amazing interface. To use it in GCP, we must configure it for access on that platform.

Caution With the new version of GCP, Google strongly recommends that you disable the classic Kubernetes UI and use the Google Console dashboard instead.

By installing and configuring Kubernetes, we have created a cluster. This has different basic components that can be accessed via HTTP/HTTPS. We can see the information about the cluster by using this command:

```
kubectl cluster-info
```

This shows us the information about the cluster (Figure 4-16).

```
pierluigi_riti@practicaldevopsgcpcli:~$ kubectl cluster-info
Kubernetes master is running at https://35.229.75.9
GLBCDefaultBackend is running at https://35.229.75.9/api/v1/namespaces/kube-system/services/default-http-backend/http/proxy
Heapster is running at https://35.229.75.9/api/v1/namespaces/kube-system/services/heapster/proxy
KubeDNS is running at https://35.229.75.9/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Kubernetes-dashboard is running at https://35.229.75.9/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy
Metrics-server is running at https://35.229.75.9/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 4-16. The Kubernetes cluster information

To access the dashboard, we need to follow this link:

<https://<kubernetes master ip>/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy>

If we attempt to access the UI, we get the error shown in Figure 4-17 . This is because we haven't configured the service account to access the Kubernetes cluster.


```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {
  },
  "status": "Failure",
  "message": "services \"https:kubernetes-dashboard:\" is forbidden: User \"system:anonymous\" cannot get services/proxy in the namespace \"kubernetes-dashboard\": Unknown user",
  "reason": "Forbidden",
  "details": {
    "name": "https:kubernetes-dashboard:",
    "kind": "services"
  },
  "code": 403
}
```

Figure 4-17. The error we have when trying to access the Kubernetes UI

To gain access to the Kubernetes UI, we must configure the identity and access management (IAM) for the service account.

Note You'll learn more about IAM in Chapter 9. For now, you only have to know how to configure the user access we need for the Kubernetes UI.

The first step is to get the key for the user. For this, we use connect to the command line interface. We can do this simply by using the Console dashboard and moving to the IAM & Admin section and, from there, to the Service Account screen (see Figure 4-18).

Email	Name ↑	Key ID	Delete key	Key creation date	Actions
practicaldevopsgcpcli@appspot.gserviceaccount.com	App Engine default service account	No keys			⋮
377223342623-compute@developer.gserviceaccount.com	Compute Engine default service account	No keys			⋮

Figure 4-18. The Service Account screen on Google Console

Now we can see that we don't have any key for the service account for our project. This generated the error we saw when we tried to access the Kubernetes UI. First, we must create the key for the account and, after, download the key in the JSON format and use that to configure access to our Kubernetes UI.

To create the key, we simply click the three dots under Actions and then select the Create key menu option (see Figure 4-19).

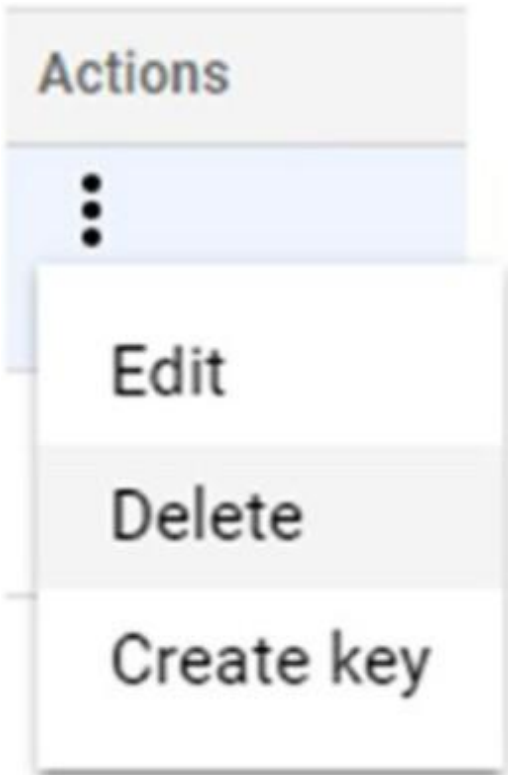


Figure 4-19. *The menu for creating a new key*

Create key opens a new modal window from which to select the type of key we want to create. Select JSON and then click Create. This creates a new key associated with our service account. At the same time, the key is downloaded to our PC. The key can be used to configure access on the Kubernetes UI (see [Figure 4-20](#)).

Service accounts

+ CREATE SERVICE ACCOUNT

DELETE

SHOW INFO PANEL

Service accounts for project 'practicaldevopsgcpcli'

A service account represents a Google Cloud service identity, such as code running on Compute Engine VMs, App Engine apps, or systems running outside Google. [Learn more](#)

Filter table

?

⋮






<input type="checkbox"/>	Email	Name ↑	Key ID	Delete key	Key creation date	Act
<input type="checkbox"/>	 practicaldevopsgcpcli@appspot.gserviceaccount.com	App Engine default service account	0a31be1c0098a65f6237553d10edd6e4e01dd891		7 Aug 2018	
<input type="checkbox"/>	 377223342623-compute@developer.gserviceaccount.com	Compute Engine default service account	No keys			

Figure 4-20. The key associated with the service account

With the key created and associated with our service account, we can configure the service account access for the Kubernetes UI. The command we need to configure the gcloudauthorization follows:

gcloud auth activate-service-account practicaldevopsgcpcli@appspot.gserviceaccount.com --key-file=<path where we download the JSON key>
We can execute the command in Cloud SDK. The result of the command tells us that the service account is now activated (see Figure 4-21).

```
C:\Program Files (x86)\Google\Cloud SDK>gcloud auth activate-service-account practicaldevopsgcpcli@appspot.gserviceaccount.com --key-file=C:\practicaldevopsgcpcli-0a31be1c0098.json
Activated service account credentials for: [practicaldevopsgcpcli@appspot.gserviceaccount.com]
```

Figure 4-21. The service account activation via Cloud SDK

Now we have configured access with Kubernetes, using our service account. The next step is to update the cluster we previously created in GCP and update it with our new service account credentials, using the following command:

```
gcloud container clusters get-credentials practicaldevopsgcpcluster --zone us-east1-b --project practicaldevopsgcpcli
```

This command adds a new entry in the kubeconfig generated for the cluster and associates the service account with the cluster. To run the dashboard, first we must see all the Kubernetes services in the cluster. We can do this with the following command:

```
kubectl get secrets -n kube-system
```

Note In case we encounter an error while running the `kubectl` command, we can install it as part of the Google Cloud SDK. The command for installation is `gcloud components install kubectl`. This installs and downloads the Kubernetes command line on the SDK.

This shows any token we have in our cluster. We can use this to view the information about the token that we require to access our dashboard (see [Figure 4-22](#)).

```
C:\Program Files (x86)\Google\Cloud SDK>kubectl get secrets -n kube-system
```

NAME	TYPE	DATA	AGE
attachdetach-controller-token-5t8vf	kubernetes.io/service-account-token	3	1d
certificate-controller-token-4vx8z	kubernetes.io/service-account-token	3	1d
cloud-provider-token-5mwbx	kubernetes.io/service-account-token	3	1d
clusterrole-aggregation-controller-token-bpzc	kubernetes.io/service-account-token	3	1d
cronjob-controller-token-q6hnw	kubernetes.io/service-account-token	3	1d
daemon-set-controller-token-dn96w	kubernetes.io/service-account-token	3	1d
default-token-7fqdw	kubernetes.io/service-account-token	3	1d
deployment-controller-token-brn6x	kubernetes.io/service-account-token	3	1d
disruption-controller-token-wxws2	kubernetes.io/service-account-token	3	1d
endpoint-controller-token-76pn8	kubernetes.io/service-account-token	3	1d
event-exporter-sa-token-474ml	kubernetes.io/service-account-token	3	1d
fluentd-gcp-token-5l8w2	kubernetes.io/service-account-token	3	1d
generic-garbage-collector-token-n9vfd	kubernetes.io/service-account-token	3	1d
heapster-token-wrx4n	kubernetes.io/service-account-token	3	1d
horizontal-pod-autoscaler-token-4qsn2	kubernetes.io/service-account-token	3	1d
job-controller-token-ph9hm	kubernetes.io/service-account-token	3	1d
kube-dns-autoscaler-token-lcw6b	kubernetes.io/service-account-token	3	1d
kube-dns-token-9mkhk	kubernetes.io/service-account-token	3	1d
kubernetes-dashboard-certs	Opaque	0	1d
kubernetes-dashboard-key-holder	Opaque	2	1d
kubernetes-dashboard-token-f5s2g	kubernetes.io/service-account-token	3	1d
metadata-proxy-token-k5r8j	kubernetes.io/service-account-token	3	1d
metrics-server-token-kgk6z	kubernetes.io/service-account-token	3	1d
namespace-controller-token-8tmhd	kubernetes.io/service-account-token	3	1d
node-controller-token-bbdp6	kubernetes.io/service-account-token	3	1d
persistent-volume-binder-token-rm86x	kubernetes.io/service-account-token	3	1d
pod-garbage-collector-token-kvvvw	kubernetes.io/service-account-token	3	1d
pv-protection-controller-token-m8pv9	kubernetes.io/service-account-token	3	1d
pvc-protection-controller-token-466xs	kubernetes.io/service-account-token	3	1d
replicaset-controller-token-thsvx	kubernetes.io/service-account-token	3	1d
replication-controller-token-wtfsb	kubernetes.io/service-account-token	3	1d
resourcequota-controller-token-wzqf9	kubernetes.io/service-account-token	3	1d
route-controller-token-kccvn	kubernetes.io/service-account-token	3	1d
service-account-controller-token-jtpcq	kubernetes.io/service-account-token	3	1d
service-controller-token-m64qh	kubernetes.io/service-account-token	3	1d
statefulset-controller-token-4mplk	kubernetes.io/service-account-token	3	1d
tll-controller-token-6bbq7	kubernetes.io/service-account-token	3	1d

Figure 4-22. The Kubernetes tokens present in our Kubernetes cluster

To connect to the UI, we must have a token. This token is used when we need to enter the Kubernetes UI. We can get the token, using the following command: `kubectl -n kube-system describe secret replicaset-controller-token-thsvx`

Note that the value of the replicaset-controller-token can be different on your configuration, in particular, the last part. When we run this command, we can see an output such as that in Figure 4-23 .

[illegible]

Figure 4-23. The Kubernetes token for access to the dashboard

With the token, we can finally run the command `kubectrl proxy`. This starts the server on our computer and shows the address and the port for access to it. This is usually `127.0.0.1:8001`. We can run the server and insert in our browser the address `localhost:8001/ui/`. This executes the Kubernetes dashboard and asks us how to authenticate on the server (see Figure 4-24).

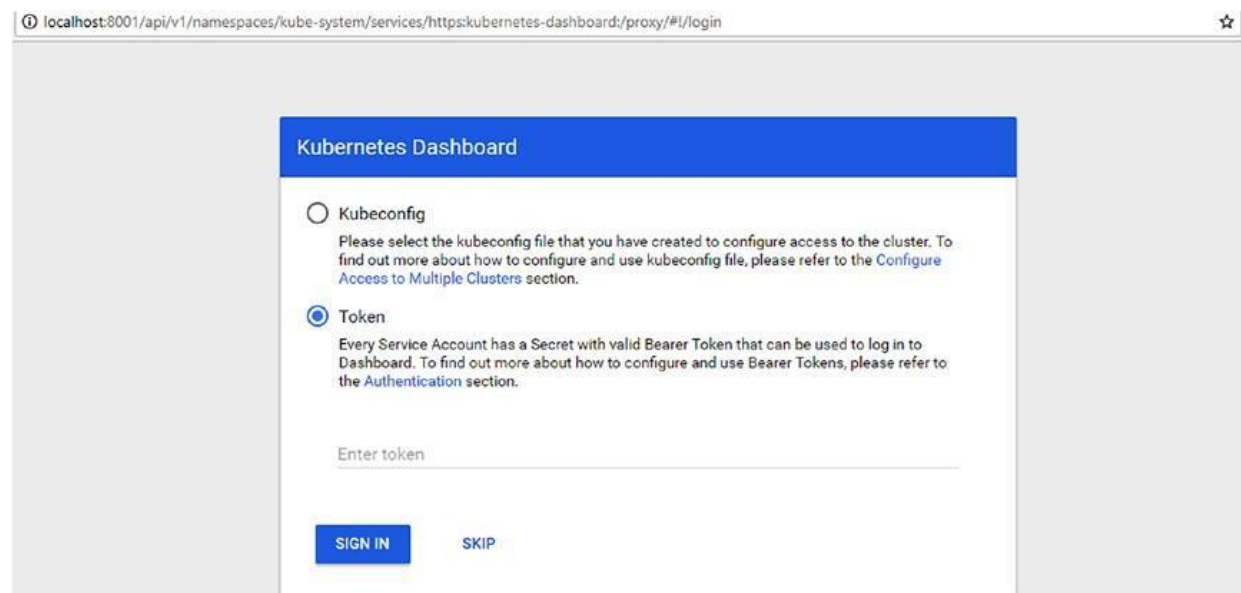


Figure 4-24. *The Kubernetes dashboard login*

We can insert the token we got before and use it to log in to the dashboard. The dashboard reveals a warning. This is because every service has a different token. The dashboard is shown in Figure 4-25 .

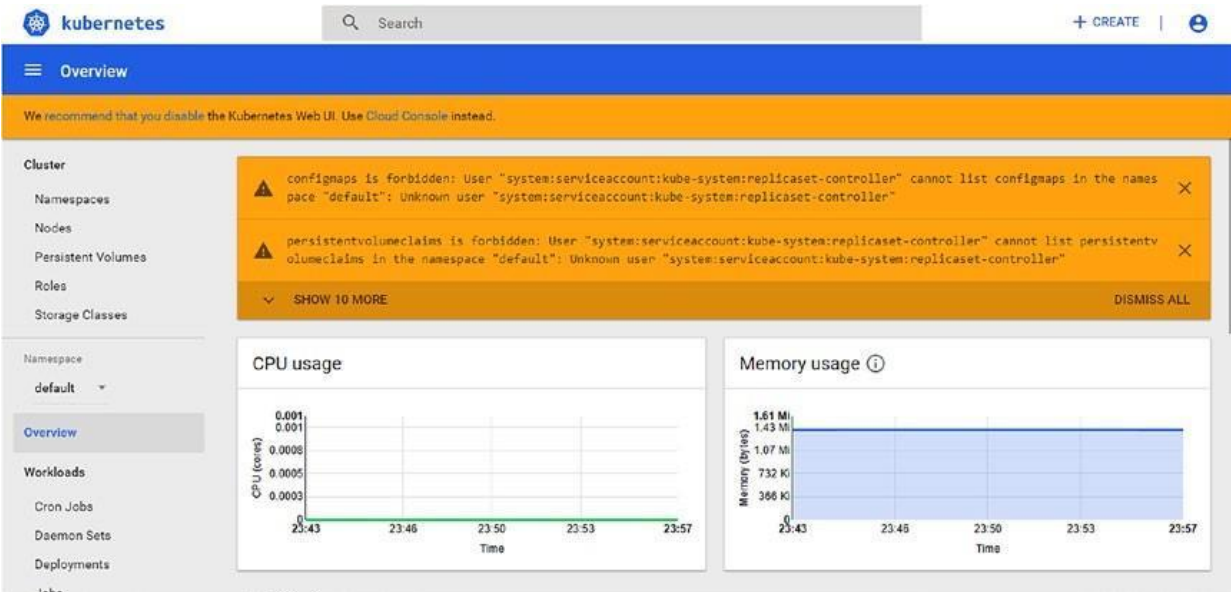


Figure 4-25. *The Kubernetes UI*

We can see that Google recommends that we disable the classic Kubernetes UI and use the Cloud Console UI. This because the latter is better integrated into the system.

Exploring the Kubernetes Instance

When we create our Kubernetes cluster, we essentially create a different compute engine. We can check the instance created, in the Kubernetes Compute Engine dashboard

Figure 4-26. *The Kubernetes Compute Engine created to manage the cluster*

We can identify the instances, because they have all been created with the prefix gke. We can check the details of an instance just by clicking the instance.

With the console, it is possible to explore the Kubernetes configuration of the cluster. If we open the console and check the Kubernetes Compute Engine, we can see the information about the cluster (Figure 4-27).

Kubernetes clusters							CREATE CLUSTER	REFRESH	DELETE	SI
Filter by label or name										
Kubernetes clusters										
<input type="checkbox"/>	Name ^	Location	Cluster size	Total cores	Total memory	Notifications	Labels			
<input type="checkbox"/>	practicaldevopsgcpcluster	us-east1-b	3	3 vCPUs	11.25 GB			Connect		

Figure 4-27. The Kubernetes cluster information

We can see that the cluster has the minimum value of three nodes. It is possible to change the value of the cluster, by clicking the pencil icon on the right (Figure 4-28).

Figure 4-28. The node pools section for the Kubernetes cluster

It is possible to manage the node of the cluster in the section Node Pools. In this section, we can configure a different parameter of the cluster, for example, change the number of the node in the pool. This is configured in the Size value. We can change the value by adding more nodes in the cluster. We can, for example, change the value to 5 and add another two nodes to the cluster. In this section, it is possible to enable the autoscaling for the cluster. This adds and deletes nodes automatically to the cluster

VM instances							CREATE INSTANCE	IMPORT VM	REFRESH					SHOW
Filter VM instances														
<input type="checkbox"/>	Name ^	Zone	Recommendation	Internal IP	External IP	Connect								
<input type="checkbox"/>	gke-practicaldevopsgcpcl-default-pool-d02feab7-4hc8	us-east1-b		10.142.0.8	35.196.61.57	SSH	⌵	⋮						
<input type="checkbox"/>	gke-practicaldevopsgcpcl-default-pool-d02feab7-7vp8	us-east1-b		10.142.0.4	35.231.194.107	SSH	⌵	⋮						
<input type="checkbox"/>	gke-practicaldevopsgcpcl-default-pool-d02feab7-bb86	us-east1-b		10.142.0.7	35.196.37.153	SSH	⌵	⋮						
<input type="checkbox"/>	gke-practicaldevopsgcpcl-default-pool-d02feab7-jmq5	us-east1-b		10.142.0.5	35.185.95.169	SSH	⌵	⋮						
<input type="checkbox"/>	gke-practicaldevopsgcpcl-default-pool-d02feab7-nwtm	us-east1-b		10.142.0.6	35.231.211.215	SSH	⌵	⋮						

Figure 4-29. The new Compute Engine instance

Deleting a Kubernetes Cluster

To complete our tour of Kubernetes, we must learn how to destroy our Kubernetes cluster. This sometimes is necessary to maintain company costs

or to dismiss old infrastructure.

We can delete the Kubernetes cluster, using this command: `kubectl delete service hello-server`. This deletes the Kubernetes load balancer. After the load balancer is deleted, we now have to delete the cluster. The command to delete the cluster is `gcloud container clusters delete practicaldevopsgcpcluster`. This command destroys the cluster we have created in Google Cloud and asks to confirm the deletion of the cluster. Select Y to delete the cluster.

Conclusion

Using containers is important for accelerating the time-to-market process and offers a lot of benefits. The container application can be used to replicate the development production environment. With the container application, we can put in place the CI and CD practice. This chapter gave a very short introduction to Kubernetes and showed how to configure the classic Kubernetes UI in GCP. The intention was not to explain Kubernetes in depth.

In GCP, we can create an instance programmatically, using the Compute Engine. The

Compute Engine is a VM used for container applications and containers. The best way to manage a container is to use the Kubernetes Compute Engine. With this, it is possible to create a cluster of engines with a specific operating system designed for maintaining containers. In the next chapters, you will learn how to use Kubernetes and integrate it with the CI and CD practice, to implement the principles of DevOps.

CHAPTER 5

Continuous Delivery with GCP and Jenkins

Continuous delivery, as an extension of continuous integration, is essentially an evolution of the latter and one of the pillars of DevOps. There is a variety of software and tools for putting a CD system in place, such as Travis CI, GoCD, and Bitbucket, but probably the tool most often used is Jenkins. In this chapter, you will see how to use Jenkins to design and define our CD pipeline in Google Cloud Platform (GCP).

An Introduction to Jenkins

Jenkins is probably the most famous software related to continuous integration and delivery (CI/CD). Jenkins is an open source software written in Java to help automate all the nonhuman processes that occur during software development.

Jenkins works with almost any type of repository, such as CVS, Git, Subversion, Mercurial, and others. It is possible to automate all the processes related to CI/CD very easily.

The most important concept in Jenkins is the plug-in. It is possible to find a plug-in for almost any necessity. Plug-ins are split into different uses, for example

- .NET development
- Android development
- Authentication and user management
- Command-line interface
- Build notifiers
- Deployment

This is just a short list of all the different areas in which plug-ins are useful. With plug-ins, we can use Jenkins to cover every aspect of continuous integration and delivery.

The popularity of Jenkins is due to the following:

- Ease of installation and provision of a consistent interface across different operating systems

- The possibility of modularity, to write a plug-in for extending Jenkins
- A simple interface to use and configure
- The possibility, with master/slave architecture, to use the master and slave in different operating systems, to achieve different results from the same build
- Easy configuration
- Easy reporting of the historical status of the builds

These are the principles that make Jenkins so ubiquitous. Jenkins is often used for CI, because the focus of CI is the rapidity of building after the commit and the fast and reliable automation of tasks. With Jenkins, it is possible to use a trigger to start the build after a commit, and we can automate any single task associated with that. This means that every time we release, we execute all the tasks necessary to build the software.

Continuous Integration and Delivery with Jenkins

Jenkins offers very good support for creating CI and CD pipelines. I use the term *pipelines* because we must execute and repeat a series of steps to achieve the same results (see Figure 5-1).

Continuous Integration Continuous Delivery

Code

Unit Test

Code System Integration Testing Stage User Production Release Acceptance Release

Figure 5-1. *The pipeline for continuous integration and delivery*

The pipelines for CI/CD require that some steps be put in place. I will now discuss these steps and explain how Jenkins can be used for automating them, to create a complete pipeline for CI/CD.

Code

The first step in creating the pipeline is the code. When the code is ready, and the developer has completed it and tested it locally, it is usually released to the central repository. Because we are implementing continuous

integration, we normally release in another branch. Usually, this is connected to a branching policy. For example, we can create a branch with the number of tasks we want to implement. Using a branching policy helps to define the different tasks actually in development and to identify what part of the code is actually released.

Unit Test

The next step in developing the pipeline is the unit test. This step consists of writing the test for the code and executing it directly when we start the process for CI and CD. Jenkins can automate this process in an easy way. With Jenkins, it is possible to connect a repository and trigger the unit test when the code is committed directly to the repo. Jenkins downloads the code from the repo. After the download, we can build the software and execute the test suite. After the building, if the test is passed, we can move on to the next steps. This entire process can be easily managed by Jenkins, when we configure the project and connect our code to the repo.

Code Integration

When the code has been tested, it is time to integrate it into the main branch. This can be automatic or, in most cases, after a code review policy. Use a code review policy to enforce the quality of the code and help share the knowledge of the system. With this policy, we require a minimum of two people to approve the code first, before it is merged into the main branch. When the code is integrated, we can start another battery of unit tests on the main branch. For a CD policy, the code must be integrated automatically, but this means that there is no review executed by a human. In this case, we can use a *static code analysis* to identify potential problems with the code. With Jenkins, it is possible to use a tool to execute the static code analysis, sometimes called code analysis, if the build fails.

System Testing

When the code is integrated into the main branch, it is possible to start some system testing. This kind of testing is different from the unit test, because we use real, rather than mock, data. This kind of testing is sometimes called

integration testing. With this type of testing, we integrate and test all components of the system. The scope of this type of testing identifies any new bug introduced with the new feature, and the data is derived from a real, rather than a mock, component of the system, because we want to test the entire system.

Stage Release

When the system testing is green-lighted, Jenkins can take care of the release of the software. This can occur in different ways, depending on how we release the software. If, for example, we want to release Docker, Jenkins can push the code directly into the registry we use for the stage server. With Jenkins, we can also execute the script in Bash or Windows, to deploy the software on the server. Release in a stage server is crucial to ensure the quality of the software. This server is used by the QA engineering team and must be similar to that used in the production environment. In this way, it is possible to emulate the real environment and grab more errors before release to production.

User Acceptance

User acceptance is an important phase in constructing continuous release pipelines. It consists of a battery of tests to verify the functionality, from the user's point of view. This phase can be automatic, based on the test written by the QA engineering team, but it can include some manual execution. In relation to continuous delivery, this test is first executed using Jenkins, and after the code is released in some canary server, the QA team tests the software manually. This test is used to develop an agreement with the end user about the most desirable software to release.

Production Release

The last and most critical phase of the pipeline process is the production release. In keeping with continuous delivery policy, this phase can occur more than once a day. In addition, we can decide not to release directly to production but opt for a sample release on a canary server and use these servers to test the release with a limited number of users. This helps to

identify any issues and to create alerts in the system. In CD, we automatize all process and release any change in the code directly to production. This can be a simple image change or a label or a fix for a bug. With a pure continuous delivery policy, we essentially release a small part of the software more times per day. Jenkins can help to automate all aspects of this phase and create a complete pipeline for CD.

Designing a Good Branching Strategy

Designing a good branching strategy is essential for having good continuous integration and delivery. The most popular strategy for branching is to create a branch for every feature (Figure 5-2). When we release the software on the branch, we execute the unit test, and if it is green-lighted, we can merge the code. For this, the Git Workflow software is very useful. This software is built and manage by Atlassian and is the de facto industry standard.

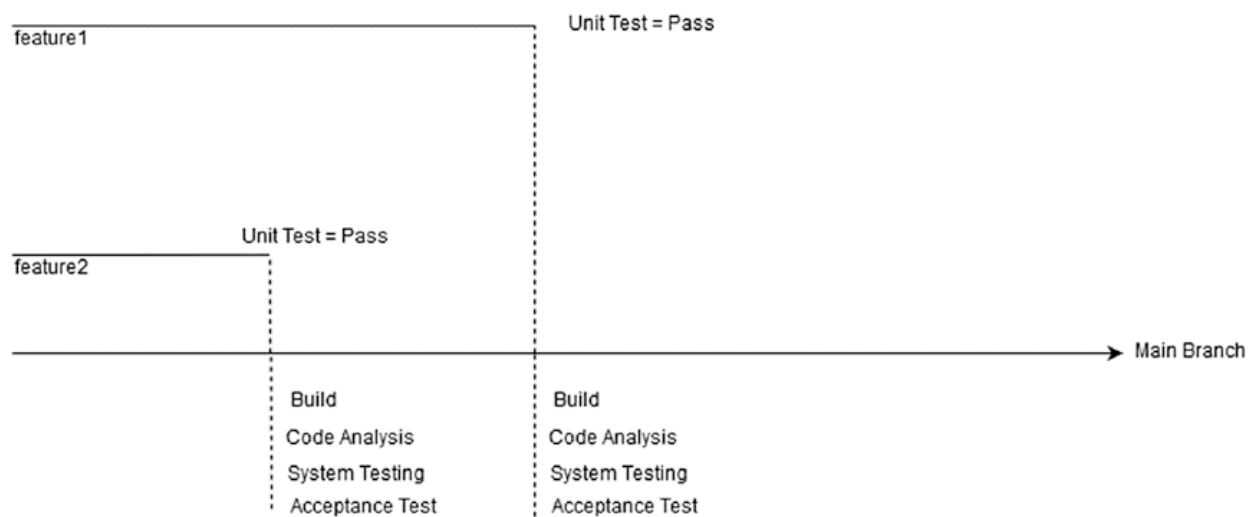


Figure 5-2. Feature branching strategies

This strategy is the simplest and is completely automatic. Every time we merge the code into the main branch, Jenkins builds the code when the build is ready. Jenkins executes some other steps to check the quality of the software we release.

The first step is the code analysis or static code analysis. This is performed using a plug-in. If the code analysis is positive, we can move to the next step, which is the *system testing* or *integration test* . This is normally

designed to test the system with real, rather than mock, data. The aim is to use the real component for the test.

The last step we execute is the *acceptance test* . Usually, this test is written by an engineer, but in some cases, this test can be written by the QA engineering team, and the goal is to test the functionality from a user's point of view. This test is intended to check for quality, to ensure that all functionality is fully implemented. Another use of this test is to make sure that the older functionality is working with the new software implementation.

If all phases have been properly executed, we have just created a new build that can be installed in production or, depending on the policy, in the canary server. These steps are performed every time we commit a new feature or fix a bug. In the repository, the process can start, for example, when we only update a label or change an image.

With Jenkins, it is possible to create a pipeline using a multibranch pipeline job. This kind of job tells Jenkins to create a new pipeline when a new branch is created on the repo. This feature is managed by the multibranch plugin.

When we want to implement CI and CD in the company, we probably need to create a multibranch system for our Jenkins. It is possible to create a multibranch project using a *Jenkinsfile* .

The multibranch solution is the accepted choice when we decide to use branching strategies. This is because we can create a CI/CD pipeline for every new branch we create in the repo.

Using Jenkins in GCP

Until now, you have had an introduction to Jenkins and some features we can use to create CI/CD pipelines. Now you'll start to learn how to use Jenkins in GCP. The reason for creating our CI/CD system in the cloud is the facility with which we can spin up a new server and scale up, according to our business needs.

To use Jenkins on GCP, we need to use Google Kubernetes Engine and Jenkins in the cluster we create via Kubernetes. Using Kubernetes to create our CD pipelines gives the system some important benefits.

- In a microservice architecture or a multi-operating system environment, one virtual host created in Kubernetes can run the job against the different operating systems. This is made possible with server-slave architecture.
 - With Kubernetes, we have ephemeral executors, meaning we can execute the build in a clean environment every time we execute new jobs. This can eliminate errors in an unclean environment.
 - Build executors are run in seconds.
 - The Kubernetes clusters are used only when our feature is active, which saves resources and leaves the cluster free for use.
- First of all, we must set up our Kubernetes Cluster Engine, which is the backbone for creating our Jenkins environment.

Configuring Jenkins on Kubernetes

For creating our Jenkins CI/CD pipelines, we must follow these steps:

1. Create a new Kubernetes cluster.
2. Install Jenkins on the cluster.
3. Install the Jenkins plug-in for work with Kubernetes.
4. Configure Kubernetes to start the Jenkins process.

The first step is to connect on the GCP, open Google Shell, and then create our Kubernetes cluster. First, we create a new network component for our Kubernetes cluster. The command for creating the network is

```
gcloud compute networks create jenkinscid
```

This command creates a new virtual private cloud (VPC) network. We use this network to create our Kubernetes Engine. The result of the operation is shown in Listing 5-1 .

Listing 5-1. Result for the Kubernetes VPC Creation

```
pierluigi_riti@practicaldevopsgcpcli:~$ gcloud compute networks create jenkinscid
```

Created [<https://www.googleapis.com/compute/v1/projects/practicaldevopsgcpcli/global/networks/jenkinscid>].

NAME SUBNET_MODE BGP_ROUTING_MODE IPV4_RANGE
GATEWAY_IPV4 jenkinscid AUTO REGIONAL

Instances on this network will not be reachable until firewall rules are created. As an example, you can allow all internal traffic between instances as well as SSH, RDP, and ICMP by running:

```
$ gcloud compute firewall-rules create <FIREWALL_NAME> --network  
jenkinscid --allow tcp,udp,icmp --source-ranges <IP_RANGE> $ gcloud  
compute firewall-rules create <FIREWALL_NAME> --network jenkinscid  
--allow tcp:22,tcp:3389,icmp
```

In our case, we don't have to expose our cluster outside, and we can easily skip the configuration of the firewall. Now that we have created a new VPC, we can start to construct our Kubernetes cluster.

Note Virtual private cloud is how Google is able to show or hide our instance on the Internet. By default, with the Google instance, we can see a default VPC enabled and exposed. We can manage the connection by a firewall for allowing or blocking traffic, with a specific port or protocol. In every project, it is possible to have multiple VPCs, and all VPCs are IPv4, as IPv6 is not supported. Something important to keep in mind is that every VPC has a "hidden" firewall rule: every Transmission Control Protocol (TCP) connection is dropped after ten minutes of inactivity.

We create the VPC because we don't want to use the default VPC. This is because Jenkins is used for internal applications, and we don't want to configure different rules. If we stay in the same network, the rules we create are reflected throughout the network.

Now that we have created the new network, we can build the new Kubernetes cluster, using the Kubernetes Engine.

```
gcloud container clusters create jenkins-cd \  
--network jenkinscid \  
--zone us-east1-b \  
--scopes "https://www.googleapis.com/auth/projecthosting,storage-rw"
```


We add an attribute `--scopes`. It allows the cluster to access *Google Cloud Storage* and the *Google Cloud Container Registry*. The parameter `--scopes` is used to specify the type of node instance. We can define different scopes for our instance. We can also define multiple scopes, but we must separate these with a comma. Scopes are defined using the full URI. In GCP, we can define this URI. (See Table 5-1.)

Table 5-1. *URI for the most common Google Cloud Resource*

Alias URI

bigquery <https://www.googleapis.com/auth/bigquery> cloud-platform
<https://www.googleapis.com/auth/cloud-platform> cloud-source-repos
https://www.googleapis.com/auth/source.full_control cloud-source-repos-ro
https://www.googleapis.com/auth/source.read_only compute-ro compute-rw
 datastore default

gke-default

logging-write monitoring
 monitoring-write pubsub

<https://www.googleapis.com/auth/compute.readonly>
<https://www.googleapis.com/auth/compute>
<https://www.googleapis.com/auth/datastore>
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/pubsub>

<https://www.googleapis.com/auth/service.management.readonly>

<https://www.googleapis.com/auth/servicecontrol>
<https://www.googleapis.com/auth/trace.append>
https://www.googleapis.com/auth/devstorage.read_only
<https://www.googleapis.com/auth/logging.write>
<https://www.googleapis.com/auth/monitoring>

<https://www.googleapis.com/auth/service.management.readonly>

<https://www.googleapis.com/auth/servicecontrol>
<https://www.googleapis.com/auth/trace.append>
<https://www.googleapis.com/auth/logging.write>
<https://www.googleapis.com/auth/monitoring>
<https://www.googleapis.com/auth/monitoring.write>
<https://www.googleapis.com/auth/pubsub>

(continued)

Table 5-1. *(continued)*

Alias

service-control service-management

sql

sql-admin

storage-full

storage-ro storage-rw

taskqueue

trace

userinfo-email

URI

<https://www.googleapis.com/auth/servicecontrol>
<https://www.googleapis.com/auth/service.management.readonly>
<https://www.googleapis.com/auth/sqlservice>
<https://www.googleapis.com/auth/sqlservice.admin>
https://www.googleapis.com/auth/devstorage.full_control
https://www.googleapis.com/auth/devstorage.read_only
https://www.googleapis.com/auth/devstorage.read_write
<https://www.googleapis.com/auth/taskqueue>
<https://www.googleapis.com/auth/trace.append>
<https://www.googleapis.com/auth/userinfo.email>

We can use the URI to define the scopes of our instance. The attribute `--network` identifies what VPC we want to use for our cluster. (The command takes some time to complete.) We must define as well the zone where we

want to create our cluster. This is defined by the attribute--zone. At the end, we see this result:

WARNING: Starting in 1.12, new clusters will have basic authentication disabled by default. Basic authentication can be enabled (or disabled) manually using the `--[no-]enable-basic-auth` flag.

WARNING: Starting in 1.12, new clusters will not have a client certificate issued. You can manually enable (or disable) the issuance of the client certificate using the `--[no-]issue-client-certificate` flag. 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.

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.

WARNING: The behavior of --scopes will change in a future gcloud release: service-control and service-management scopes will no longer be added to what is specified in --scopes.

To use these scopes, add them explicitly to --scopes. To use the new behavior, set container/new_scopes_behavior property (gcloud config set container/new_scopes_behavior true).

WARNING: Starting in Kubernetes v1.10, new clusters will no longer get compute-rw and storage-ro scopes added to what is specified in --scopes (though the latter will remain included in the default --scopes). To use these scopes, add them explicitly to --scopes. To use the new behavior, set container/new_scopes_behavior property (gcloud config set container/new_scopes_behavior true).

Creating cluster jenkins-cd...done.

Created [<https://container.googleapis.com/v1/projects/practicaldevopsgcpcli/zones/us-east1-b/clusters/jenkins-cd>]. To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_/gcloud/us-east1-b/jenkins-cd?project=practicaldevopsgcpcli

kubeconfig entry generated for jenkins-cd.

NAME LOCATION MASTER_VERSION MASTER_IP

MACHINE_TYPE NODE_VERSION NUM_NODES STATUS

```
jenkins-cd us-east1-b 1.9.7-gke.3 jenkins-cd us-east1-b 1.9.7-gke.3 standard-  
1 1.9.7-gke.3 3 RUNNING
```

In this configuration, we use Google Cloud Container Registry and Google Cloud Storage, because we must allow Jenkins to access the service.

With Google Cloud Container Registry, we allow Jenkins to access the repositories we can use to store our code, in this case, projecthosting. The other scope is Google Cloud Storage, which allows Jenkins to access Cloud Storage and store the container or the build we have created. The parameter-rw indicates an access on read-write. The next step is to create the Jenkins/Kubernetes architecture for our environment.

We must get the credentials for the cluster we created. These credentials are used to gain access to the Kubernetes cluster that hosts our Jenkins deployment. To do this, we can use the following command:

```
gcloud container clusters get-credentials <cluster name>
```

In our case, the cluster name is jenkins-cd. The result of the command follows:

```
pierluigi_riti@practicaldevopsgcpcli:~$ gcloud container clusters  
getcredentials --zone us-east1-b jenkins-cd  
Fetching cluster endpoint and auth data.  
kubeconfig entry generated for jenkins-cd.
```

The credentials are now configured, and we can now think about and implement our Jenkins architecture.

Designing the Jenkins Architecture

What we want to design is a master/slave Jenkins architecture. This type of architecture is used to manage distributed Jenkins architecture. In simple terms, the master launches the job with one slave and monitors the slave to check the status of the job.

This type of architecture is used in particular when we want to scale up our Jenkins configuration. The job can be executed for any available slave, or we can configure the master for executing the job in a specific slave node. Fixing the job in a slave node can be used, for example, when we must

execute a job in a particular SO, or when we must use a specific configuration.

Figure 5-3 shows the architecture we want to use to implement our Jenkins CD pipelines in Kubernetes. This configuration is built using two nodes, the minimal requirement for a Kubernetes cluster. The cluster allows us to expand our test necessity, adding another node, in case the number of tests increases.

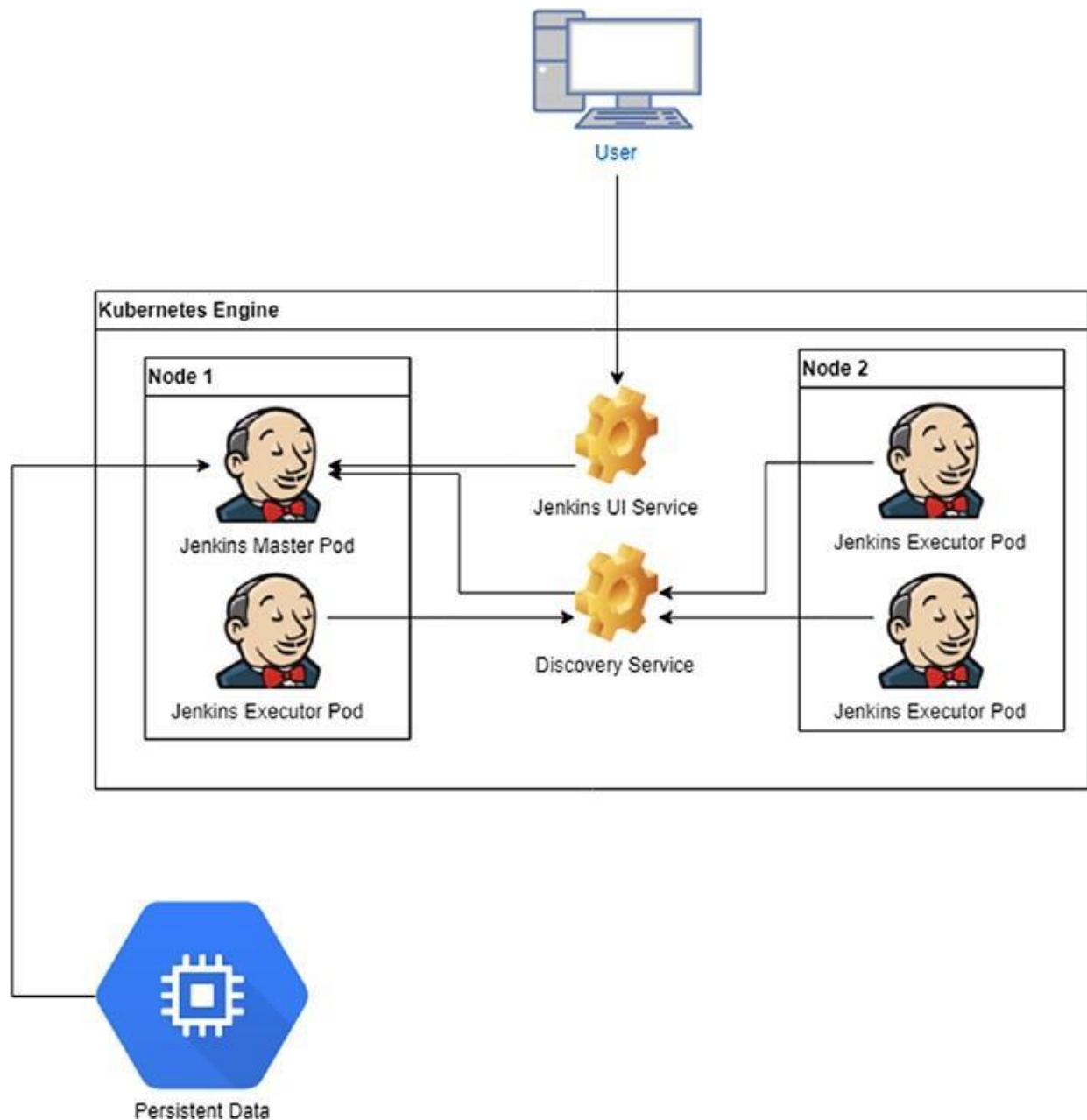


Figure 5-3. *The Jenkins master/slave architecture in Kubernetes*

This architecture essentially deploys Jenkins in a multi-node cluster. We can have more than two nodes. So as not to overload the other networks, we deploy the Jenkins master into a separate *namespace* in the Kubernetes cluster. There are two major advantages to deploying Jenkins in a separate namespace:

- Namespace allows us to create a specific quota.
- A namespace is used to create a logical separation from Jenkins and another deployment.

First, to continue with our deployment, we must better understand the concepts of namespaces, pods, services, quotas, and deployments in Kubernetes.

Namespaces, Pods, Services, Quotas, and Deployments in Kubernetes

Creating Jenkins pipelines in GCP involves an intensive use of Kubernetes. First, to move forward with the configuration of Jenkins, I must define and explain some important concepts related to Kubernetes.

Namespaces

Namespaces is a logical division of our deployment in the cluster. It is important to have different resources logically organized in the cluster.

A Kubernetes cluster can satisfy multiple users or groups of users, which means that different users can have a different project in the same cluster. Using namespaces helps to identify the project for every team. This is done when we satisfy two requirements:

- Provide names for the namespaces
- Launch a mechanism to define and attach different policies and the authorization to access a subsection of the cluster

Each user or group of users may want to create a different isolated environment for our resource, and for this environment, we want to define our own policies and authorization. This is done by creating a namespace. Every namespace allows the user to perform some unique features.

- Uniquely named resources
- Delegated managed authority to defined users
- Ability to limit the resource consumption, using quotas

First, to create our namespace, we want to see if there is another namespace we can use. With Kubernetes, we can see a list of all namespaces, with the following command: `kubectl get namespace`

This command produces an output similar to the following:

```
pierluigi_riti@practicaldevopsgcpcli:~$ kubectl get namespace NAME
```

```
STATUS AGE
```

```
default Active 2d
```

```
kube-public Active 2d
```

```
kube-system Active 2d
```

Kubernetes GCP has three basic namespaces:

- default: This namespace is for all the objects without a specific namespace.
- kube-public: This namespace is readable for all users, including those users who are not authenticated.
- kube-system: This namespace is for the object created by the Kubernetes system.

These namespaces are present in every cluster and are created when we build the cluster itself. When we have established the cluster, we can see a summary of it, using the following command:

```
kubectl get namespace <name>
```

If, for example, we want to see all the details concerning the default namespace, we can use the command `kubectl get namespace default`. The result looks like this:

```
pierluigi_riti@practicaldevopsgcpcli:~$ kubectl get namespace default
```

```
NAME STATUS AGE
```

```
default Active 2d
```

If we want more details, we can use this command:

```
kubectl describe namespace <name>
```

The result follows:

```
pierluigi_riti@practicaldevopsgcpcli:~$ kubectl describe namespace default
Name: default
Labels: <none>
Annotations: <none>
Status: Active
No resource quota.
Resource Limits
Type Resource Min Max Default Request Default Limit Max Limit/
```

Request Ratio

Container cpu - - 100m - - We can see that the command gives us detailed information about the namespace. For devising a namespace in Kubernetes, we must construct a YAML file with the values necessary for creating the namespace. This value is the name of the namespace, which must be compatible with the DNS rules. An example of this file is given in Listing 5-2 .

Listing 5-2. The Code for Creating a Namespace in Kubernetes

```
apiVersion: v1
kind: Namespace
metadata:
```

```
name: practicaldevopsgcpnamespace
```

When the file is ready we can run the command: Kubectl create -f ./<namespace file>.yaml

Pods

A pod is the smallest deployable unit in the Kubernetes world. It is a group of one or more containers, for example, Docker, with shared network/storage. In the Figure 5-3 , for example, the pod shares the network of the node that is installed.

The pod can be used, for example, to run software, such as NGINX. We can use more than one pod to create a stack, but the primary roles of the pod are to operate and support colocated and comanaged software. We use the pod in

our architecture to operate and manage the Jenkins node for our CI/CD pipelines.

Services

A service is an abstraction that defines logical groups of pods and a policy for accessing them. We can view the service as a microservice. Essentially, we create a set of service groupings with different pods. Because every pod is basically a container, we can group different applications to reply as one entity.

To create a service, we must define a YAML file with the parameter for creating the service. For example, we can have a file like the following in Listing 5-3 :

Listing 5-3. A YAML file for Defining a Service

```
kind: Service
apiVersion: v1
metadata:

name: practicaldevopgcp-service
spec:
selector:
app: PracticalDevOpsGCPApp
ports:
- protocol: TCP
port: 80
targetPort: 8080
```

The service file uses the metadata to assign the name to the service itself. In this case, practicaldevopgcp-service. The service points out any pod with the label PracticalDevOpsGCPApp and with the port 8080 exposed. The service exposes a common port, in this case, 80. Because we use a selector, the service creates end points called practicaldevopgcp-service. We can create a service without the selector, in which case, we must manually create an end points file, and this file is necessary to expose the IP and the port for the service (see Listing 5-4).

Listing 5-4. The End Points YAML File Definition

```
kind: Endpoints
apiVersion: v1
metadata:

name: practicaldevopgcp-service subsets:
- addresses:
- IP: 1.2.3.4
ports:
- port: 8080
```

Quotas

Quotas are a way to limit the usage of resources across the Kubernetes cluster. A quota is defined using the object ResourceQuota, and this is initiated by the administrator of the cluster.

The types of resources we can limit are as follows:

- *CPU* : We can limit the number of requests or the number of CPUs used by the pod in a nonterminal state.
- *Memory* : We can limit the number of requests or limit the memory that can be used by the pod in a nonterminal state.
- *Storage* : We can limit the sum of requests across all storage volumes and the number of storage volumes.

Setting the correct limit for the quota is crucial for defining our cluster.

Deployments

Deployments are used to declare and manage the state of the pods and ReplicaSet. We can describe the state of the object in the deployment object, and the deployment controller changes the state of the object at the desired state. This is essentially the basis for defining high availability, because the deployment takes care of the state and, in case the state changes, forces the pods or the ReplicaSet to move in the state defined in the deployment.

Deployment is essentially the heart of a Kubernetes cluster. It is used to create the number of pods we want and to change the state for the pod. A

sample deployment is offered in Listing 5-5 .

Listing 5-5. Sample YAML Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:

name: nginx-deployment
labels:
app: nginx
spec:
replicas: 3
selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx

spec:
containers:
- name: nginx

image: nginx:1.7.9
ports:
- containerPort: 80
```

In this case, we create a new deployment for the NGINX service. We can define the number of pods that define the value replicas, in our case three. This indicates the number of pods the deployment must create.

The deployments define the kinds of containers required to run, in this case, a copy of NGINX version 1.7.9. This is defined in the section containers. The template section defines the label associated with the containers, in this case, NGINX. This means that all the pods use the same name and expose the same port.

Create Jenkins Service

Now that we have discussed the general uses of Kubernetes, we will translate our Jenkins diagram into a set of Kubernetes files. This is necessary to create our CI/CD pipelines. The first piece of our architecture defines the two Jenkins services required by the Kubernetes cluster:

- Jenkins UI
- Jenkins Discovery

The first service we create is the UI. the relevant file is given in Listing 5-6 .

Listing 5-6. The Code for Creating theservice-ui

```
kind: Service
apiVersion: v1
metadata:
  name: jenkins-ui
  namespace: jenkinsgcp

spec:
  type: NodePort
  selector:

  app: master
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      name: ui
```

The code defines a typing service called jenkins-ui, which is defined in the namespacejenkinsgcp. This uses a typeNodePort, which is used to allow an external service and pod access to the Jenkins UI. This service is the first of the two services we must create for our cluster.

The other service we must create is the discovery service. The YAML file for creating this service follows (Listing 5-7):

Listing 5-7. Code for Creating the Discovery Service

```
kind: Service
apiVersion: v1
metadata:

name: jenkins-discovery
namespace: jenkinsgcp
spec:

selector:
app: master
ports:
- protocol: TCP
port: 50000
targetPort: 50000
name: slaves
```

In this case, we expose the port 50000, which is used by the internal Jenkins service to talk to the master and execute the job. The complete file looks like Listing 5-8 .

Listing 5-8. Code for the Complete Service File Created

```
# [START jenkins_service_ui]
--
kind: Service
apiVersion: v1
metadata:
name: jenkins-ui
namespace: jenkinsgcp
spec:
type: NodePort
selector:
app: master
ports:
- protocol: TCP
port: 8080
targetPort: 8080
name: ui
# [END jenkins_service_ui]
```

```
# [START jenkins_service_discovery]
--
kind: Service
apiVersion: v1
metadata:
  name: jenkins-discovery
  namespace: jenkinsgcp
spec:
  selector:
    app: master
  ports:
    - protocol: TCP
      port: 50000
      targetPort: 50000
  name: slaves
# [END jenkins_service_discovery] The last and the most important file we
must create is the deployment file, which is given in Listing 5-9 .
```

Listing 5-9. Deployment File Code for the Kubernetes Jenkinsfile

```
# [START jenkins_deployment] apiVersion: extensions/v1beta1 kind:
Deployment
metadata:

  name: jenkins
  namespace: jenkinsgcp
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: master
    spec:

      containers:
        - name: master
          image: jenkins/jenkins:tls
          ports:
            - containerPort: 8080
```

```
- containerPort: 50000 readinessProbe:
httpGet:
path: /login
port: 8080
periodSeconds: 10
timeoutSeconds: 5
successThreshold: 2 failureThreshold: 5 env:
- name: JENKINS_OPTS valueFrom:
secretKeyRef:
name: jenkins
key: options
- name: JAVA_OPTS
value: '-Xmx1400m'
volumeMounts:
- mountPath: /var/jenkins_home
name: jenkins-home
resources:
limits:
cpu: 500m
memory: 1500Mi
requests:
cpu: 500m
memory: 1500Mi
volumes:
- name: jenkins-home
gcePersistentDisk:
pdName: jenkins-home
fsType: ext4
partition: 1
# [END jenkins_deployment]
```

The deployment file is used to define the service and how many replicas we require for every service. For the master service, we define a replica of 1.

```
spec:
replicas: 1
template:
```

```
metadata:  
labels:  
app: master
```

This is because we want to be sure that only one master at the time is alive in our cluster. In case the node of the cluster fails, Kubernetes starts another node elsewhere in the cluster. The deployment defines the type of container we want to run in this section.

```
spec:  
containers:  
- name: master  
image: jenkins/jenkins:tls  
ports:  
- containerPort: 8080  
- containerPort: 50000
```

This section defines the Docker image we want to use, in this case, the latest version of Jenkins, and the two container ports. Now we define the ports 8080 and 50000. This port used by the UI service is 8080 and that used by the discovery service is 50000.

We can define when the container must be restarted, using the `readinessProbe` section. The `readinessProbe` is very important when we try to spin up a new Docker image, because some images have a lot of data and can't be available immediately. For example, if we spin up a new Jenkins, it can become live in minutes. In this scenario, Docker kills the image. With the `readinessProbe` section, we can indicate the seconds to wait to check beforehand. We also can indicate a time-out and how many times to try before killing the image. The `periodSeconds` parameter indicates how often we try to perform the probe. The `timeoutSeconds` parameter indicates the time to wait after the first probe. The `successThreshold` parameter sets the number of successful probes to consider before the image is live. The `failureThreshold` parameter indicates the number of failures to consider before the image fails to start. The code for this is

```
readinessProbe:  
httpGet:
```



```
path: /login
port: 8080
```

```
periodSeconds: 10
timeoutSeconds: 5
successThreshold: 2
failureThreshold: 5
```

This section defines the parameter used by *kubelet* for identifying when the container is ready to accept traffic. A pod is ready when all containers in the pod are ready.

With `readinessProbe`, we define the parameter we use for monitoring the container and check if the container is alive or not. The first parameter is `periodSeconds`. This value defines how many seconds *kubelet* requires to check the container, in our case every ten seconds.

Some containers require some external parameters, to work correctly. In our deployment, we send this parameter with the following code section:

```
env:
- name: JENKINS_OPTS
  valueFrom:
    secretKeyRef:

name: jenkins
key: options
- name: JAVA_OPTS
  value: '-Xmx1400m'
```

The final two sections are used to define the quota and the volume connected with the Jenkins master. The code for the quota is

```
resources:
  limits:
    cpu: 500m
    memory: 1500Mi

  requests:
    cpu: 500m
```

memory: 1500Mi

The resource section requires a little explanation, particularly about how to define the resource. The first resource we define is the CPU. This has a value of 1, to indicate 1 GCP Core. In our case, because we are using GCP, we set a value of 500m. This value indicates that we want to use a maximum of 500 millicores. The other value we see is the memory. Memory usage is expressed in bytes. We can use a suffix to indicate the value of memory we want to use. These suffixes are E, P, T, G, M, and K. We also can have, as in our case, two letters. In this case, after the suffix, we add the letter i. Here we have we defined a limit of 1500Mi, or 1500 megabytes.

The last part of the resource is the volume. This indicates where the data is stored, in our case, it is split in two parts. The first is volumeMounts:

- mountPath: /var/jenkins_home
name: jenkins-home

This first part indicates what path we want to mount and its name. The second part defines what type of volume we use.
volumes:

- name: jenkins-home
gcePersistentDisk:
pdName: jenkins-home
fsType: ext4
partition: 1

In this case, we define that we use an ext4 file system with only one partition. Now we can define all the parameters we need to create our Jenkins deployments.

Deploying Jenkins on Kubernetes

With the file for the deployment and service ready, we must define some other files, to allow Jenkins to work properly. The first file we must create is

the option file, and inside this file, we can set our password for Jenkins. This file contains only one line:

```
--argumentsRealm.passwd.jenkins=CHANGE_ME --argumentsRealm.roles.jenkins=admin
```

We can set our own password, or we can generate the password at runtime. To generate the password, we can use the following command:

```
openssl rand -base64 15
```

This generates a random password that we can put in our option file. We can now update the password to `CHANGE_ME`, to the value generated for the password. The next step is to create a secret in Kubernetes. To do that, we can use the following command:

```
kubectl create secret generic jenkins --from-file=options  
--namespace=jenkinsgcp
```

Note A Kubernetes secret is used to hold sensitive information, such as a password, OAuth tokens, or SSH keys. A secret offers more security and flexibility than storing the same value in a pod or container.

The result of the command is a simple line advising that the secret has been created correctly.

```
pierluigi_riti@practicaldevopsgcpcli:~/practicalgcp-jenkins$ kubectl create  
secret generic jenkins --from-file=options --namespace=jenkinsgcp secret  
"Jenkins" created
```

With the secret created, we want to add our account to the administrative roles of the role-based access control (RBAC). This gives us the right to manage the cluster. The command for doing that is

```
kubectl create clusterrolebinding cluster-admin-binding -  
clusterrole=cluster-admin --user=$(gcloud config get-value account)
```

The command creates a cluster role binding and adds our actual account to the role of `cluster-admin`. This gives us the right to manage the Kubernetes cluster. The result of this command shows us the user we have added and the result.

```
pierluigi_riti@practicaldevopsgcpcli:~/practicalgcp-jenkins$ kubectl create
clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin
--user=$(gcloud config get-value account)
Your active configuration is: [cloudshell-20307]
clusterrolebinding "cluster-admin-binding" created
```

With all the security configured, what remains is to create the volume we require to use with Jenkins. To create this volume, we can use the following command:

```
gcloud compute images create jenkins-home-image --source-uri https://
storage.googleapis.com/solutions-public-assets/jenkins-cd/jenkins-home-v3.
tar.gz
gcloud compute disks create jenkins-home --image jenkins-home-image
```

The result for the command is shown in Listing 5-10 .

Listing 5-10. Result of the Creation for the Volume

```
pierluigi_riti@practicaldevopsgcpcli:~$ gcloud compute images create
jenkins-home-image --source-uri
https://storage.googleapis.com/solutionspublic-assets/jenkins-cd/jenkins-
home-v3.tar.gz
Created
[https://www.googleapis.com/compute/v1/projects/practicaldevopsgcpcli/
global/images/jenkins-home-image].
NAME PROJECT FAMILY DEPRECATED STATUS jenkins-home-image
practicaldevopsgcpcli READY
```

With the volume created, we can finally run the Kubernetes deployment and the service. The command to run the deployment is

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

This command executes all the YAML files in the folder K8s. The result of this command follows:

```
deployment "jenkins" created
service "jenkins-ui" created
service "jenkins-discovery" created
```

In the cluster, we now can see that we have created one deployment and two services. Next, we must check if the pods are running. We can check the master with the following command:

```
kubectl get pods --namespace jenkinsgcp
```

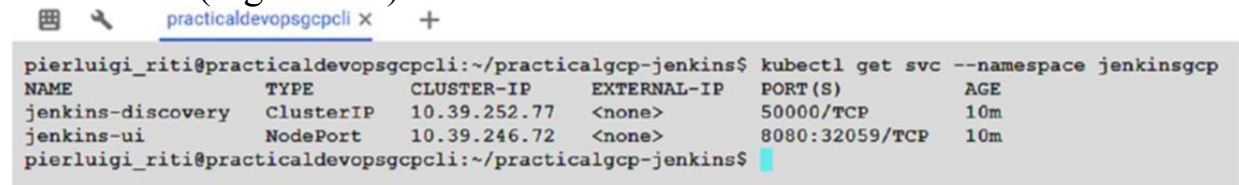
This shows the information about the container.

```
NAME READY STATUS RESTARTS AGE jenkins-87c47bbb8-5mgh4 1/1
ContainerCreating 0 4m
```

We can check the status of the service with this command:

```
kubectl get svc --namespace jenkinsgcp
```

The result of the command shows us the two services and the information about them (Figure 5-4).



```
pierluigi_riti@practicaldevopsgcpcli:~/practicalgcp-jenkins$ kubectl get svc --namespace jenkinsgcp
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
jenkins-discovery    ClusterIP    10.39.252.77   <none>          50000/TCP         10m
jenkins-ui           NodePort     10.39.246.72   <none>          8080:32059/TCP    10m
pierluigi_riti@practicaldevopsgcpcli:~/practicalgcp-jenkins$
```

Figure 5-4. The status of the Jenkins service

Exposing the Service

Now that we have the service configured, what we must do is expose the service to the Internet. Kubernetes offers a very good API system for using the Ingress resource. This is used to allow external resources to access an internal cluster, typically an HTTP resource.

For this, we create a new file, called `ingress.yaml`, in the K8s folder. This file contains all the information we need to expose the service. The file looks like this:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:

name: jenkins
namespace: jenkinsgcp
spec:
```

tls:

- secretName: tls

backend:

serviceName: jenkins-ui

servicePort: 8080

We want a TLS communication to expose the port. For this, we must create our own certificate. We can create the certificate with the openssl command. This command ensures that the certificate is self-signed, so the browser can raise an exception related to that. We must accept the certificate and the browser stop, to raise the error.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /tmp/tls.key  
-out /tmp/tls.crt -subj "/CN=jenkins/O=jenkins"
```

This essentially creates a certificate with a validity of 365 days. The command creates the certificate and puts it into the tmp folder. Now we can use the secret, to store our certificate and open the communication.

```
kubectl create secret generic tls --from-file=/tmp/tls.crt --from-file=  
tmp/tls.key --namespace jenkinsgcp
```

Now that we have the secret, we can run the ingress.yaml file present in the ingress folder.

```
kubectl apply -f ingress/
```

With the ingress created, we can now find the IP associated with it. To do this, we can use this command:

```
kubectl get ingress --namespace jenkinsgcp
```

This operation can take several minutes to finish. As a result of that, GCP creates a new load balancer. This is used to expose Jenkins to our infrastructure. The beauty and power of GCP takes care of creating the necessary infrastructure to expose the service for us, in this case, via the load balancer. The result shows a table with the IP associated with the Jenkins UI.

NAME	HOSTS	ADDRESS	PORTS	AGE
------	-------	---------	-------	-----

jenkins	*	35.x.x.x	80, 443	2m
---------	---	----------	---------	----

The IP we see is the IP we can use to access our Jenkins UI. First, to use the UI, we must check with the load balancer where the instance is healthy. We can use the following command to check against the load balancer:

```
kubectl describe ingress jenkins --namespace jenkinsgcp
```

This command returns the details of the cluster, and with those, we can see when the UI is available (Figure 5-5).

```
Welcome to Cloud Shell! Type "help" to get started.
pierluigi_riti@practicaldevopsgcpcli:~$ kubectl describe ingress jenkins --namespace jenkins
Name:          jenkins
Namespace:     jenkins
Address:       35.186.199.190
Default backend: jenkins-ui:8080 (10.36.2.6:8080)
TLS:
  tls terminates
Rules:
  Host  Path  Backends
  ----  ---  -
  *    *    jenkins-ui:8080 (10.36.2.6:8080)
Annotations:
  backends:      {"k8s-be-31846--2da4de3aa48e7c04":"HEALTHY"}
  forwarding-rule: k8s-fw-jenkins-jenkins--2da4de3aa48e7c04
  target-proxy:   k8s-tp-jenkins-jenkins--2da4de3aa48e7c04
  url-map:        k8s-um-jenkins-jenkins--2da4de3aa48e7c04
Events:
  Type    Reason    Age          From          Message
  ----    -
  Normal  Service  6m (x16 over 2h)  loadbalancer-controller  default backend set to jenkins-ui:31846
pierluigi_riti@practicaldevopsgcpcli:~$
```

Figure 5-5. The Kubernetes health status

We can see that the load balancer created in Google Console moves to Networking Services and then selects the load balancing choice from the menu. To view the details of the load balancer, click that, and something similar to Figure 5-6 is shown.

The screenshot shows the 'Load balancer details' page in the Google Cloud Platform console. The breadcrumb is 'Load balancer details'. There are 'EDIT' and 'DELETE' buttons. The load balancer name is 'k8s-um-jenkins-jenkins--2da4de3aa48e7c04'. Below the name are tabs for 'Details', 'Monitoring', and 'Caching'. The 'Frontend' section shows a table with one entry: HTTP protocol, IP:Port 35.186.199.190:80, no certificate, and Premium network tier. The 'Host and path rules' section shows a table with one entry: All unmatched (default) path, All unmatched (default) paths, and k8s-be-31846--2da4de3aa48e7c04 back end. The 'Backend' section shows 'Backend services' with one service: k8s-be-31846--2da4de3aa48e7c04. Below this, it shows 'Endpoint protocol: HTTP', 'Named port: port31846', 'Timeout: 30 seconds', 'Cloud CDN: disabled', and 'Health check: k8s-be-31846--2da4de3aa48e7c04'. There is a link for 'Advanced configurations'. At the bottom, there is a table for the backend service instances.

Instance group	Zone	Healthy	Auto-scaling	Balancing mode	Capacity
k8s-ig--2da4de3aa48e7c04	us-east1-b	0 / 0	Off	Max. RPS: 1 (per instance)	100%

Figure 5-6. The load balancer details created by GCP

With the health status of the cluster, we can now access Jenkins. To access the Jenkins UI, we can use the IP of the load balancer, in my case, 35.186.199.190. This shows us the Jenkins login page (Figure 5-7).

The screenshot shows the Jenkins web interface. The header has the Jenkins logo and a search bar. Below the header, there is a sidebar with links: People, Build History, Project Relationship, Check File Fingerprint, and Credentials. The main content area says 'Welcome to Jenkins!' and has a button that says 'Log in to create new jobs.'.

Figure 5-7. The Jenkins page run on the cluster

For access, we can use the username Jenkins; the password is the same one we defined in the file. We are now ready to create our CD pipelines.

Creating a Continuous Delivery Project

With the Jenkins page up and running, we can finally create our CD pipeline. To try out our Jenkins feature, we create a very simple app, as follows:

For trying the feature of Continuous Delivery in GCP we use the example code build by Google, this is a very simple page showing the information about the system because what we really need is understand only how to create the environment, the project is based on the *gceme* image present in the Google repository, gcr.io/cloud-solutions-images/gceme

For creating a CD pipeline, we first must create the different environment we use to release the software. We actually create three environments.

- *Production* : This is the environment in which we release the software for production.
- *Services* : We use the service environment to describe how many layers we have. In our case, we have a back end and front end. This is not an environment like production or canary, but it is something we can use to have a more logical division of our service layers. This allows better management of that.
- *Canary* : This creates the canary server used for the system.

Note A canary server is a server used in CD to test a feature in a real environment before it is released for production. A canary server emulates the production environment, only in a limited manner. The canary server can be used to isolate errors in code and, normally, for a limited number of users selected for production purposes. Using these servers is useful not only to uncover errors but to test the UI/UX of the application.

These environments are used to simulate the phases of CD pipeline development. To simulate them, we first must create a new namespace in Kubernetes.

```
kubectl create ns practical-gcp-production
```

This command creates a new namespace in the Kubernetes cluster. We use this namespace to build our production environment. With the new namespace created, we can now create the services necessary for the code.

The first service we create is for the production. The production has two services: back end and front end (see Listing 5-11).

Listing 5-11. Kubernetes File for Creating the Back-End Service in the Production Namespace

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:

name: practical-gcp-backend-production
spec:
replicas: 1
template:
  metadata:
    name: backend
  labels:
    app: gceme
    role: backend
    env: production
  spec:
    containers:
    - name: backend
      image: gcr.io/cloud-solutions-images/gceme:1.0.0
      resources:
        limits:
          memory: "500Mi"
          cpu: "100m"
        imagePullPolicy: Always
      readinessProbe:
        httpGet:
          path: /healthz
          port: 8080
      command: ["sh", "-c", "app -port=8080"]
      ports:
      - name: backend
        containerPort: 8080
```

The preceding is the Kubernetes file for creating the back-end service for the production. Now we can see that the service uses the applicationgceme, version 1.0.0.

containers:

- name: backend

image: gcr.io/cloud-solutions-images/gceme:1.0.0

resources:

limits:

memory: "500Mi"

cpu: "100m"

imagePullPolicy: Always

The image is pulled from the Google repository and built in the Kubernetes file. The other service we need to create is the front-end service (Listing 5-12).

Listing 5-12. Front-End Production Service

kind: Deployment

apiVersion: extensions/v1beta1

metadata:

name: practical-gcp-frontend-production

spec:

replicas:

template:

metadata:

name: frontend

labels:

app: gceme

role: frontend

env: production

spec:

containers:

- name: frontend

image: gcr.io/cloud-solutions-images/gceme:1.0.0

resources:

```
limits:
memory: "500Mi"
cpu: "100m"
imagePullPolicy: Always
readinessProbe:
httpGet:
path: /healthz
port: 80
command: ["sh", "-c", "app -frontend=true -backend-service=http:// gceme-
backend:8080 -port=80"]
ports:
- name: frontend
containerPort: 80
```

These files are in the same folder, and we can use them for a deployment. A deployment is used to create and set a specific state for a pod and ReplicaSet. We use the following command to deploy our application:

```
kubectl --namespace=practical-gcp-production apply -f k8s/production
```

The result of the command is

```
deployment gceme-backend-production" created
deployment "gceme-frontend-production" created
```

Now, when we create the service, we can create the other deployments for the service and the canary. Essentially, we tell Kubernetes to create one Kubernetes service, the services line, and one deployment for the canary service. The first command creates a service. A service in Kubernetes is a REST object similar to a pod. When we create a service, we essentially POST the resource to the API server, to create the new resource. When we create the canary, we essentially create a new deployment. In Kubernetes, this follows the rules for every deployment: the application goes under the previous Kubernetes application. In this case, we can use the load balancer to deploy to the Internet and be reachable from the outside.

```
kubectl --namespace=practical-gcp-production apply -f k8s/services kubectl
--namespace=practical-gcp-production apply -f k8s/canary
```

We don't change the namespace, because we want to use the same environment. When all the deployments are created, we can scale the production for the front end. We do this because we want to have multiple instances running in our front-end application.

```
kubectl --namespace=practical-gcp-production\  
scale deployment practical-gcp-frontend \  
--replicas=4
```

When we have scaled the deployment, we can determine the external IP assigned to our application with the following command:

```
kubectl --namespace=practical-gcp-production get service practical-  
gcpfrontend
```

This command shows the IP for the service we can use to access our application (Figure 5-8).

```
pierluigi_riti@practicaldevopsgcpcli:~/sample-app$ kubectl --namespace=practical-gcp-production get service practical-gcp-frontend  
NAME                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE  
practical-gcp-frontend  LoadBalancer  10.39.242.99   35.196.130.194  80:31284/TCP     1m  
pierluigi_riti@practicaldevopsgcpcli:~/sample-app$
```

Figure 5-8. The IP for our application, deployed in Kubernetes

We can see when the application being run has put the public IP in the browser. It will look something like Figure 5-9.

Backend that serviced this request	
Name	gke-jenkins-cd-default-pool-004eeb4b-wcmt
Version	1.0.0
ID	1945285849896574662
Hostname	gke-jenkins-cd-default-pool-004eeb4b-wcmt.c.practicaldevopsgcpcli.internal
Zone	us-east1-b
Project	practicaldevopsgcpcli
Internal IP	10.142.0.4
External IP	35.231.174.253
Proxy that handled this request	
Address	
Request	GET / HTTP/1.1 Host: gceme-backend:8080 Accept-Encoding: gzip User-Agent: Go-http-client/1.1
Error	

Figure 5-9. *The application up and running*

Creating the Repository

The next step in our CD development process is to create the repository in which we can put the code to use for our pipelines. In Google Cloud, we have a Google Cloud Repository. This is a private Git repository that we can use for maintaining our code. We must create a repository in which to put our code. To do this, open the Google

Cloud Console, and in the find text box, type *repository* . This yields a drop-down menu. Select Source code, which opens the window for the source repository (Figure 5-10).

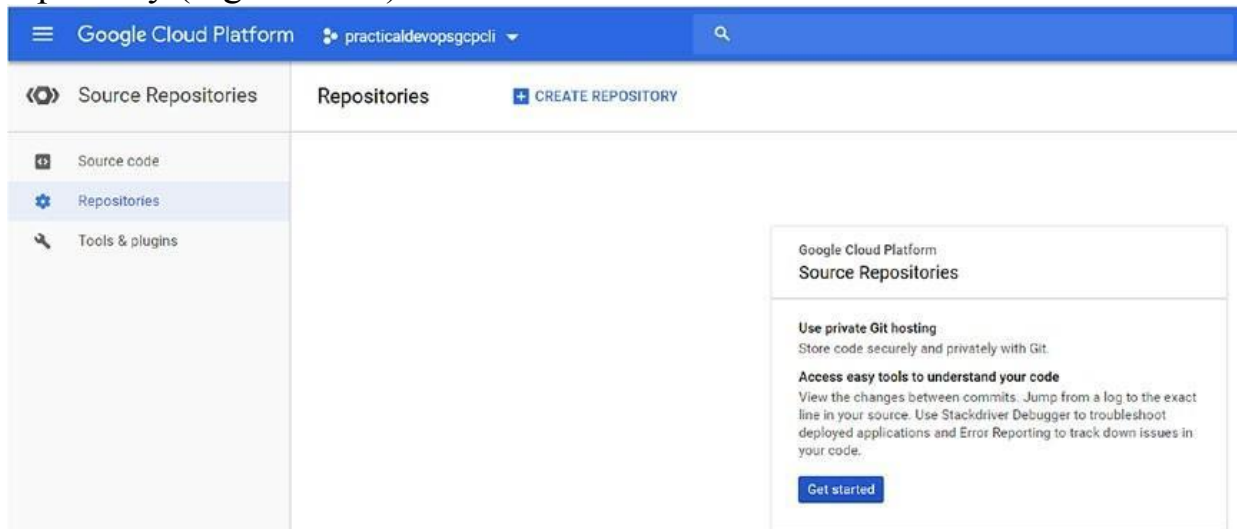


Figure 5-10. The Source Repositories service in Google Cloud Platform
Create a new repository called *practicaldevopsgcp* . With the repo created, we see something like Figure 5-11 .

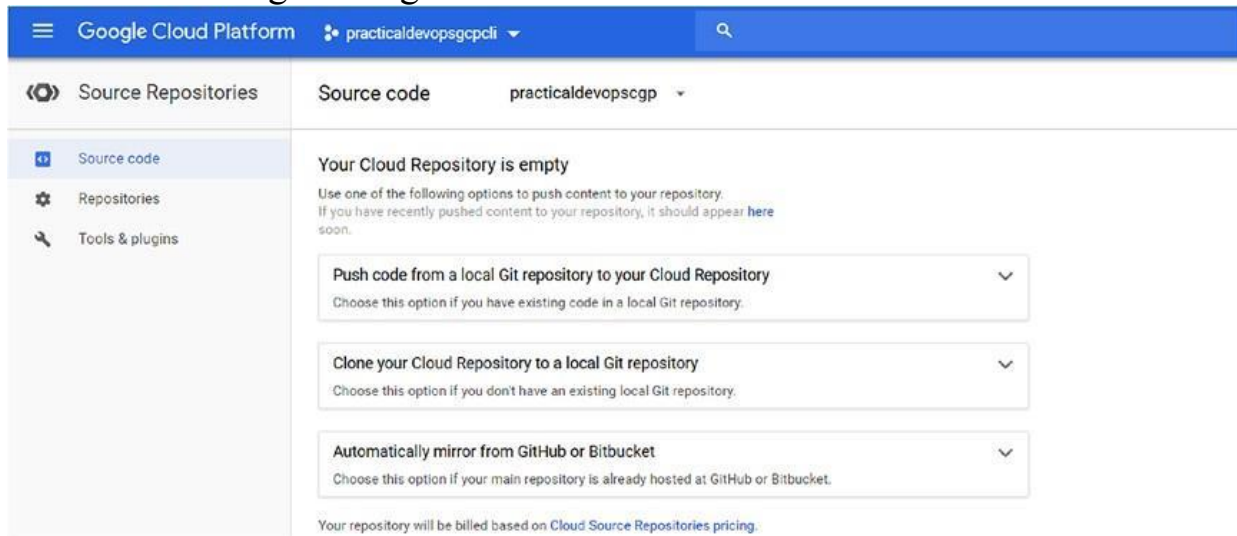


Figure 5-11. The source code with the new repository

With the repo created, we can now start to initialize the Git and push the code on the remote repository, insert the *sample-app* folder, and use the command for initializing the repository. To do this, we must execute these commands:

```
gcloud init && git config credential.https://source.developers.google.com.  
helper gcloud.cmd
```

This creates the basic configuration for our Git repository. After that, we must make our repository remote, with this command:

```
git remote add google https://source.developers.google.com/p/  
practicaldevopsgcpcli/r/practicaldevopscgp
```

With the repo created and configured, we can start to push the first file in the repo. First, we must add the file to the local repository.

```
git add .
```

```
git commit -m "First commit"
```

The preceding commands are used to add and commit the file in our local repository. With the file committed, we now must push the file in the remote repo.

```
git push --all google
```

This pushes all the code we have in the remote repository and creates a new branch called master (Figure 5-12).

```
pierluigi_riti@practicaldevopsgcp-197023:~/continuous-deployment-on-kubernetes/sample-app$ git push --all google  
git: 'credential-gcloud.cmd' is not a git command. See 'git --help'.  
Counting objects: 21, done.  
Compressing objects: 100% (21/21), done.  
Writing objects: 100% (21/21), 5.95 KiB | 0 bytes/s, done.  
Total 21 (delta 9), reused 0 (delta 0)  
remote: Resolving deltas: 100% (9/9)  
To https://source.developers.google.com/p/practicaldevopsgcpcli/r/practicaldevopscgp  
* [new branch]      master -> master  
pierluigi_riti@practicaldevopsgcp-197023:~/continuous-deployment-on-kubernetes/sample-app$
```

Figure 5-12. The first commit in our repository

We have finally committed all our files to the remote repository. We can manage and view the information about the file directly from the Google platform. We just need to connect to the platform and move to the repository section (Figure 5-13).

Source code			
<div>practicaldevopscgp master</div>			
/ ▾			
Name	Latest commit	Author	Date (UTC+1)
📁 k8s	First commit	Pierluigi Riti	14:58
📄 Dockerfile	First commit	Pierluigi Riti	14:58
📄 html.go	First commit	Pierluigi Riti	14:58
📄 Jenkinsfile	First commit	Pierluigi Riti	14:58
📄 main.go	First commit	Pierluigi Riti	14:58
📄 main_test.go	First commit	Pierluigi Riti	14:58

Figure 5-13. The repo with the file committed

Creating Jenkins Pipelines

With the repository complete, we must create a pipeline for CD in Jenkins. For this, we set up the credential we must use for our service. Because it is an automatic service, and we don't have a user connection to the UI, we want to provide a different credential for the service. To add a credential, connect to the Jenkins UI, open the Credentials section, and click the link Global, which opens a page on which to add the new credential on the server (Figure 5-14).



Figure 5-14. *The Credentials section in Jenkins*

Click the Add Credentials link on the left panel of the window. This opens another page, on which we can select the credential. Select Google Service Account from metadata (Figure 5-15).

Figure 5-15. *The credential for the Google account*

Click OK, to add the credential for the service. This adds the credential to the system. Because Jenkins runs under GKE, Google exposes the credential. This credential can automatically get by Jenkins and configure in it. The credential now looks like Figure 5-16 .

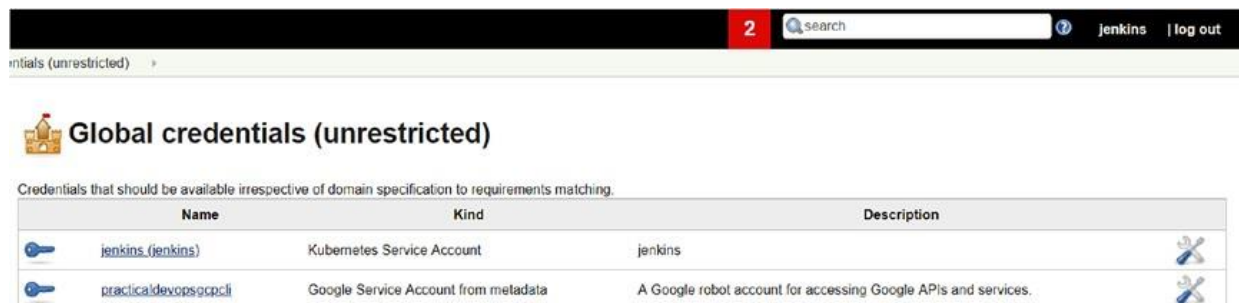


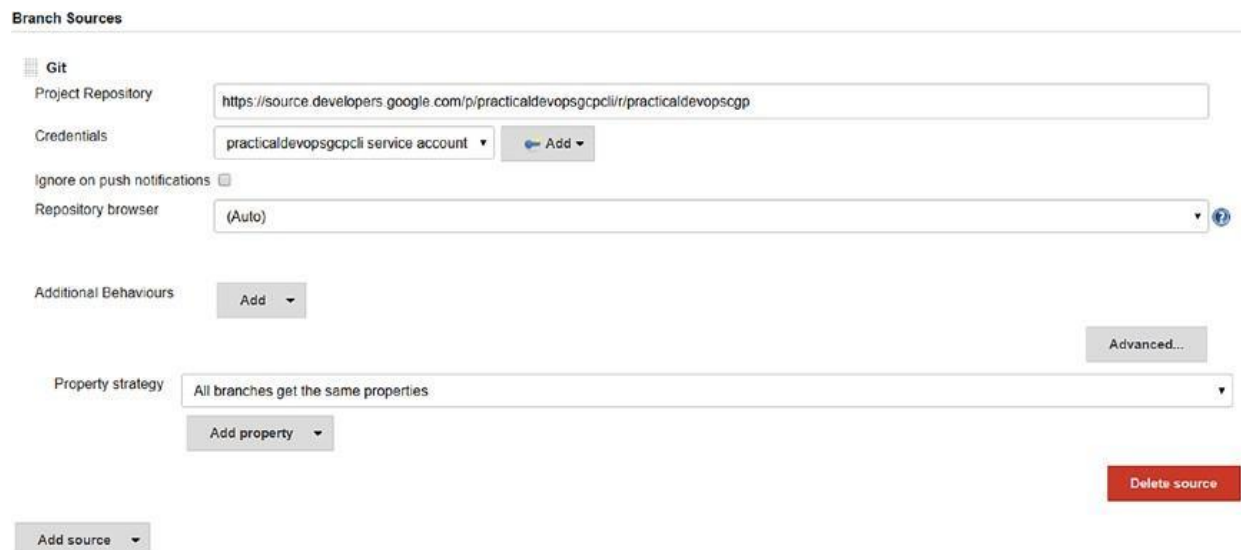
Figure 5-16. *The updated credential*

With the credential configured, what we need now is to create the jobs for the build. In the Jenkins UI, select the link to create the new job and then

select the Multibranch Pipeline type of project. This opens the configuration section for the project (Figure 5-17).

Figure 5-17. *The multibranch project configuration section*

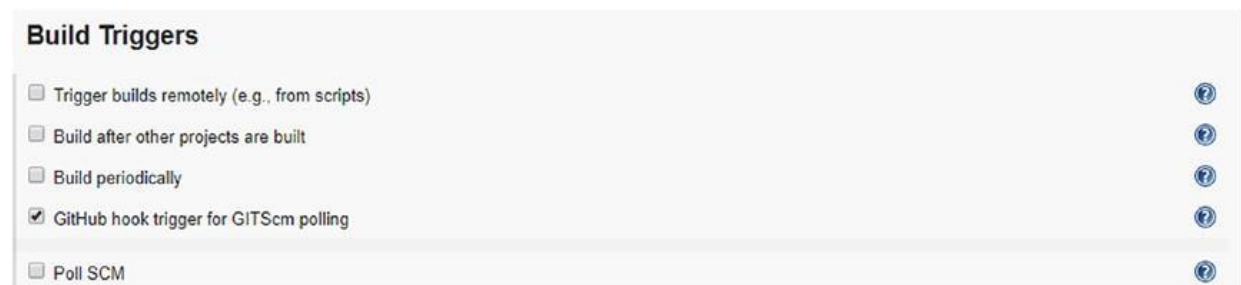
For adding the source code for the pipelines, must click the Add Source drop-down, then select the Git repository, the credential created previously, and, finally, the link for the code repository we previously created. In the end, we should see something like Figure 5-18 .



The screenshot shows the 'Branch Sources' configuration page in Jenkins. It is for a 'Git' project. The 'Project Repository' field contains the URL 'https://source.developers.google.com/p/practicaldevopsgcpcli/r/practicaldevopscgp'. The 'Credentials' dropdown is set to 'practicaldevopsgcpcli service account'. There is an 'Add' button next to the credentials dropdown. The 'Ignore on push notifications' checkbox is unchecked. The 'Repository browser' dropdown is set to '(Auto)'. There is an 'Add' button for 'Additional Behaviours'. A 'Property strategy' dropdown is set to 'All branches get the same properties', with an 'Add property' button below it. An 'Advanced...' button is on the right. At the bottom left is an 'Add source' dropdown, and at the bottom right is a red 'Delete source' button.

Figure 5-18. *The Jenkins GIT configuration section*

Now we must define when to execute the build. Because we want to have CD, we trigger the build every time we commit on the repository. This practice leads the CD, and we can have hundreds of builds per day. To configure Jenkins to do this, we must select the GitHub hook trigger for GITScm polling option (Figure 5-19).



The screenshot shows the 'Build Triggers' configuration page in Jenkins. It lists several options with checkboxes and help icons: 'Trigger builds remotely (e.g., from scripts)', 'Build after other projects are built', 'Build periodically', 'GitHub hook trigger for GITScm polling' (which is checked), and 'Poll SCM'.

Figure 5-19. *The configuration of the GitHub trigger*

On the main page, we can see the log of the GitHub hook, because we have

connected our Jenkins with Git (Figure 5-20).

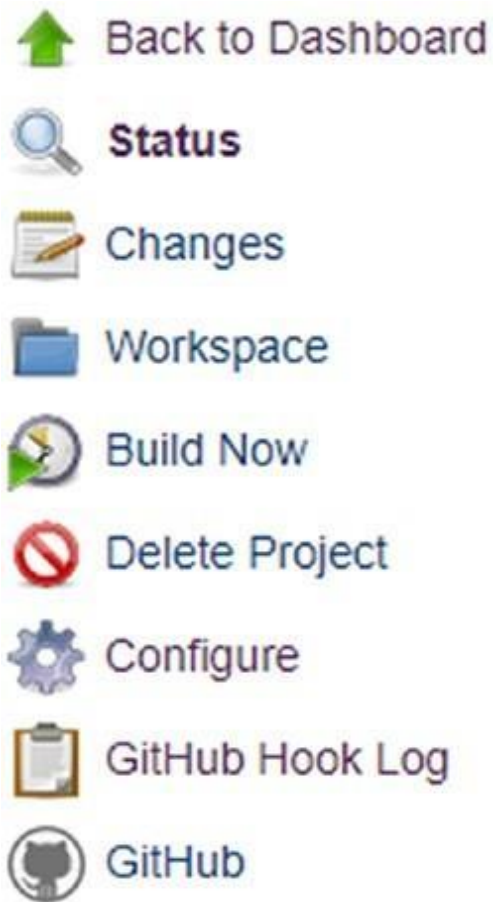


Figure 5-20. The menu with the GitHub

hook log

Save the configuration. This starts our first build in the branch (Figure 5-21).

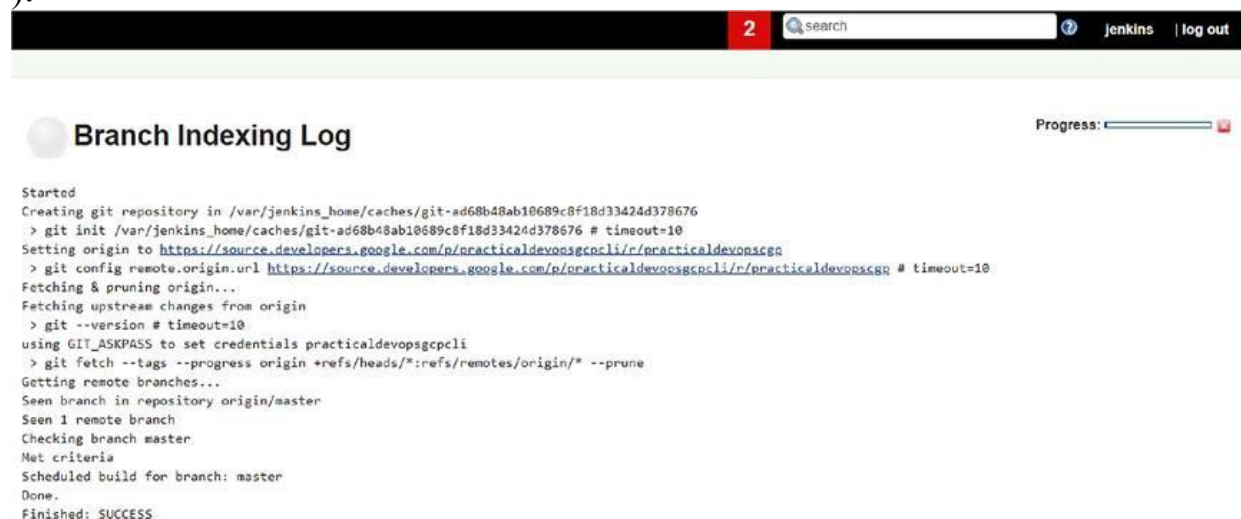


Figure 5-21. The first build in Jenkins

Because ours is a multipipeline project, we must configure a Jenkinsfile to

configure and manage the build. What you must learn now is how to create the Jenkinsfile.

Creating the Jenkinsfile

We now have Jenkins up and running on the server. Now we must create the structure for the application and the CD pipelines.

To construct a CD pipeline with Jenkins, we must create a pipeline project and a Jenkinsfile. This file is used to describe the branch we want to deploy and must be placed on every single branch we create. You will see more details regarding these steps when I discuss the microservice. For now, it is important to understand the anatomy of a Jenkinsfile (Listing 5-13).

Listing 5-13. The Jenkinsfile Necessary for Continuous Deployment

```
node {
def project = 'practicaldevopsgcp-197023'
def appName = 'pdopsgcp'
def feSvcName = "${appName}-frontend"
def imageTag =
"gcr.io/${project}/${appName}:${env.BRANCH_NAME}.${env.
BUILD_NUMBER}"
checkout scm

stage 'Build image'
sh("docker build -t ${imageTag} .")
stage 'Run Go tests'
sh("docker run ${imageTag} go test")
stage 'Push image to registry'
sh("gcloud docker -- push ${imageTag}")

stage "Deploy Application"
switch (env.BRANCH_NAME) {
// Roll out to canary environment
case "canary":
// Change deployed image in canary to the one we just built sh("sed -i.bak
's#gcr.io/cloud-solutions-images/
```

```
pdopsgcp:1.0.0#${imageTag}#' ./k8s/canary/*.yaml")
sh("kubectl --namespace=production apply -f k8s/services/") sh("kubectl --
namespace=production apply -f k8s/canary/") sh("echo http://`kubectl --
namespace=production get service/${feSvcName} --output=json | jq -r
'.status.loadBalancer. ingress[0].ip' > ${feSvcName}")
break
```

```
// Roll out to production
case "master":
// Change deployed image in canary to the one we just built
sh("sed -i.bak 's#gcr.io/cloud-solutions-images/
pdopsgcp:1.0.0#${imageTag}#' ./k8s/production/*.yaml")
sh("kubectl --namespace=production apply -f k8s/services/")
sh("kubectl --namespace=production apply -f k8s/production/")
sh("echo http://`kubectl --namespace=production get
service/${feSvcName} --output=json | jq -r '.status.loadBalancer.
ingress[0].ip' > ${feSvcName}")
break
```

```
// Roll out a dev environment
default:
// Create namespace if it doesn't exist
sh("kubectl get ns ${env.BRANCH_NAME} || kubectl create ns ${env.
BRANCH_NAME}")
// Don't use public load balancing for development branches sh("sed -i.bak
's#LoadBalancer#ClusterIP#' ./k8s/services/frontend. yaml")
sh("sed -i.bak 's#gcr.io/cloud-solutions-images/
pdopsgcp:1.0.0#${imageTag}#' ./k8s/dev/*.yaml")
sh("kubectl --namespace=${env.BRANCH_NAME} apply -f k8s/services/")
sh("kubectl --namespace=${env.BRANCH_NAME} apply -f k8s/dev/")
echo 'To access your environment run `kubectl proxy`' echo "Then access
your service via http://localhost:8001/api/v1/
proxy/namespaces/${env.BRANCH_NAME}/services/${feSvcName}:80/"

}
}
```

This file is used by the pipelines project to build the software we made. Now we can see the file, called a Dockerfile. This is because we build Docker images for our software.

The file is used to define the stage at which the pipelines can be executed automatically by Jenkins, in particular the files we found in the following phases:

- *Build* : Used to create the Docker image with the last code
- *Run* : Runs some tests against the image just created
- *Push* : Pushes the image on the repo, if the tests have passed
- *Deploy* : Used to deploy the application on different servers

These steps are the basis of our pipelines (Figure 5-22). Now we can configure some parameters for the branch. These can be changed every time a developer creates a new branch, and they represent the best way of creating a multibranching policy.

Caution Be careful when selecting the project ID in the Jenkinsfile. You can get the information from the home page of our GCP.

Pipeline master

Full project name: practicaldevops-gcp/master



[Recent Changes](#)

Stage View

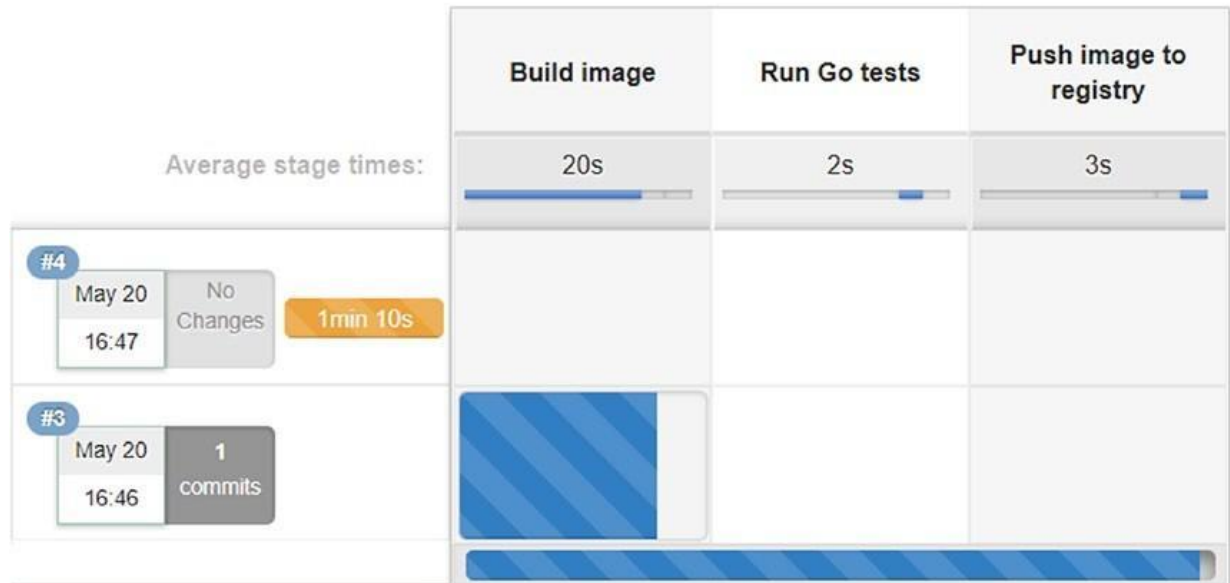


Figure 5-22. The built pipelines

Now we can see that Jenkins creates the pipelines, based on the stage we define in the Jenkinsfile.

stage 'Build image'

```
sh("docker build -t ${imageTag} .")
```

stage 'Run Go tests'

```
sh("docker run ${imageTag} go test")
```

stage 'Push image to registry'

```
sh("gcloud docker -- push ${imageTag}")
```

Now, when we make any changes to the code, we execute all the pipelines, from beginning to end, which allows us to create a complete software, starting from the code.

Conclusion

This chapter covers how to create pipelines projects for CD. This is very important when we want to implement a complete DevOps resource. Jenkins offers a very effective tool for implementing CD, and with Kubernetes in the GCP, we can create a very powerful CD system. With our new multibranch project, we can have a graphical representation of its stages. Jenkinsfile is a very easy and powerful means of configuring the CD system. We can use code to define every stage and every environment we require. We can release the file with the code, and in this way, every developer has the system correctly configured, which reduces the time for maintenance and, at the same time, improves the stability of the project.

CHAPTER 6

Microservices Architecture with GCP

Every day, we probably use a service based on a microservices architecture. Every time we watch a Netflix series, we essentially use a service based on a microservice. A microservice offers more advantages than a classic architecture and is perfect for a cloud application. In this chapter, you will learn how to design a microservices architecture and how to implement it in Google Cloud Platform (GCP).

Introduction to Microservices Architecture

Microservices architecture is a programming technique derived directly from serviceoriented architecture (SOA). Microservices architecture is, in fact, a variation of that style of architecture. In microservices architecture, we create services that are *loosely coupled* , *fine-grained* , and connected with a lightweight protocol.

A loosely coupled service doesn't have to know more about the definition of the other services. It needs to know only how to call and what it gets back;

no other details are needed. This means that we can change the implementation of the service without any problems.

The service is normally fine-grained. Granularity is a key consideration that must be made when we design a service, as it defines the business functionality and the payload message between the services. In a fine-grained service, we can define a small business case in which the payload message can be big or small. This depends on how many transactions we have. For example, we can define a microservices user for which we define the interface for creating and reading the user.

Adopting a microservices architecture helps to decompose big architecture into smaller pieces. This helps to improve the scalability of the architecture. Microservice is normally deployed with a container architecture. This means that we can add services to respond to the high traffic and remove them when they are no longer needed.

Another advantage of this architecture is that it allows the team to write code in parallel and deploy it independently. This is because, for the test, we just need to know the data for the input and the data we receive from the output. Because the entire service is independent, we can, and must, implement the continuous integration and deployment easily. For microservice to be successful, it is essential to adopt continuous integration and deployment.

Following are some principles associated with microservices architecture:

- *Microservices is designed for big systems* : A microservice is born of the necessity to scale a big system, but how a big system is defined is relative. The important idea behind microservices is to design a system to be reactive to changes and improve scalability of the system itself.
- *Microservices is goal-oriented* : When we design a microservices architecture, we don't have to follow set rules, but we can devise solutions tailored to resolve specific problems. This becomes more clear when, for example, we must add another service to an existing architecture. In that case, we must consider how to design the service and what service we must use to implement the architecture.

- *Microservices is designed to be replaced* : When we think of microservices architecture, we design a service to be loosely coupled to others, which allows us to change a service for another, without creating problems for the system.

These principles derive directly from the nature of microservices architecture, because when we define the service, we design an architecture for modularity. Most microservices architectures use Docker or a similar container technology to define and deploy the service. In this way, we can easily deploy a new service in the architecture, to fix a bug or to scale the architecture.

Microservices architecture naturally leads the process of CI/CD. This is connected to the implementation of the architecture. When we implement the architecture, we define a set of services designed to exchange a small amount of data. Normally, these are designed to be released easily and drive the CI/CD process. We continuously integrate the software and release new services ready for production.

Implementing a Microservices Architecture

When we think about microservices architecture, we must think about all the cultural changes we must put in place for a correct implementation. When we implement a microservices architecture, we make a choice to have a small service designed to respond to the business need of communicating with a minimum amount of data. This requires some cultural change to accommodate that.

This change is, in particular, connected to the philosophy behind the microservices architecture, which is the Unix philosophy to “do one thing and do it well.” Based on this, we can define the philosophy and the architectural principles of the microservices.

- The service has a fine granularity that serves only one function.
- We must embrace a continuous testing and continuous integration policy.
- When we design the system, we must embrace the failure and the faults that can occur on the system and, based on that failure, improve our system.

- The service must have some specific constraints. It must be
- *Elastic* : Each service must be able to scale up and down.
- *Resilient* : In case a service fails, the failure doesn't affect another service in the system. In some systems, we can have one microservice depend on another, in which case we must be sure to manage the failure of the dependencies.
- *Composable* : Because the microservices are loosely coupled, each service must offer a uniform interface that doesn't change over time.
- *Minimal* : Each service is designed for doing one thing and doing it well. Therefore, the service is designed to be a highly cohesive entity, with each part specialized in only one thing.
- *Complete* : Each microservice must have complete functionality, because each service is specialized.

These architectural practices require a change in company culture. This change is similar to that required for adopting DevOps practices. When we implement a microservices architecture, there are some pros and cons that must be considered (Table 6-1):

Table 6-1. Pros and Cons of a Microservice Architecture

Pros

Strong modularity : When we move from a monolithic service to a microservice, we design the application to be modular and divisible into small modules. This becomes critical where there is a large crossdistributed team.

Independent deployment : When we design a microservice, we design a small, independent service. By its nature, the service can be released and deployed independently from others.

Language independence : As each service is deployed and developed independently from each other, we can choose any language we want for developing the service. This means we can have potentially a different language for every service we develop.

Cons

Microservices are distributed : Developing a distributed system can be more difficult, because the system can be geographically hosted in another place. This can create an issue with response times. Another issue is determining if a service is alive or not.

Consistency : When we deploy the system, we must be sure that the service is correctly deployed across the instances and have the consistency of the interface. We must remember that the microservices talk to each other via messages. If we change something, this can corrupt the system.

Complexity of operations : When we design a microservices architecture, we have hundreds of services connected to one another. This increases the complexity of the operativity of the system. We need a very experienced team to maintain the architecture.

Microservices Architecture with Docker and Kubernetes

When I think about a microservices architecture in GCP, I usually think of Docker and Kubernetes. My choice of stack derives from the goal I want to achieve. I want to design a modular, completely independent service-based architecture, I want to use some RESTful web service, managed and deployed using the CD principle.

When we consider and design an architecture using Kubernetes, the natural choice is Docker. With Docker, we can create a level of isolation we need for our microservices architecture. The first thing to bear in mind in the design of the architecture we want to implement is how it is different from a monolithic architecture. Figure [6-1](#) illustrates the differences.

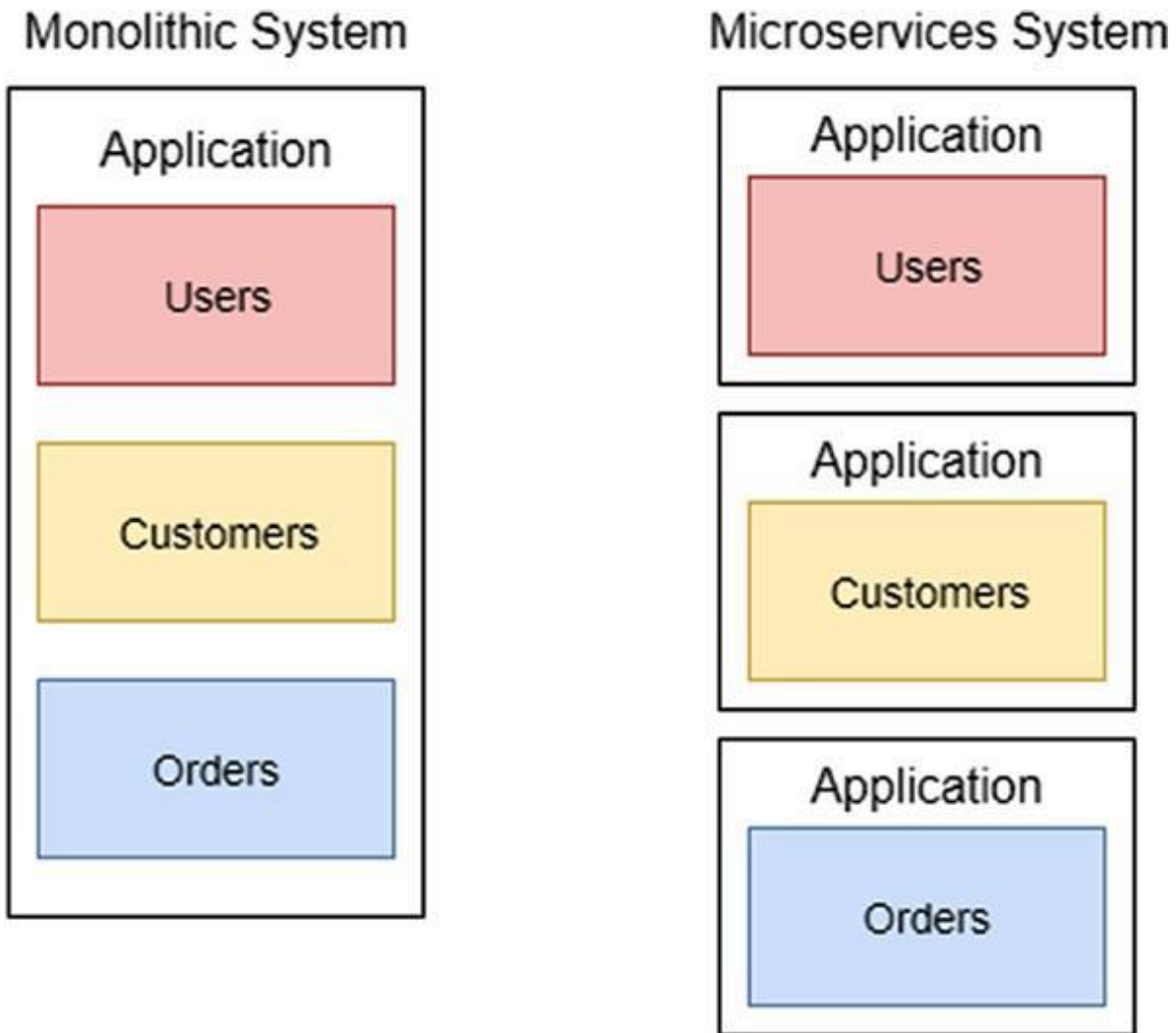


Figure 6-1. *A monolithic system vs. a microservices system*

When we move from a monolithic application to a microservices application, we must think about each piece of the system as a separate application.

Each application, for example, for orders, customers, and users, is not a simpler component of the application but a small application itself. This is the first major change that we must reflect in our architecture. To achieve that shift, we can use Docker and Kubernetes.

This kind of application can be defined by Container as a Service (CaaS), in which each Docker is a single application, and Kubernetes helps to manage them all.

As was noted previously, microservices architecture has some specific characteristics.

- *Independence* : Each service must be independent of another.
- *Decentralization* : The service can be installed on a different server without impacting the application.
- *Message-based* : Each service uses a message to communicate to another.
- *Automatically released* : When we implement a microservices architecture, we usually use the practice of CD to release the application.
- *Isolated* : Each service must be isolated from another

These four principles drive the choice for Docker and Kubernetes. With Docker, we can create independent containers that talk to each other only via messages and are released in accordance with CD practices.

Designing a Kubernetes Microservices Architecture

When we design a microservices architecture in Kubernetes, we must “translate” in the Kubernetes world what we need to implement. An example of a microservices application in Kubernetes is shown in Figure [6-2](#) .

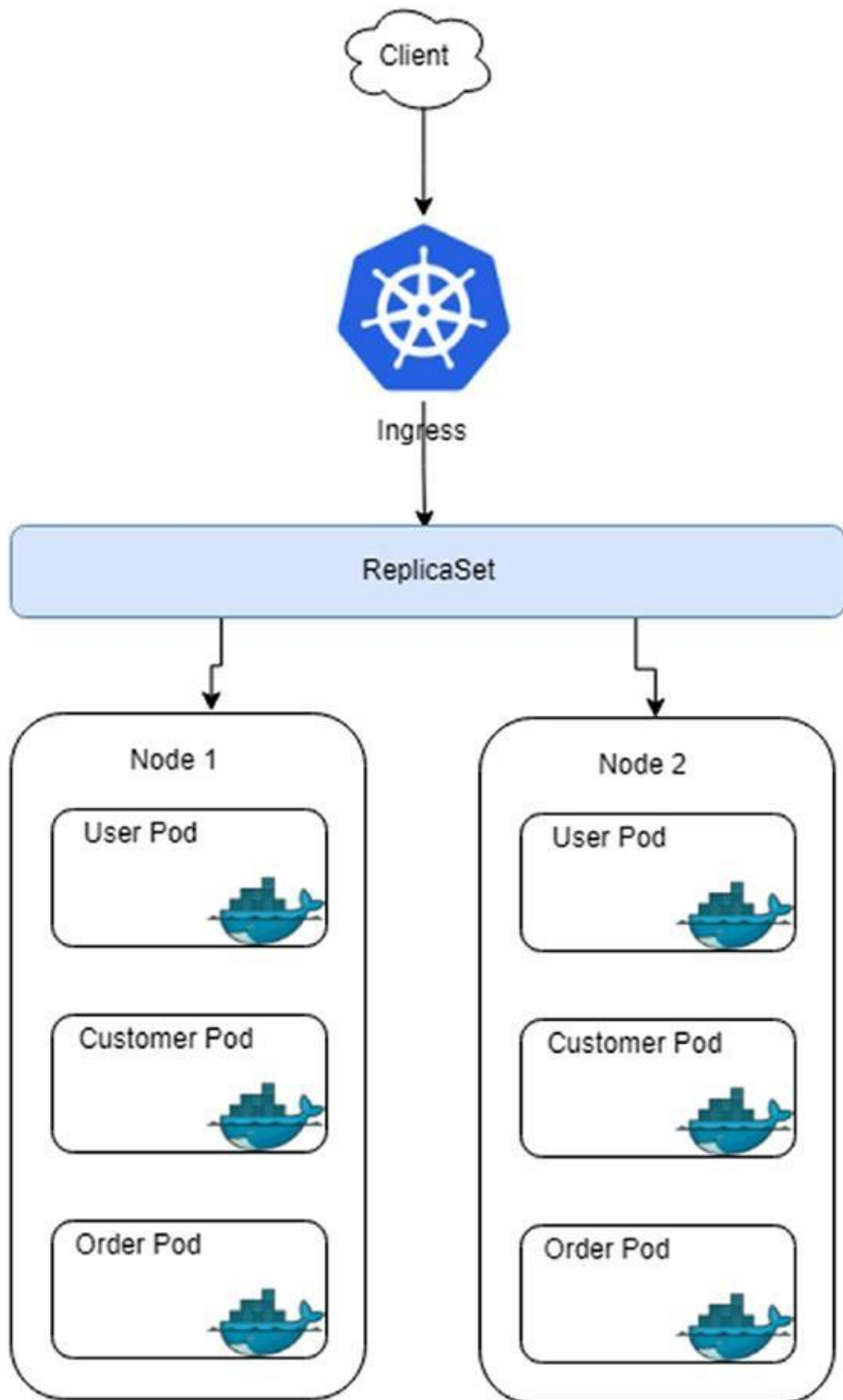


Figure 6-2.

Kubernetes microservices architecture

The preceding drawing shows a translation from a monolithic application to a Kubernetes/Docker microservices architecture. Every microservice is deployed in a pod. This is the smallest unit in our architecture. By managing more pods in a service, this creates our complete application.

In this architecture, we introduce a new component, called ingress. This component is used to manage the external access to the service, typically with HTTP. With ingress, we can create a load balancer, to manage access to our application.

Creating the Microservices Architecture in GCP

I have discussed the theory behind microservices architecture. It is only a very brief introduction. To impart a thorough understanding of microservices architecture would require a book dedicated to the subject. Given the scope of this book, my intention is to describe only how to use GCP to implement a microservices architecture and to use DevOps practices to manage it. So, let's get our hands dirty with GCP and implement a simple microservices architecture.

Note There is a big difference between DevOps and microservices architecture. The important distinction is that DevOps is a practice to help a company reduce time to market, the time to recover a system in the case of failure, and to improve the quality of what it releases. Microservices architecture is a software development practice derived from SOA architecture, and there are some points of commonality with DevOps. When we implement a microservices architecture, we put in place some practices common in DevOps, in particular, CD. The cultural change required for DevOps is essentially the same as that required to be successful when changing from a monolithic to a microservices architecture.

The first steps we must take are to design the microservices and define the Kubernetes component, service, and pod. Once the first steps are identified, and the components of our architecture are designed, we create a diagram with all the components we want to implement. In our case, we want to implement part of the architecture we designed earlier and the components

we need to implement it. For creating a microservices architecture, we must “dockerize” the application we want to transform in the microservice. In our case, we want to implement a basic front-end service for showing a list of users retrieved by a back-end system.

To implement that, first we must create the Go code and, after, the Dockerfile to dockerize the application. When the application is in Docker, we can start to integrate it into a CI pipeline and use Jenkins and Kubernetes to expose the application in the real world.

Creating the Services

The first step is to create the code for our Go service. So, let’s begin to write a simple code for creating the service necessary for exposing the users. In our case, we don’t use a real database, so we read a text file and offer this file to another service. The code looks like that in Listing 6-1 .

Listing 6-1. The Go Code for Reading a File
package main

```
import (  
    "bufio"  
    "fmt"  
    "log"  
    "os"  
)  
  
func main() {  
    file, err := os.Open("users.txt")  
    if err != nil {  
  
        log.Fatal(err)  
    }  
    defer file.Close()  
  
    scanner := bufio.NewScanner(file)  
    for scanner.Scan() {
```

```
fmt.Println(scanner.Text())
}

if err := scanner.Err(); err != nil {
log.Fatal(err)
}
}
```

The code is simple: read a text file and show the result to the shell. In our case, we create the.txt file directly in the folder of the Go file. The.txt file looks like this:

John Doe

Pierluigi Riti It is essentially just a list of names. The next step we must take to create our microservices architecture is to construct a Dockerfile (Listing 6-2).

Listing 6-2. The Dockerfile for the Application

```
FROM golang:latest
ADD . /app/
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

We can see that the Dockerfile is very simple. We start by importing the latest Golang from Docker, then we create the directory for the application with the following command:

```
RUN mkdir /app
```

This command executes the command mkdir on the Docker basic image and creates a new directory called app. When the directory is created, we must add the file we have in the path with the Dockerfile in the Docker directory. To do this, we use the following command:

```
ADD . /app/
```

This copies all the files in our Docker container. In our case, this is the file with the user and the Go file. The next line of our Dockerfile moves the work dir to the app dir. The command for that is

WORKDIR /app

Then we build the file for creating the Go executable file.

RUN go build -o main

This creates the Go executable file. We invoke this file every time we run the image, For that, we add the command

CMD ["/app/main"]

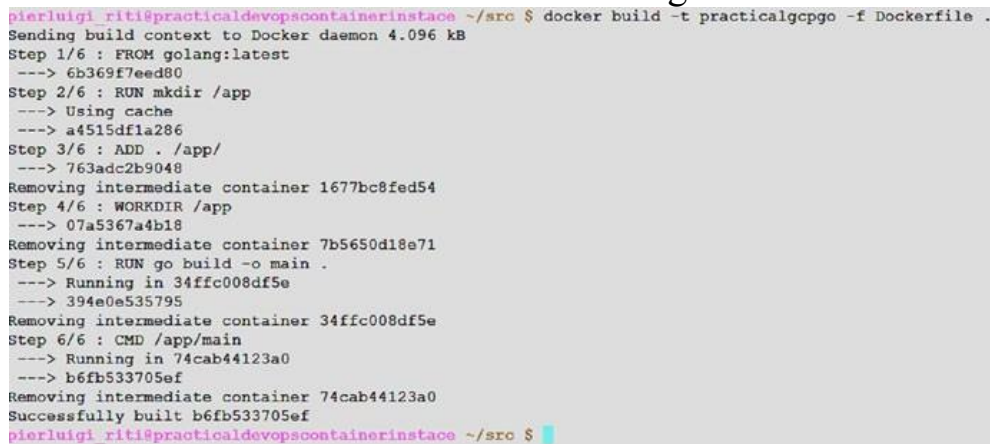
This executes the *main* file created previously with the go build command.

With the Dockerfile ready, we must compile the Docker image and execute.

The command to build the Docker image is

`docker build -t practicalgcpgo -f Dockerfile .`

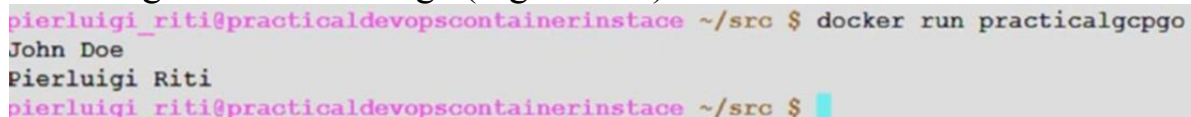
The result of the command is shown in Figure 6-3 .



```
pierluigi_riti@practicaldevopscontainerinstance ~/src $ docker build -t practicalgcpgo -f Dockerfile .
Sending build context to Docker daemon 4.096 kB
Step 1/6 : FROM golang:latest
----> 6b369f7eed80
Step 2/6 : RUN mkdir /app
----> Using cache
----> a4515df1a286
Step 3/6 : ADD . /app/
----> 763adc2b9048
Removing intermediate container 1677bc8fed54
Step 4/6 : WORKDIR /app
----> 07a5367a4b18
Removing intermediate container 7b5650d18e71
Step 5/6 : RUN go build -o main .
----> Running in 34ffc008df5e
----> 394e0e535795
Removing intermediate container 34ffc008df5e
Step 6/6 : CMD /app/main
----> Running in 74cab44123a0
----> b6fb533705ef
Removing intermediate container 74cab44123a0
Successfully built b6fb533705ef
pierluigi_riti@practicaldevopscontainerinstance ~/src $
```

Figure 6-3. The result of the Docker build

The image is now created. We can try out our image, and our code, just by executing the Docker image (Figure 6-4).



```
pierluigi_riti@practicaldevopscontainerinstance ~/src $ docker run practicalgcpgo
John Doe
Pierluigi Riti
pierluigi_riti@practicaldevopscontainerinstance ~/src $
```

Figure 6-4. The Docker image executed

Publishing the Image in the Registry

With the image now created, what we must do is to publish the image in our private repo. This is an essential step for our architecture, because with the image published, we can plan a CI/CD system.

Google has its own internal private repository. Before we can start to pull and push Docker images in the repository, we must be sure to have the credential correctly configured. To do that, open the Google SDK and execute the following command:

`gcloud auth configure-docker`

This command configures the credentials for access to the private Google Docker registry. The result of the command is shown in Figure 6-5 .

```
Welcome to Cloud Shell! Type "help" to get started.
pierluigi_riti@practicaldevopsgcpcli:~$ gcloud auth configure-docker
The following settings will be added to your Docker config file
located at [/home/pierluigi_riti/.docker/config.json]:
{
  "credHelpers": {
    "gcr.io": "gcloud",
    "us.gcr.io": "gcloud",
    "eu.gcr.io": "gcloud",
    "asia.gcr.io": "gcloud",
    "staging-k8s.gcr.io": "gcloud"
  }
}

Do you want to continue (Y/n)? y

Docker configuration file updated.
pierluigi_riti@practicaldevopsgcpcli:~$
```

Figure 6-5. The Google Cloud Docker Repository credentials configured

Tagging the Local Image

With the registry configured, we can start to push the image into the registry, but first, to push the image, we must create a tag with the registry name. After that, we can push the image.

Tagging the image is important for the management of the image itself. With the correct tagging, it is easy to understand where the image is located and what version of the code is in the image. Google recommends creating a tag following this naming convention:

[HOSTNAME]/[PROJECT-ID]/[IMAGE]

Using this naming convention, we can easily identify our image, based on the project ID and the location. The hostname is essentially where we store the image. There are four hostnames:

- gcr.io: This hostname stores the image in the United States, but this location can be changed in the future.
- us.gcr.io: This always stores the images in the United States but uses a different bucket instead of gcr.io.
- eu.gcr.io: This stores the images in Europe.
- asia.gcr.io: This stores the images in Asia.

It is important to choose the nearest location when we store images. This is because we don't want to have too much latency when we push or pull the image. We can tag the local image in the registry with the following command:

```
docker tag [SOURCE_IMAGE] [HOSTNAME]/[PROJECT-ID]/[IMAGE]
```

The command tags an image with the *latest* version. If we want to create a specific version, we can use this syntax:

```
docker tag [SOURCE_IMAGE] [HOSTNAME]/[PROJECT-ID]/[IMAGE]:  
[VERSION]
```

We can now create the command for tagging our first image.

The `SOURCE_IMAGE` is the image we just built with the Docker command.

In our case, the command is `docker tag practicalgcpgo`

`eu.gcr.io/practicaldevopsgcpcli/practicalgpc:1.0`

We can see the new image tagged, using the command `docker images`. The result shows us all the images present in our system (Figure 6-6).

```
pi@practicaldevopscontainerinstance:~/src $ docker tag practicalgcpgo eu.gcr.io/practicaldevopsgcp/practicalgpc:1.0
pi@practicaldevopscontainerinstance:~/src $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
practicalgcpgo	latest	b6fb533705ef	About an hour ago	796 MB
eu.gcr.io/practicaldevopsgcp/practicalgpc	1.0	b6fb533705ef	About an hour ago	796 MB

Figure 6-6. The Docker images result with the images tagged

From the result, we can see that we have two images with the same size and the same image ID. The first is the result of the Docker build. What we see is the latest version, the one after we found our new image tagged with the naming convention we indicated.

With the image tagged, we can now push the image on the registry. The syntax for pushing the image follows the same rules for creating the tag. The syntax looks like the following:

```
docker push [HOSTNAME]/[PROJECT-ID]/[IMAGE]:[VERSION]
```

If we omit VERSION, the image is pushed using the latest tag.

Note We must be sure to select the correct project ID. In our case, it is practicaldevopsgcpcli. If the wrong project ID is selected, Google can show an error telling us that the API is not enabled.

The result of the push is shown in Figure 6-7 .

```
pierluigi_riti@practicaldevopsgcpcli:~/src$ docker push eu.gcr.io/practicaldevopsgcpcli/practicalgpc:1.0
The push refers to a repository [eu.gcr.io/practicaldevopsgcpcli/practicalgpc]
ff7a0a6c9c6d: Pushed
7a516d560708: Pushed
5f6d9fba4d6c: Pushed
5ea538dd9f75: Layer already exists
6fa15c0fadae: Layer already exists
db38deaf0d0c: Layer already exists
fbbadfbfd3ca: Layer already exists
0f6f641d80ca: Layer already exists
76a66da94657: Layer already exists
0f3a12fef684: Layer already exists
1.0: digest: sha256:f6ada45e6b0a76fb5ac175234e1bda35bed1e04d0e8372b5ba695ae0395816cd size: 2419
pierluigi_riti@practicaldevopsgcpcli:~/src$
```

Figure 6-7. The result of pushing our image in the registry

It is possible to verify all the images we pushed in the registry with the following command:

```
gcloud container images list-tags [HOSTNAME]/[PROJECTID]/[IMAGE]
```

The result of the command shows the info about the image (Figure 6-8).

```
pierluigi_riti@practicaldevopsgcpcli:~/src$ gcloud container images list-tags eu.gcr.io/practicaldevopsgcpcli/practicalgpc
DIGEST      TAGS    TIMESTAMP
f6ada45e6b0a 1.0     2018-05-16T01:27:25
pierluigi_riti@practicaldevopsgcpcli:~/src$
```

Figure 6-8. The list of images pushed in the registry

Now that the image is pushed in the registry, this means we are the first service in the common registry. Of course, the service only shows the result for a text file. We need now to update the service for a reply with a JSON response and create a Kubernetes cluster for manage the service.

The next steps for implementing our microservices architecture are to create the cluster and put in place a system for CD and, based on that, update the cluster with the latest image.

Creating the Kubernetes Cluster

The Docker image is created, and the next step is to create a cluster in Kubernetes that we can use for scaling and maintaining the applications.

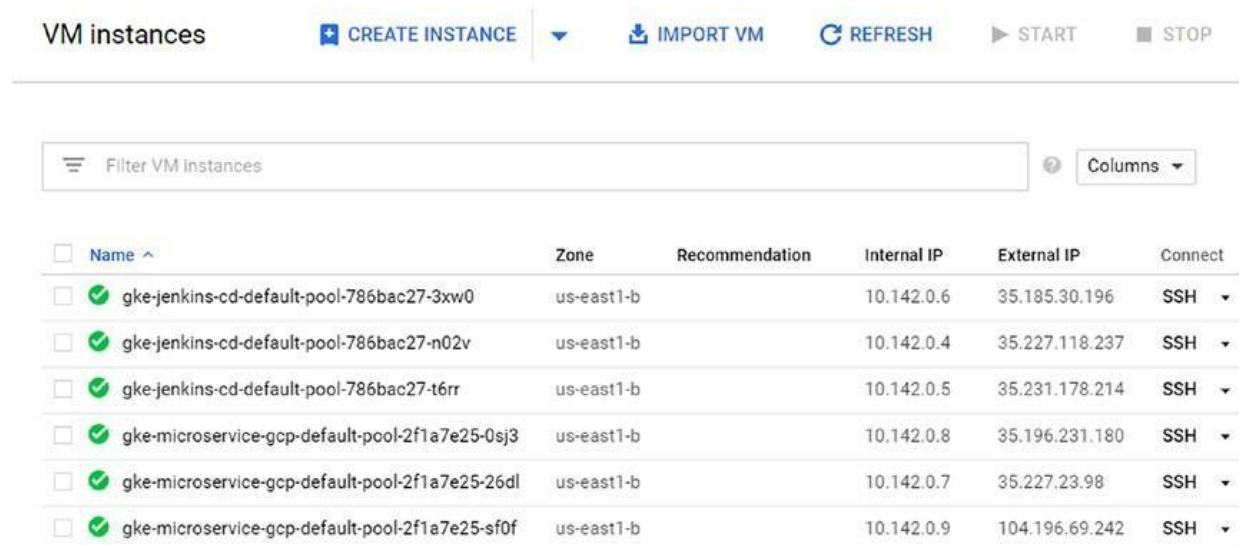
In the previous chapter, you saw how to create a Kubernetes cluster, but for reinforcement, we repeat some of these steps and consider why we do that. First, we must create the basic cluster. For our basic application, we can create a three-node cluster. The steps to create a Kubernetes cluster are these:

```
gcloud config set compute/zone us-east1-b
```

Set the compute zone to your zone (in my case,us-east1-b), then execute the command to create the cluster.

```
gcloud container clusters create microservice-gcp --num-nodes 3 \
--scopes https://www.googleapis.com/auth/projecthosting,storage-rw
```

This command creates a three-node cluster called *microservice-gcp* . This is a specific cluster in which we go to deploy our previous applications. Because we had to create the Jenkins cluster for the CI/CD, we now have two Kubernetes clusters in our system (Figure 6-9).



The screenshot shows the 'VM instances' page in the Google Cloud console. At the top, there are buttons for 'CREATE INSTANCE', 'IMPORT VM', 'REFRESH', 'START', and 'STOP'. Below these is a search bar labeled 'Filter VM instances' and a 'Columns' dropdown menu. The main table lists six VM instances, all in the 'us-east1-b' zone. Each instance has a checkbox, a green checkmark icon, a name, a zone, a recommendation, an internal IP, an external IP, and a 'Connect' button with an SSH icon. The instances are grouped into two clusters: 'gke-jenkins-cd' (three instances) and 'gke-microservice-gcp' (three instances).

<input type="checkbox"/>	Name ^	Zone	Recommendation	Internal IP	External IP	Connect
<input type="checkbox"/>	gke-jenkins-cd-default-pool-786bac27-3xw0	us-east1-b		10.142.0.6	35.185.30.196	SSH ▾
<input type="checkbox"/>	gke-jenkins-cd-default-pool-786bac27-n02v	us-east1-b		10.142.0.4	35.227.118.237	SSH ▾
<input type="checkbox"/>	gke-jenkins-cd-default-pool-786bac27-t6rr	us-east1-b		10.142.0.5	35.231.178.214	SSH ▾
<input type="checkbox"/>	gke-microservice-gcp-default-pool-2f1a7e25-0sj3	us-east1-b		10.142.0.8	35.196.231.180	SSH ▾
<input type="checkbox"/>	gke-microservice-gcp-default-pool-2f1a7e25-26dl	us-east1-b		10.142.0.7	35.227.23.98	SSH ▾
<input type="checkbox"/>	gke-microservice-gcp-default-pool-2f1a7e25-sf0f	us-east1-b		10.142.0.9	104.196.69.242	SSH ▾

Figure 6-9. The Kubernetes clusters present in our cloud

To better manage the cluster, we create a new namespace, called *microservice-gcp*. The command for creating the namespace is `kubectl create ns microservice-gcp`

With the namespace created, we can now start to create the YAML file to define the Kubernetes service we want to implement.

We must create the deployment file to download the image we created previously. The file is similar to that in Listing 6-3 .

Listing 6-3. The Service for Our Microservice

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:

name: practical-microservice
spec:
replicas: 1
template:
metadata:
name: backend
labels:
app: gcp-microservice
role: backend
env: production
spec:
containers:
- name: practicalgcp-microservice
image: eu.gcr.io/practicaldevopsgcpcli/practicalgpc:1.0
resources:
limits:
memory: "500Mi"
cpu: "100m"
imagePullPolicy: Always
command: ["sh", "main"]
```

Next, we must create the new deployment in the namespace we previously created, using the following command:

```
kubectl --namespace=microservice-gcp apply service.yaml
```

The service is now created and ready to be used. We can now create a system for the CD and continue the development of the microservices architecture.

Note There is a technical limitation with the free tier of GCP: only one public IP is allowed. This means that we can encounter an issue when we try to display the service for the microservices architecture. In case this happens, we can integrate the service with the code we deployed in Chapter 5 and use this service instead of the simple-app.

Conclusion

In this chapter, you saw how to create a microservices architecture. By the term *microservices*, we mean a way to implement and organize our architecture.

Microservices have a natural connection with Docker. When we create a container for our microservice, we can easily create a microservice for our architecture and, more important, we can integrate it into our CD system. With GCP, it is easy to create and maintain a microservice, because Kubernetes is naturally integrated into Google, allowing the cluster to be maintained in an easy and very flexible way.

CHAPTER 7

Monitoring in GCP

Monitoring is one of the most critical stages in our DevOps journey. Having a good monitoring system in place is crucial for obtaining accurate feedback about the infrastructure and the software we have created and, of course, for overseeing access to the system. In this chapter, I discuss what is fundamental to building an effective monitoring system in Google Cloud Platform (GCP). If someone asks, “Why must I use a monitoring system?” You can reply with this analogy: “Having a large distributed system with no monitoring is like riding a bike, only that you are in Hell, and you and the bike are onfire.”

What Is a Monitoring System?

Arriving at a correct definition of monitoring is not simple. For example, we must define first if we're talking about *software monitoring* or *system monitoring*. Software monitoring can be defined as monitoring software to ensure the quality of the service, based on the terms of the service-level agreement (SLA). This is usually composed of a set of metrics designed on the software and graphed by specific software. These metrics are collected and analyzed to provide a picture of the software and how it works.

System monitoring can be defined as monitoring a system to ensure the correct state of the machinery. This consists of a set of monitors and alerts to check the general status of the server and the network, to confirm correct functionality of the current infrastructure.

Based on both definitions, we can build a common definition. Monitoring is a set of practices and software necessary to assure the status of the system. It is made up of a combination of software monitoring and system monitoring. When we want to create a monitoring system, we must be sure to put in place all the metrics and the alerts required to ensure continuous feedback about the status of the system.

In keeping with this definition, we can identify the key requirements of our monitoring system:

- *Metrics* : We must define a set of metrics to collect data concerning the current status of the system.
- *Alerts* : We must define a system to provide alerts when the metrics indicate an incorrect value.
- *Monitoring* : We must use software to collect and display the actual state of the system. Normally, this is done by specific software.
- *Feedback* : We must have proactive feedback, in case of errors in the system. This can be provided via e-mail with an escalation system.

These four areas are essentially the heart of every monitoring system. Real-time monitoring is crucial for debugging a problem when it occurs in production.

Effective real-time monitoring indicates the actual state of the system when the problem occurs and provides a picture of all the states of the system, not only the software, but, for example, the network, database, disk space, etc. This information can be used to solve the actual problem and, more

important, to prevent the occurrence of the same problem again. This is because it shows where the system failed and helps us to identify the true root cause related to that.

When we have good real-time monitoring, we can easily identify errors when, for example, we release a new feature in our software. We can see the error ratio for the release, and in the event that we observe too many failures, we can easily push a rollback of the code, for example, by using the system for continuous delivery (CD), and release a previous version of the software. Another use for a monitoring system is predictive analysis. When we have a monitoring system in place, we can collect data about the status of the system. Based on this data, collected with a log such as Splunk, for example, we can start to make some predictive analysis. Imagine that we have a graph displaying the usage of the CPU (see Figure 7-1).

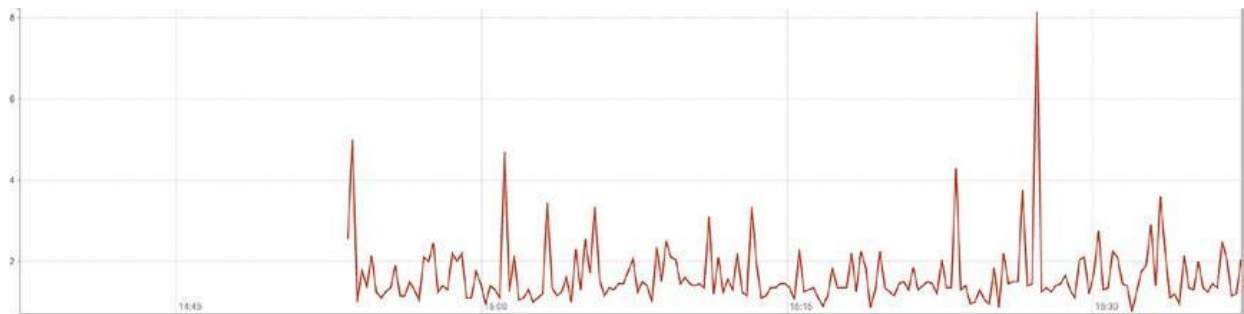


Figure 7-1. *A sample memory graph*

The preceding graph shows the usage of the CPU at different times. We can see some spikes and some normal usage. The same graph can be created for different time frames, for example, days, weeks, and months. The graph can display the usage across different data frames. This information can be used to perform calculations and determine the busiest periods during which our infrastructure is in use.

A good monitoring system helps us to identify this period and then adapts the software and the infrastructure to respond better to the busiest periods. In addition, it provides information about the historical performance of the system and can help us to calibrate and identify what we need when we must spin up a new server in our infrastructure. This translates into cost savings and better identification of any bottlenecks in the system.

When we think about monitoring, we can identify two types: white-box monitoring and black-box monitoring. White-box monitoring gives us details about the internal state of the application, for example, the HTTP connection, number of users, etc. Blackbox monitoring, on the other hand, checks how well the system is operating, but from the outside. Black box doesn't give you the information about the number of HTTP connections or users actually connected to the system.

A good example of white-box monitoring is Prometheus, and a good example of black-box monitoring is Nagios. We can use both types, to build a good and strong monitoring system.

Factors Involved in the Monitoring System

When we think of building our monitoring system, we must consider some key factors. The first assumption we must make is that there is no clear definition of monitoring. However, we can identify some common terms related to monitoring systems.

- *Monitoring* : I have defined monitoring previously, but, to summarize,

it can be defined as collecting, processing, aggregating, and displaying real-time information about a system, errors, lifetime of the server, user connection, space used, etc.

- *White-box monitoring* : This type of monitoring is of the internal metrics exposed by the system to the outside. An example of whitebox monitoring is logging a service to check if the software is live, such as an HTTP check, and any other service that can be used to view inside the system.

- *Black-box monitoring* : This is a kind of monitoring performed from outside the system, for example, to monitor an application. As opposed to white-box monitoring, it is used to check the log produced from the system itself. With black-box monitoring, we don't really check the log but review the output of the service, for example, checking whether the service is live.

- *Dashboard* : The dashboard is essentially a graphical user interface (GUI) used to indicate the status of the system. This is usually a web application. A

dashboard normally allows the user to filter, or search, for a specific resource on the system.

- *Alert* : An alert is the reaction of the monitoring system to a specific condition, for example, when we have a high CPU usage or a disk space problem. An alert can be raised in different ways. For example, we can have a *ticketing alert* , meaning that when a system identifies a problem, a ticket is created and directly assigned to a team, or a *mailing alert*, in which case the system sends an e-mail to a specific mailing list when it detects an issue.
- *Root cause* : The root cause is essentially the principal source of an issue. Identifying the root cause of a problem is integral to the stability of the system, and a good monitoring system can easily identify the cause and put in place a new alert to prevent the problem from occurring again.
- *Node/machine* : This term identifies the hardware, physical or virtual, in which the error occurs. Every node or machine can run more than one service, and an alert can be connected to every server.
- *Push* : This is a release of the software, made by an automatic system, such as Puppet or Chef.
- *Rollback* : This is the procedure followed to release an older version of software to make a system stable again. This is normally done by pushing an older version of the software in the system.

This terminology is common to all the monitoring systems and is often used to describe the operations we undertake to solve a problem on the system.

Why Monitoring Is Important

Monitoring is important for many reasons, not only to ensure the stability of the system itself. For example, a monitoring system can be used for the following:

- *Temporal analysis* : The monitoring system can be used to observe the critical aspects of a system's infrastructure. We can, for example, collect data about the growth of our database or we can determine the time of the year or

month it is most active. As well, we can gauge the quality of our software across different releases.

- *Comparing data* : With a monitoring system, we can collect data from one system and compare it with another. This data can then be used for making a decision about what software is better for our needs.
- *Alerts* : As mention previously, a monitoring system is used to raise alerts when errors in the system occur. The monitoring system checks the status of the system and, in the case of an error, raises an alert.
- *Visualizing the status of the system* : When we have a monitoring system in place, we usually have a dashboard for visualizing the present status of the system. A monitoring system offers a clear picture of the entire system.
- *Debugging* : A monitoring system is based on log analysis, which means that we can use the system to analyze the status of the software at a specific time and see if there is any particular condition responsible for a specific error.

The preceding are a few of the most common reasons a good monitoring system can be used to ensure security. We can monitor access to the system and use this information to prevent a data breach. A good monitoring system must respond to two simple questions:

- What is broken?
- Why is it broken?

Every monitoring system must answer both these questions, to be truly effective.

What Is Broken and Why?

A good and effective monitoring system must be able to give us an answer to these simple questions. First, we must determine *what* is broken. This is the identification of the component of the system that doesn't work properly. This is the basis for monitoring the system.

The second answer the monitoring system must find is to the question, Why is it broken? The answer to this is more complex to propose, because it involves analysis of the error. A good monitoring system can help us to do that, because it gives us an actual picture of the status of the system.

Table 7-1 shows two common errors that can occur in every system. We can see that the “what” is the *symptom* and the result of *cause* of the error. A good and effective monitoring system must be able to identify the *what* and send an alert to the engineering team to begin the analysis and identify the *why* of the failure.

Table 7-1. Examples of What May Be Broken and Why

The web site shows a 505.

The FPT fails.

Why

The web service we use for getting the data is down, and we can't reach it.

The disk is full, and the file can't be uploaded.

White-Box and Black-Box Monitoring

As I mentioned previously, there are two kinds of system monitoring: white-box and the black-box monitoring. White-box monitoring can be performed on all the software and systems and can be set to reveal the internal state of the software. With white-box monitoring, we use software to check the system at an operating-system level. Whitebox monitoring is not directly related to the infrastructure. The software we can use for white-box monitoring is, for example, Prometheus, Nagios, Zabbix, and Splunk. All these softwares are used to check the status of the infrastructure and reveal it directly.

Conversely, black-box monitoring is used to check the system by using metrics. To look at the system as a black box means that we can peer directly inside software, but based on a statistic defined by some related metric. Software for black-box monitoring includes Prometheus, Nagios, and Splunk.

We can think of black-box monitoring as telling us what is broken, but not in a predictive way. Black-box monitoring indicates when a problem is present but doesn't give us a way to predict when a problem can appear.

White-box monitoring is based on inspection of the actual state of the system. This means that we can use the data collected. For example, we can use Splunk to analyze the log and see when a condition exists that results in a potential issue.

White-box monitoring directly checks the operating system and identifies any condition or metric that can be used for debugging or for identifying the cause of an issue, allowing users to prevent it from occurring. With white box, for example, we can see when a system starts to use too much memory, and this can slow the system.

There is a golden rule we can apply to monitoring systems. It defines four golden signals that we can use to gain a picture of the system. These signals are:

- Latency
- Traffic
- Error
- Saturation

Each signal is used to describe a condition that is not necessarily associated with an error but can be the cause of subsequent errors.

Latency

Latency is the time we require to service a request. High latency is not always a problem, but it can signal a slow response by a system, and, in some cases, this can lead to an error. It is important to prevent the error, by monitoring the normal time response the system has for identifying when there is a latency problem and acting faster to fix the issue.

Traffic

Traffic is not an actual error, but it can be used to understand how to scale our system to avoid problems or failures. Traffic measures how much demand there is for the system. We can put in place different high-level measurements for defining this, such as how many pages are requested per second or how many requests our server receives per second. This measure can be adapted, depending on the nature of the system we must monitor.

Error

When we measure an error, we must not only determine the evident error, such as a 500 internal server error or a `NullPointerException`, but also whether the function finishes correcting any work, such as a page returning a 200 error but having different content, such as different data. It is important to track both situations. Normally, a monitoring system tracks only the evident error condition, but a very good monitoring system also can track partial error conditions, such as a 200 error, and any resulting incorrect data.

Saturation

Saturation is used to measure how “full” our system is. We usually use different indicators to determine saturation, such as the memory, the file system, the network, and the I/O pipeline. Saturation can be used for predictive analysis, because, for example, we can use this measure to understand how many times we must saturate the database or the filesystem. In a complex system, identifying the value is not so easy, because a 10% improvement in the resource can have the result of slowing down the entire system. Having an effective means of measuring saturation is very important for preventing critical errors in our system.

Monitoring all these four signals is a good start to establishing a very good and effective monitoring system.

Building a Monitoring System

Until now, we have analyzed what a monitoring system must include and potentially identify in our system. What we must do now is define the requirements and the software required to build our monitoring system.

Imagine that we want to build our monitoring system from scratch. We must consider the main components of our system, such as the following:

- *OS monitoring* : We must define a set of scripts to interact with a very good and effective monitor. For every measure, we can raise an alert, connected to an e-mail, that can act expeditiously and prevent similar problems from happening. The operating system can offer feedback about the status of the system.
- *Log analysis* : We must have the capacity to read the log, to identify errors in the software and collect the data.
- *Alerts* : We must have the capacity to send out an e-mail or connect to some external software, to track any error occurring in the system.
- *Dashboard* : For a visual representation of the system, and to obtain observable analysis, a dashboard can be designed for use with the software, and we can use the dashboard to gain a picture of the actual system.
- *Software integration* : We must design some third-party integration to use and connect to our software, with some external alerting or data analysis system.
- *Monitoring the four signals* : To have a very good and effective monitoring system, we must monitor all four signals—latency, traffic, error, and saturation. When we monitor these signals and raise the proper alert—if one of the signals shows an anomaly—we have the basis of a very effective monitoring system.

The preceding bullet points are essentially the main requirements of our system. We can build the system from scratch or use some existing software to perform the monitoring. GCP has an integrated monitoring system called *Stackdriver*. So, let's start to use Stackdriver, to monitor our cloud.

Configuring Stackdriver on GCP

Google Cloud offers its own solution for creating a monitoring system. It is called Stackdriver and offers the power of monitoring all cloud applications

in a single place. It is possible to create graphs and add the signals we want, based on the service we have.

Stackdriver offers various integration with some common software used for our application, such as NGINX, Cassandra, Apache, and Elasticsearch. The alert generated by Stackdriver can be sent directly to alerting software, such as PagerDuty, or chat, such as Slack or HipChat.

To use Stackdriver in our GCP, the first step is to create an account. To do that, simply connect to the GCP console, open the left menu, and in the Stackdriver section, select Monitoring (see Figure 7-2).

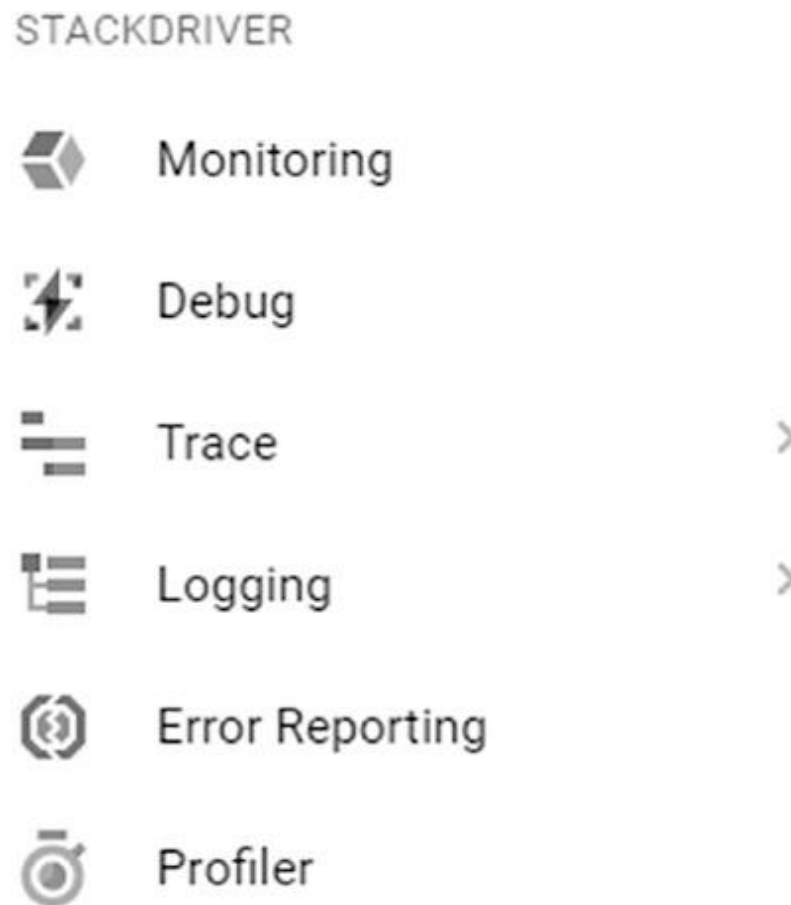


Figure 7-2. *The*

Stackdriver menu from GCP

When the monitoring board is open, it asks to select the project we want to monitor and, if it is our first time, to remind us to start our 30-day free trial (Figure 7-3).

Figure 7-3. *The Stackdriver monitoring section*

To start Stackdriver, select the project PracticalDevOpsGCP and then click the CreateAccount button. When the account is created, in our case, it is associated with our GCP.

The next step is to associate the project we want to monitor under Stackdriver. In our case, we can see the two projects presented in our cloud. The first selected is shown in Figure 7-4 .



Figure 7-4. Stackdriver section to add a project to monitor

Scroll down the page and click Continue. This will open the next page for configuring Stackdriver. The next page asks to configure the AWS account. We can skip this part, as it is not necessary for our purposes.

The last section is used to configure the Stackdriver agent. This page shows the command we must execute to configure the client. The agent runs in our cloud instance and collects all the information we require for putting our monitoring system in place (Figure 7-5).

Figure 7-5. The command for configuring the Stackdriver client in our platform

To configure the client, open the command line and execute the command we require for configuring the client. The first command we must execute is curl, to download the script necessary to install the client.

```
curl -sSO https://dl.google.com/cloudagents/install-monitoring-agent.sh
```

The file is very small, and the command returns immediately. The next command we execute is the command to install the client.

```
sudo bash install-monitoring-agent.sh
```

The script returns the output for the installation, but the part we must pay

attention to is the last line of the script, where we find the status of the installation.

Created new plugin context.

option = PIDFile; value = /var/run/stackdriver-agent.pid;

option = Interval; value = 60.000000;

option = Hostname; value = ;option = FQDNLookup; value = false; Created new plugin context..

```
=====
===== Installation of stackdriver-agent-5.5.2-382
completed successfully. Please consult the documentation for
troubleshooting advice:
```

<https://cloud.google.com/monitoring/agent>

You can monitor the monitoring agent's logfile at:

/var/log/syslog

```
=====
=====
```

Now that the installation has been completed successfully, we have correctly installed the Stackdriver monitoring client. To complete the installation of Stackdriver, we must install the logging client. This client is used to stream logs from the VM and other third-party software. This can be used to identify issues related to reading the log of the application. To install the logging client, we first must download the script necessary to install it.

`curl -sSO https://dl.google.com/cloudagents/install-logging-agent.sh`

When the script is downloaded, we can now execute the script to install the client. `sudo bash install-logging-agent.sh`

As with other scripts, this one returns the status of the script, but for us, what is important is the last part of the script, where we can see the status of the installation (Listing 7-1).

Listing 7-1. The Result of Logging the Stackdriver Client Installation

Installing default conffile /etc/google-fluentd/google-fluentd.conf ... invoke-rc.d: could not determine current runlevel

invoke-rc.d: policy-rc.d denied execution of start.
Setting up google-fluentd-catch-all-config (0.7) ...
Restarting google-fluentd: google-fluentd.

===== Installation of google-fluentd complete.
Logs from this machine should be visible in the log viewer at:

https://console.cloud.google.com/logs/viewer?project=&resource=gce_instance/instance_id/6349597365373579285

A test message has been sent to syslog to help verify proper operation.
Please consult the documentation for troubleshooting advice:

<https://cloud.google.com/logging/docs/agent>

You can monitor the logging agent's logfile at:
[/var/log/google-fluentd/google-fluentd.log](#)

Now that we have correctly installed our Stackdriver logging client, we now have all the clients we need to start to put in place our monitoring system.

Creating the Application

Now that we have configured and installed all the necessary elements for Stackdriver, we can start to see how Stackdriver works. For demonstration purposes only, we create a new PHP 7 with Apache CloudStack in GCP and see how we can monitor the system.

The first step we take is to create a new Compute Engine Instance. To do that, we must open the console and create the Compute Engine. We can create an instance by clicking the Create button from the Create New Instance pop-up. This shows the details we need to insert for our new instance (Figure 7-6).

Create an instance

Name [?](#)
stackdriverinstance

Region [?](#) Zone [?](#)
us-east1 (South Carolina) us-east1-b

Machine type
Customise to select cores, memory and GPUs.
small (1 shared... 1.7 GB memory Customise
[Upgrade your account](#) to create instances with up to 96 cores

\$14.20 per month estimated
Effective hourly rate \$0.019 (730 hours per month)
[Details](#)

Container [?](#)
☐ Deploy a container image to this VM instance. [Learn more](#)

Boot disk [?](#)
New 10 GB standard persistent disk
Image
Debian GNU/Linux 9 (stretch) [Change](#)

Identity and API access [?](#)
Service account [?](#)
Compute Engine default service account

Access scopes [?](#)

Figure 7-6. The page for creating the new instance we want to monitor

Because the instance is used primarily to show how monitoring works with GCP, we call the instance *stackdriverinstance*. Select a small instance and allow both HTTP and HTTPS traffic, leaving the rest of the files to the default details, and click Create, to create the instance (Figure 7-7).

Filter VM instances [?](#) [Columns](#)

<input type="checkbox"/>	Name ^	Zone	Recommendation	Internal IP	External IP	Connect
<input checked="" type="checkbox"/>	stackdriverinstance	us-east1-b		10.142.0.2 (nic0)	35.231.28.26 ↗	SSH ⋮

Figure 7-7. The Stackdriver instance created on GCP

With the instance created, we now must install the software we want to monitor. To do this, we open an SSH shell, to connect to the instance. We can do this by simply clicking the SSH label. I suggest opening a new browser, as this opens a shell directly on our VM instance (Listing 7-2).

Listing 7-2. The SSH Connection with the New Instance Created

```
Connected, host fingerprint: ssh-rsa 2048
3A:76:C8:B9:E5:0C:9B:30:BF:47:B5:
92:9C:23:CA:88:BA:09:4E:76:48:39:C5:32:D0:C9:0B:37:94:6D:C4:15
Linux stackdriverinstance 4.9.0-7-amd64 #1 SMP Debian 4.9.110-3+deb9u2
(2018-08-13) x86_64
The programs included with the Debian GNU/Linux system are free
software; the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the
extent permitted by applicable law.
pierluigi_riti@stackdriverinstance:~$
```

With access to the instance, what we must do now is install the software. We want to install PHP 7 and Apache HTTP web server. So, the first command we need to execute is one to update the library. We choose a Debian operating system, so the command for updating the library is

```
sudo apt-get update
```

This command generates the output with the list of the package updated from the system. The command done, we must now install PHP 7 and Apache Web Server. For that, we use the following command:

```
sudo apt-get install apache2 php7.0
```

The software starts to check the packages necessary for the installation and asks us to confirm the installation of the software. Click Y, and the software continues the installation (Listing 7-3).

Listing 7-3. The PHP-Apache Installation Package

```
pierluigi_riti@stackdriverinstance:~$ sudo apt-get install apache2 php7.0
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
```


apache2-bin apache2-data apache2-utils libapache2-mod-php7.0 libapr1
libaprutil1 libaprutil1-dbd-sqlite3
libaprutil1-ldap libicu57 liblua5.2-0 libperl5.24 libxml2 perl perlmodules-
5.24 php-common php7.0-cli
php7.0-common php7.0-json php7.0-opcache php7.0-readline psmisc
rename sgml-base ssl-cert xml-core

Suggested packages:

www-browser apache2-doc apache2-suexec-pristine | apache2-suexec-
custom php-pear perl-doc
libterm-readline-gnu-perl | libterm-readline-perl-perl make sgml-base-doc
openssl-blacklist debhelper

The following NEW packages will be installed:

apache2 apache2-bin apache2-data apache2-utils libapache2-mod-php7.0
libapr1 libaprutil1
libaprutil1-dbd-sqlite3 libaprutil1-ldap libicu57 liblua5.2-0 libperl5.24
libxml2 perl perl-modules-5.24
php-common php7.0 php7.0-cli php7.0-common php7.0-json php7.0-
opcache php7.0-readline psmisc rename sgml-base
ssl-cert xml-core

0 upgraded, 27 newly installed, 0 to remove and 0 not upgraded. Need to get
21.0 MB of archives.

After this operation, 95.1 MB of additional disk space will be used. Do you
want to continue? [Y/n]

With the software installed, we can now check if the server is responsive. To
connect to the server, we can use the link http://<external_ip>. In my case,
the link is <http://35.227.27.47/>. This shows the default page of the Apache
HTTP server (Figure 7-8).

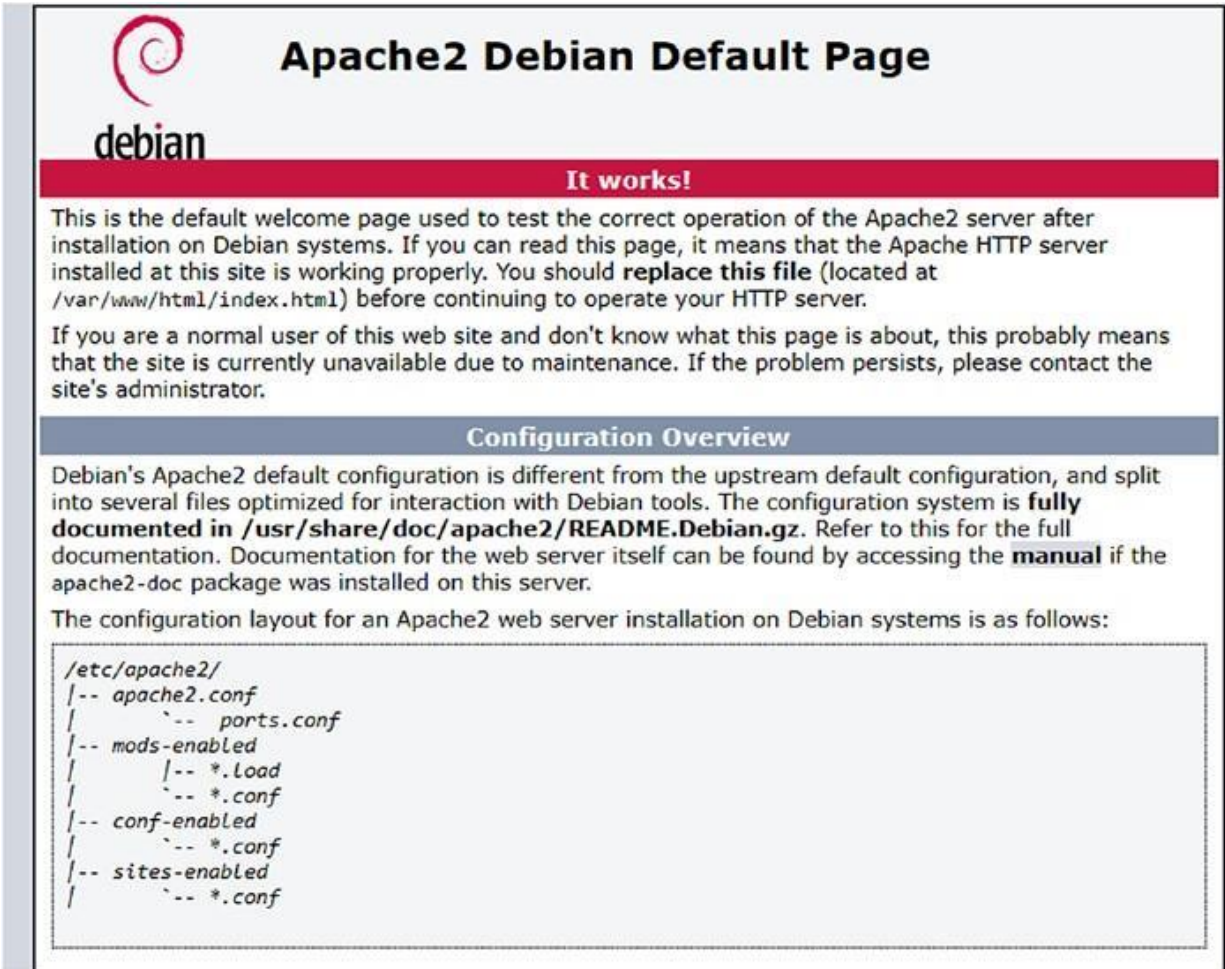


Figure 7-8. The Apache HTTP server default page

Log Analysis with Stackdriver

With Stackdriver and the application we want to monitor configured, we can now start to see how Stackdriver works. We had previously installed the logging client for Stackdriver. This means we can see the log for the application directly from the Compute Engine page. To access the log, click the three dots near the name of the instance and select View logs (Figure 7-9).



Figure 7-9. View logs command for a new instance

View logs opens the log dashboard for Stackdriver (Figure 7-10).

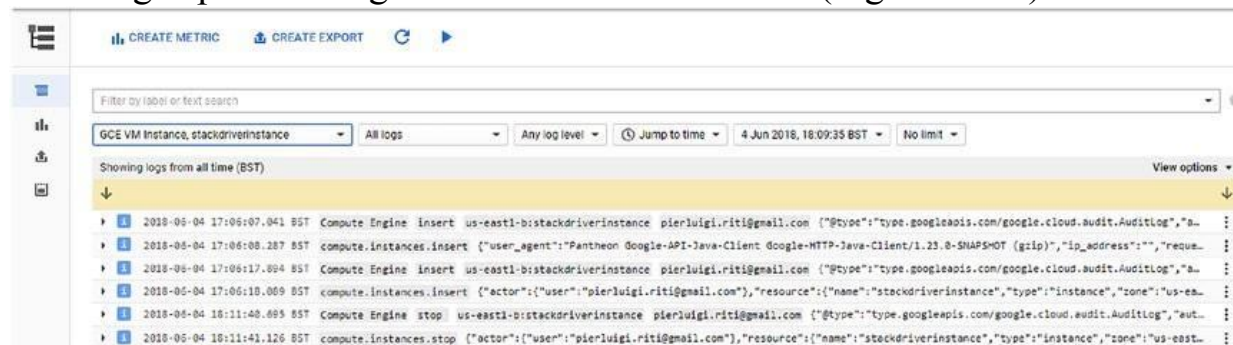


Figure 7-10. The log dashboard for Stackdriver

Stackdriver is a very mature system for monitoring, and the logging dashboard gives us numerous instruments for making our analysis. To help to analyze the log, we can apply some filters to the log we choose to see.

We can change the log we want to analyze, using the first drop-down menu on the left. We can choose what kind of log we want to analyze. For example, we can read the log based on these families of logs: Firewall Rule, Project, VM Instance, and Bucket. These are just examples of the different families of logs. For every section, we can have sub-detailed logs. For example, in case of VM Instance, we have a choice of all the instances we have in our system (Figure 7-11).

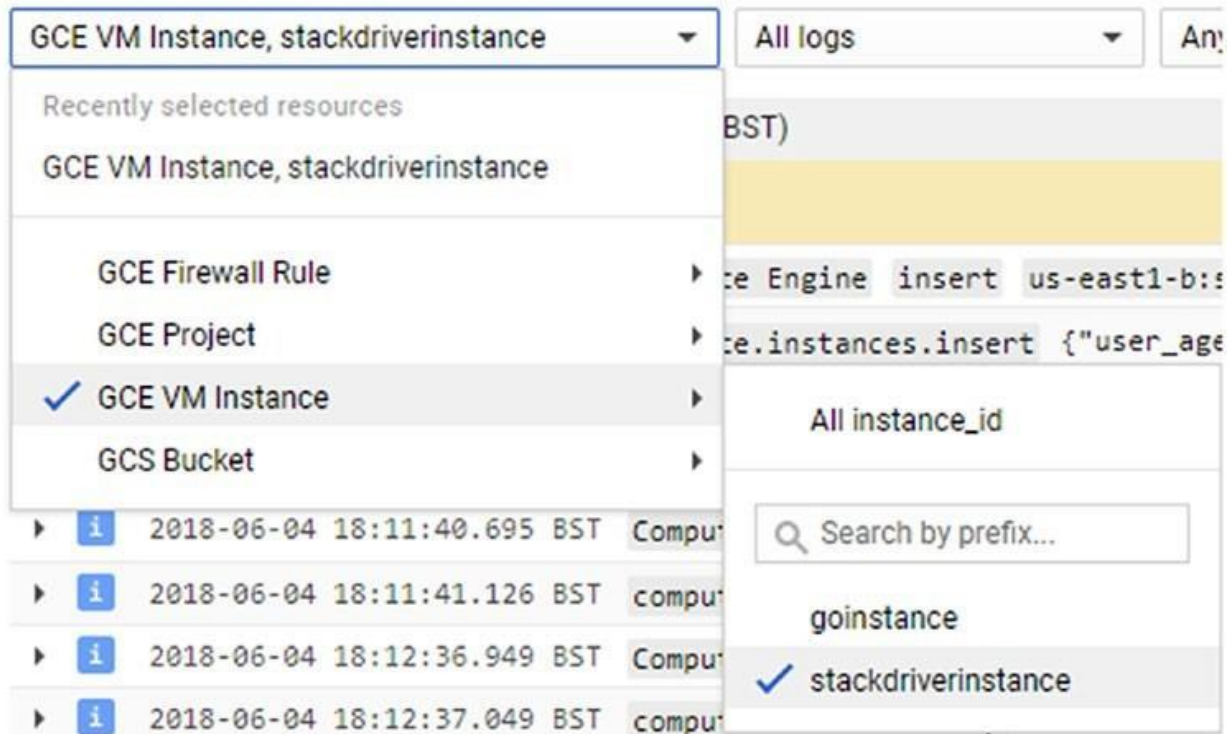


Figure 7-11. The VM Instance submenu

We can change the type of log we want to analyze, and based on our selection, we can have more submenus. This helps us to undertake a very detailed analysis of the system.

We can add another filter. For example, we can choose what log we want to read, by selecting among the choices on the menu. By default, we see “All logs,” but if we change the log that we see, we can add another filter. For example, we can choose to see only the error or the debug information. We also can add a temporal filter. We can choose to see only the last hour’s log. We can add a text filter, which makes a free text search in the log.

We can use the filter to analyze the log and debug our application. We can use the filter, to move back on the time, and the log dashboard, to analyze and determine what the root cause of an error is.

Alerts in Stackdriver

A good monitoring system can’t be put in place without a good alert system. To create an alert in Stackdriver, open the Monitor page in the Stackdriver section. This shows the dashboard for creating the alert and managing the alert policy in Stackdriver (Figure 7-12).

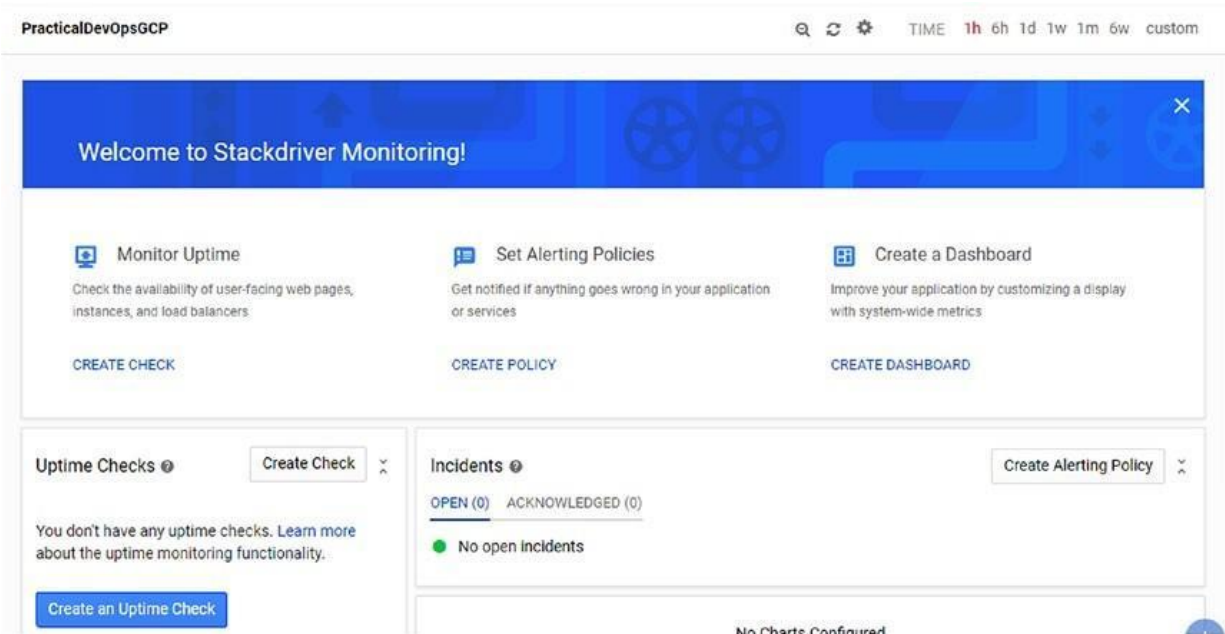


Figure 7-12. *The Stackdriver monitoring dashboard*

We can see that we have created some basic alerts connected to the application. The basic alert we want to create is the uptime, to know if the application is live or not.

To create an uptime alert, we must first create a monitor. To do this, we select from the dashboard **CREATE CHECK**, under **Monitor Uptime**. This opens the configuration page for the alert. We can configure different parameters on the check. We can set the Title of the alert, in our case, *practicaldevopsgcp_uptime*, and the Type of check we want. We can select three different types:

- HTTP
- HTTPS
- TCP

In our case, we use the HTTP type. This is because the instance we want to check responds to an HTTP link. To be effective, an alert must know what type of resource to check. This can be configured in the Resource Type section. We can select one of the following types:

- Instance
- Cloud Load Balancer

- App Engine
- URL

Because we want to monitor an instance, we choose Instance. We can now select what instance is applied to the alert and whether only for a single or a group. We can configure that on the Applies To section. To be effective, an alert must check the system after a certain amount of time. We can configure that in the Check Every section. By default, the check starts every five minutes. We can change this interval simply by selecting a different time from the drop-down menu. Now that we have set all the values for the alert, we can save the alert and start to monitor our system. We can test the alert, to see if it is working properly. To test it, we just have to click the Test button. This shows the results of the operation (Figure 7-13).

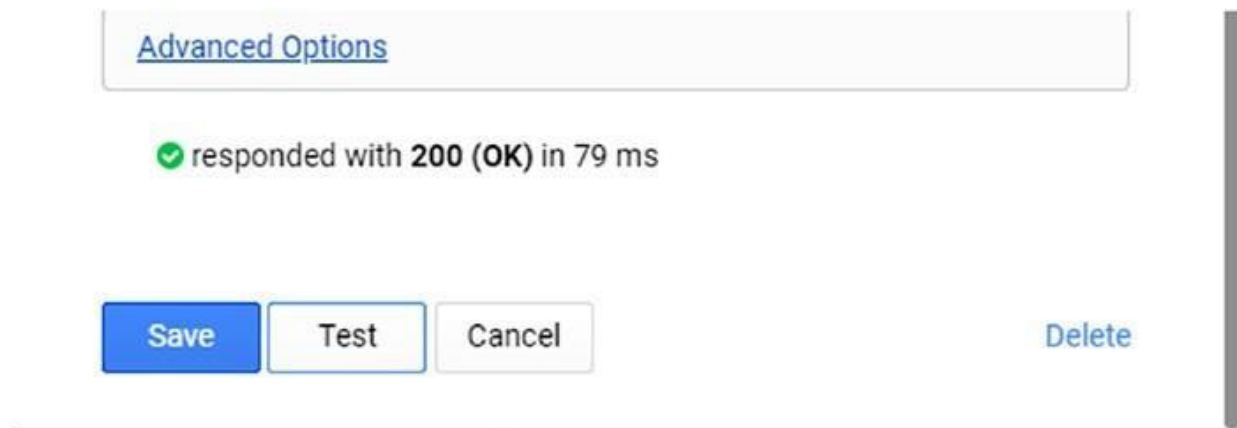


Figure 7-13. *The test result from the alert we created*

We can see that the test was successfully executed, and the result is a 200. This means that the alert can reach the instance and the service we want to monitor.

Policy Alert Configuration

We have now an alert to check the status of our system. An alert by itself is not really useful. This is because intercepting it requires a human to stay in front of the dashboard, to see when an alert is raised.

We can associate the alert with a specific policy alert. A policy alert is used to advise a human of an error occurring on the system. We can, for example, send an e-mail or add a notification in a HipChat room, or we can create a

policy alert by clicking the Create alert policy button. This shows a page on which we can configure our policy. We can configure four aspects of the policy.

- Conditions
- Notifications
- Documentation
- Name

The page allows us to create specific values for our policy. The first section is Conditions. The condition is all the choices we can make to create our alert. There are four types from which we can choose to create the alert.

- *Basic Types* : In this section, we can select some basic values for our alert. We also can indicate a metric threshold, for example, the CPU usage or the disk I/O read or write. We can indicate the value of the threshold and whether it must be above or below that threshold and for how long. Another condition we can put in place is the absence of metrics. For example, we can raise an alert if we don't have any CPU usage for 30 minutes.

- *Advanced Types* : This metric is used to define the increase or decrease in the use of a resource within a specific time frame. It can be used, for example, to look into an increase in usage of the resource and to scale up or down accordingly.

- *Basic Health* : This metric is used to give us a basic check regarding the uptime of the service. We configure this condition with the alert we previously created.

- *Advanced Health* : This is used to check the process on the instance. We can specify the process we want to check and how long it is live.

The next section is Notifications. This is used to send an alert to a human for processing or for acknowledgment. The basic notification is an e-mail. This means we send an e-mail to a specific recipient we have configured. In a monitoring system, this e-mail can be sent to a group. This way, all the members of the group receive the alert and can address it.

Documentation is a field we can use to document the type of alert we want. It is important to define good documentation for the alert. We can use some markup to create our documentation in connection with the alert.

The last section is the Name of the policy, which is essentially the name of the policy. We can now create our policy for the alert we created previously. In the monitoring section, we create a new policy alert. We created the policy for the uptime, but we must create a basic health check, to notify by e-mail when an error is raised. We select basic health and the instance we want to monitor. We connect an e-mail to the notification section, add some documentation, give a name, and save. Now we have a policy connected to our alert.

Creating a Dashboard

One of the basic components of a monitoring system is a dashboard for having a graphical representation of our system. This is used to get a clear and immediate idea of the status of the system.

To create a dashboard in Stackdriver, we select Dashboard and then Create Dashboard. This takes us to a blank dashboard, which we can use to create our personal dashboard (Figure 7-14).

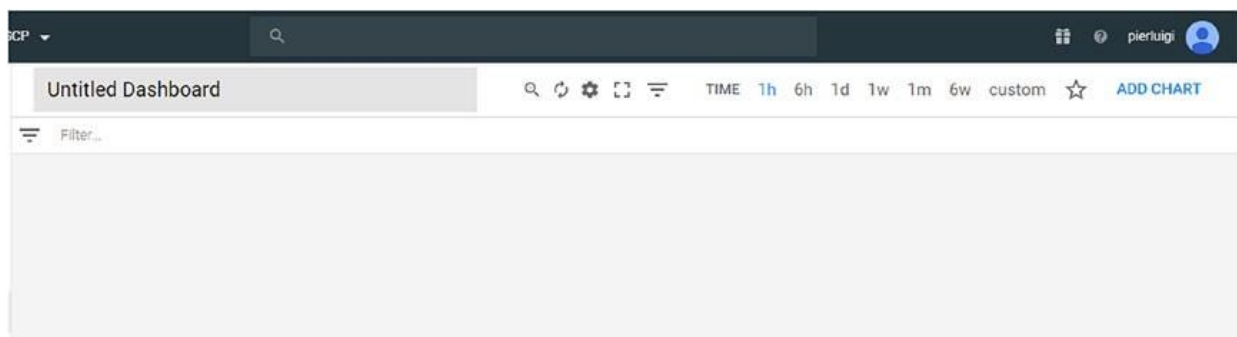


Figure 7-14. *The blank dashboard we use to create our personal dashboard*

The first step is to change the name of the dashboard. We can change the name simply by clicking Untitled Dashboard and then configuring the name we want for the dashboard, in my case, Stackdriver GCP Dashboard. We can add the first graph simply by clicking the ADD CHART button. This opens the page for creating our first graph. First, we must create the resource we

want to add to the graph. When we click the Find Resource type and metrics text box, the page shows a drop-down box from which we can start to choose the type of resource we want to add to the graph. In our case, we add a GCE VM Instance. After that, we can choose what metric we want to monitor. In our case, we choose Uptime. Click the Save button to create the chart (see Figure 7-15).

Figure 7-15. *The Add Chart page*

We have now created the first chart. We can add another chart on the dashboard. If we want to add the CPU load average, we can repeat the steps we just took, but in this instance, we check the metric CPU usage. The dashboard will look like that in Figure 7-16 .

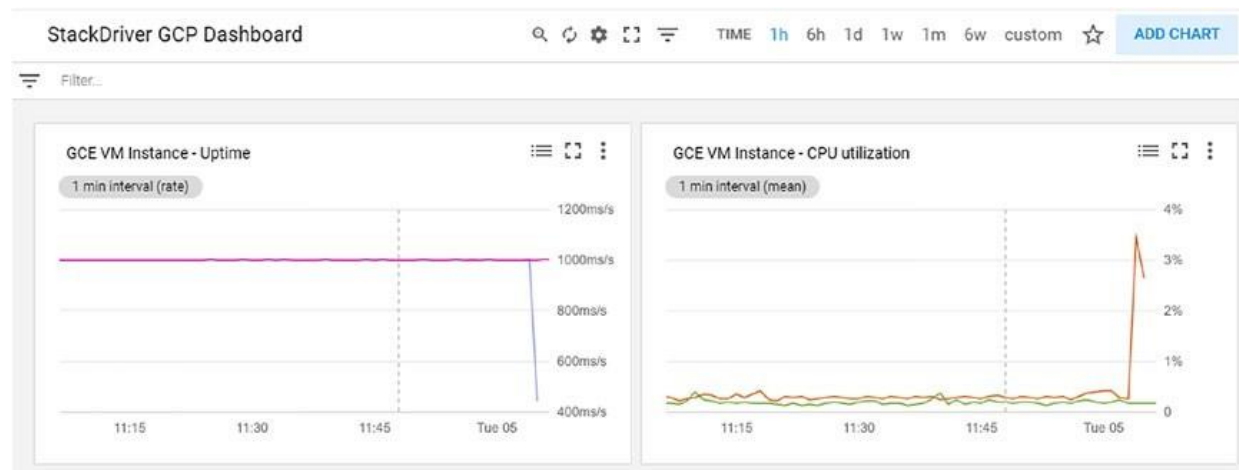


Figure 7-16. *The dashboard with two monitors in place*

Testing the Dashboard

Now that we have our monitoring system in place, we must make sure it works properly. So, we stop our instance and, after five minutes, check if the alert has been raised correctly. Stop the instance and see what's happening. When the time to check arrives, we can see that the dashboard reveals the incident (Figure 7-17).

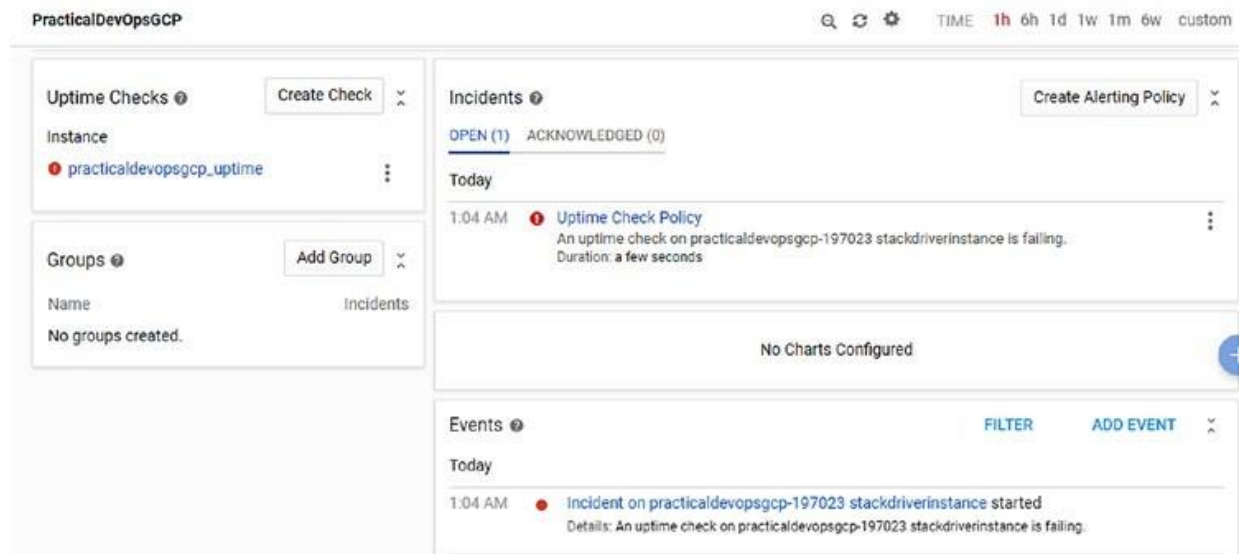


Figure 7-17. The dashboard shows the error with the uptime

Because we have set up an e-mail alert, we receive a message telling us about the incident occurring at the moment. The message looks like that in Figure 7-18 .

Figure 7-18. The mail sent by the policy alert

The monitoring system shows the error and advises the human about it occurring in the system. We have now set up our first monitoring system on GCP.

Conclusion

Creating a good monitoring system is essential to achieve good results with our DevOps. To build a good monitoring system isn't easy. It requires experience and patient observation of our software. The main goal of the monitoring system is to identify and react to any errors that occur and, possibly, to be used to debug errors in production. There is a lot of software we can use to create our own monitoring system. In this chapter, we used Stackdriver, the solution offered by Google, to monitor the GCP. Monitoring is a very huge area, and in this chapter, we saw common techniques we can use to design our system. All the techniques can be adapted to your favorite software. The important part of monitoring is the theory behind it and the alert system.

CHAPTER 8

Creating and Managing Infrastructure in GCP

One of the most critical aspects of DevOps and the cloud is infrastructure. When we design and implement our infrastructure in the cloud, we always want the results to be the same. For this, it is important to create and maintain our own virtual image. In this chapter, you will learn how to create and manage your own virtual image in Google Cloud Platform (GCP).

Infrastructure As Code

You have seen that the DevOps movement was born officially at the Toronto Agile conference in 2008, when Patrick Debois presented the talk “Agile Infrastructure and Operations.” During his talk, Debois presented a way of putting a development together with both the infrastructure and operational features. Thus, the DevOps movement was born.

This vision opens the way to what we call *infrastructure as code* , or IaC. When we talk about IaC, we are referring to infrastructure that can be programmatically designed and released. An example is when we create our Docker images in connection with our software, or when Chef, Puppet, or any other configuration software is programmed to prepare our infrastructure.

When we think of adopting an IaC, it is primary to ensure consistency across the server and achieve a consistent result with the operation, every time we execute the same operation.

Adopting IaC can drastically reduce the operation and release times. This is because we can create the staging server with exactly the same software we use in production, which means that the development team can test the software in an environment similar to production. This is possible, because we define the infrastructure and deploy it in a similar way as we deploy the software.

With IaC, we essentially promote the procedure used in development when we design and implement our infrastructure. This is done to promote the consistency of the operation, the stability of the system, and a repeatable result of the operations.

When we adopt IaC for our infrastructure, it is because we want to achieve some specific goals.

- *Easy support* : When we implement IaC, the support is simpler, because it is easier to return a functional status to the infrastructure.

- *Make the infrastructure change routines* : With IaC, we can update our infrastructure every day or every hour. This is because we follow the same procedure we use when we release the software.

- *Easy failure recovery* : With IaC, in case of disaster, the infrastructure can come back in a definite state, because we can add software, such as Terraform or GCD, to fix it.

- *Continuous improvement* : IaC improves our infrastructure daily, because we follow the same principles we have for the software development. Any small improvement is released in staging, tested, and, finally, released into production.

The cloud is the natural environment for IaC. When we create a new machine in the cloud environment, we create a new virtual machine (VM), with a specific version of OS and with a specific version of the software. When we want to spin up a new server, we can easily start the VM we created and use it as the basis of our new server.

Infrastructure as Code Principles

To move our infrastructure toward IaC, we must adhere to some basic principles:

- Repeatability
- Consistency
- Disposability

These three principles form the backbone of IaC, we and must keep them in mind when we design our infrastructure. We will now consider these principles in greater detail.

The first principle we must bear in mind is that every infrastructure must be repeatable. When we design IaC, it must be repeatable. Infrastructure is repeatable when it has code and/or procedures for replicating it that always have the same type of infrastructure.

The second principle derives directly from the first. Every IaC must be consistent. This means that every time we rebuild or add a new node in the infrastructure, we must always have the same result. At least, we mustn't change the definition of our infrastructure.

The last principle we must follow is that every IaC must be disposable. This means that we must have the capacity to destroy, update, and resize at every opportunity we require. The ability to be disposable helps to scale the infrastructure and fix an issue when the infrastructure is running.

To implement this principle, we must follow certain practices. These will drive our implementation and ensure the correct functionality of our infrastructure.

- *Define the infrastructure in a file* : To ensure a consistent and repeatable process, it must always give the same result. For this, we can define our infrastructure in a file. In this file, we can define all we need for our infrastructure, for example, DNS, disk space, OS, etc. Keeping all this information in a file can help to define our infrastructure programmatically.
- *Define the document system and the process*: It is important to have a clear definition of what a system or a process does in our infrastructure. To achieve this, it is very important to document the system and the process. Having clear documentation helps to maintain the infrastructure over time. Members of a team can read the script and follow the documentation to put improvements in place or fix issues with the infrastructure.
- *Versioning the infrastructure file* : To maintain the consistency of the infrastructure and to have a good process in place, versioning the definition

file for the infrastructure is important. By versioning the file, we can easily recover the infrastructure to any defined state. This helps the consistency of the process and the availability of the infrastructure. Versioning the file for the infrastructure is important to roll back the infrastructure, in case of a fatal error.

- *Test the infrastructure first* : We define our infrastructure in a file, and we use a program to put it in place. This means that we can test the infrastructure first, to move that into production. Testing the infrastructure, for example, creates a staging infrastructure to test our definition file.

- *Make small changes* : It is better to make a small change to the infrastructure than a big one. A small change in the infrastructure makes it easier to isolate the root cause of an issue and then fix it. If we release a big change in the infrastructure, it can be more difficult to do this, and we have to roll back everything for the infrastructure to work again.

When we follow these practices, we can define a state for our infrastructure and then ensure that we have this specific state. We create what is defined as a *defined configuration state* .

Note By *defined configuration state* , we mean an infrastructure that can be observed and found in a specific well-defined state. This can be achieved by describing the infrastructure in the file and defining the *state* of the infrastructure at every step. We can also *query* the infrastructure, to obtain an answer for the infrastructure at a specific moment.

When we have an infrastructure in a defined configuration state, we can easily observe the infrastructure and monitor it. Because the infrastructure is defined in a file, we can define the state for every step of the infrastructure, and we can, for example, disable the alert for a specific piece of the infrastructure when, for example, we update it. This reduces the noise and makes the alert more efficient.

Architecture for Infrastructure As Code

When we want to implement IaC, we essentially want to design a centralized system for managing our infrastructure. There are three different approaches

we can use to define our definition file for our infrastructure:

- Declarative
- Imperative
- Intelligent

The *declarative* approach describes *what* the configuration should be. With the declarative approach, we define the desired state of the infrastructure. The system executes all the operations necessary for putting the infrastructure in the state we define.

The *imperative* approach concentrates on *how* the configuration should be. The system executes the operations one after the other, to move the system in the defined state.

The *intelligent* approach concentrates on *why* the infrastructure must have this specific state and status. This decision takes into consideration the state of all the components of the infrastructure and analyzes the status of the co-components of our infrastructure. To configure the infrastructure, the tool for the configuration can use two different methods:

- Pull
- Push

The difference between the two methods is only in how the central server configures the client. With the *pull* method, the client asks the central server what the configuration state is. With the *push* method, the server tells the client what the configuration state is.

Infrastructure As Code in Google Cloud Platform

In GCP, we can create our IaC using a tool called *Google Deployment Manager* (GDM) This allows us to define all the resources for our infrastructure, by creating a YAML file in a declarative way.

We can also use Python or the Jinja2 template to create a configurable template file. We can, for example, define a parameter for the load balancer, autoscaling, and instance group. With GDM, we can define a template. This

can be used for reuse in our deployment and, of course, stored in the repository.

GDM is built around some fundamental components that we must understand to use the tool effectively. These components are

- Configuration
- Templates
- Resource
- Types
- Manifest
- Deployment

Configuration

The configuration is the YAML file in which we define the list of resources we want for our deployment. This can be the entire infrastructure or, more likely, a piece of our infrastructure.

To define a resource in the file, we must use the syntax `resources:`. Every resource must be defined by three components:

- `name`: the name for the resource we want to create
- `type`: the type of resource we want to create
- `properties`: the parameter defined for the resource

A sample resource file given in Listing 8-1 .

Listing 8-1. Sample Configuration File

```
resources:
- name: gcp-devops-vm
  type: compute.v1.instance
  properties:
    zone: us-central1-a
    machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp/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/debiancloud/global/images/rhel-6-v20180611
networkInterfaces:
- network: https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp/global/networks/default
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
```

Templates

The configuration is fine when we want to define a simple image, such as in the preceding listing, but when we want to define a more complex infrastructure, this is not sufficient. In that case, we can create a template.

A template is essentially a reusable piece of configuration. The template is defined in an external file and is used in the configuration as a type. The template can be written using the Jinja 2.8 syntax, which is similar to the YAML syntax, or with Python 2.7. An example of a static Jinja template is shown in Listing 8-2 .

Listing 8-2. Example of a Jinja Template

```
- name: gcp-jinja-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-8
networkInterfaces:
- network: global/networks/default
```

We can see that the Jinja template is similar to a normal template definition. Using Python gives us more flexibility. This is because we can use the language and the library to define our template. An example of the template defined in Python follows (Listing 8-3).

Listing 8-3. Python Template Definition `"""Creates a Compute Engine Instance."""`

```
def GenerateConfig(context): """Generate configuration."""
resources = []

# [START use_basic_template]
resources.append({
'name': 'gcp-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-8'

}
}],
'networkInterfaces': [{
```

```
'network': 'global/networks/default'
}]
}
})
```

```
# [END use_basic_template]
return {'resources': resources}
```

The two necessary requirements when we use Python to define our template are these:

- The code must define a method called either `GenerateConfiguration` or `generate_configuration`. If both methods are used, the compiler gives priority to the `generate_configuration` method.
- Because the configuration is defined in a YAML file, the method must return a valid YAML file.

In the preceding code, we defined a `GenerateConfiguration` method simply by writing our YAML file. There is another way to define the configuration. We can use a Python variable, to make some part of the file dynamic (Listing 8-4).

Listing 8-4. The Template Created with Python Variables `"""Creates a Compute Engine Instance."""`

```
def GenerateConfig(context): """Generate configuration."""
```

```
resources = []
# [START use_template_with_variables]
resources.append({
    'name': 'gcp-' + context.env['deployment'],
    'type': 'compute.v1.instance',
    'properties': {
        'zone': 'us-central1-a',
        'machineType': ".join(['zones/', context.properties['zone'],
        '/machineTypes/n1-standard-1'])", 'disks': [{
        'deviceName': 'boot',
        'type': 'PERSISTENT',
        'boot': True,
        'autoDelete': True,
```

```

'initializeParams': {
'sourceImage':
'projects/debian-cloud/global/images/family/debian-8' }
}],
'networkInterfaces': [{
'network': 'global/networks/default'
}]
}

})
# [END use_template_with_variables]
return {'resources': resources}

```

In the preceding code, we define some variable for making the configuration more dynamic, instead of creating only the YAML file.

With Python, we can easily create a very complex dynamic configuration, and we can adapt the configuration, in our case, for example, the name, based on the environment variable. This is useful when, for example, we want to identify the VM based on the server it runs.

When we have defined our template, we can import it in our configuration file. We can import our template with the extensions .py, .jinja, or any non-template extension, such as.txt. To import the file, we can indicate the path of our file, and if we want to assign a name, indicate the name. It is possible to import more than one template, and, of course, we can mix templates. For example, it is possible to import one Python and one Jinja template (Listing 8-5).

Listing 8-5. Code to Import a Template in the Configuration File

```

imports:
- path: /path/to/gcp_template.jinja
- path: gcp_new_template.py

```

```

name: gcp_infrastructure.py

```

When we have imported our template, we can use it like atype in the

configuration file (Listing 8-6).

Listing 8-6. Importing and Using a Template

```
imports:
- path: /path/to/gcp_template.jinja
resources:
- name: my_gcp_template
-type: /path/to/gcp_template.jinja
```

It is possible to use a template without a configuration file. To do this, we can use the command line and compile the template directly. The relevant command is shown in Listing 8-7 .

Listing 8-7. How to Invoke the Template by the Command Line

```
gcloud deployment-manager deployments create my-vm-gcp \
--template gcp_template.jinja \
--properties zone:us-central1-b
```

Resource

The resource represents a single API resource. This API can be provided by Google or we can create our resource by defining a new type. For example, a Google Compute Engine is a resource. We can create our resource using a Jinja or Python template. A resource is the basic information we must define to create our configuration file.

Types

A resource is used to define what we want to do, but a type is used to identify the resource and allow us to create a deployment process.

The type is used to identify the resource. There are two types of resources:

- *Base type*: This is a single resource, like Google Compute Engine.
- *Composite type* : This is a set or resource.

The base type is essentially a single type of resource offered by Google, such as Google Compute Engine, a basic storage, or SQL.

The composite type contains one or more templates. These templates are preconfigured to work together and, of course, can be defined using Python or Jinja. A composite type is normally used to define a piece of the template and can be reused easily. For example, we can create a load-balancer resource for our network.

Manifest

A manifest is a description of our original configuration. This is read-only and is used normally to give as much information as possible about the deployment. We use the manifest to describe the resource and the types of our configuration. A manifest returns a value in the following form (Listing 8-8).

Listing 8-8. Example of a Manifest

```
config:
content: |
resources:
- name: gcp-first-vm

type: compute.v1.instance properties:
zone: us-east1-b
machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-east1-b/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/
practicaldevopsgcp-197023/global/networks/default accessConfigs:
- name: External NAT
```

type: ONE_TO_ONE_NAT
- name: gcp-second-vm
type: compute.v1.instance
properties:
zone: us-east1-b
machineType: <https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/g1-small>
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/practicaldevopsgcp-197023/global/networks/default> accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
expandedConfig: |
resources:
- name: gcp-first-vm
properties:
disks:
- autoDelete: true
boot: true
deviceName: boot
initializeParams:
sourceImage: <https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/family/debian-9>
type: PERSISTENT
machineType: <https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/f1-micro>
networkInterfaces:
- accessConfigs:
- name: External NAT

type: ONE_TO_ONE_NAT
network: <https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/networks/default> zone: us-east1-b
type: compute.v1.instance
- name: gcp-second-vm
properties:
disks:
- autoDelete: true
boot: true
deviceName: boot
initializeParams:
sourceImage: <https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/family/debian-9>
type: PERSISTENT
machineType: <https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/g1-small>
- accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
network: <https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/networks/default>
zone: us-east1-b
type: compute.v1.instance
id: '2742029496220147117'
insertTime: '2018-08-16T13:49:06.882-07:00'
layout: |
resources:
- name: gcp-first-vm
type: compute.v1.instance
- name: gcp-second-vm
type: compute.v1.instance
name: manifest-1534452546871
selfLink: <https://www.googleapis.com/deploymentmanager/v2/projects/practicaldevopsgcp-197023/global/deployments/gcp-vm-deployment/manifests/manifest-1534452546871>
manifest-1534452546871

We can see that the manifest describes all the resources actually defined and installed in our deployment.

Deployment

A deployment is a collection of resources deployed and managed altogether. We can create a deployment using a configuration file and a command line. The command for executing the deployment is shown in Listing 8-9 .

Listing 8-9. Command for Creating a New Deployment in GCP
gcloud deployment-manager deployments create gcp-first-deployment \\\n--config gcp-vm.yaml

To define the configuration file, we must employ the --config option, followed by the name of the configuration file. When the deployment is complete, we can see whether the deployment is correctly configured, by using the following command (Listing 8-10):

Listing 8-10. Command to Describe the Deployment gcloud deployment-manager deployments describe gcp-first-deployment

Starting with Google Cloud Deployment Manager

Up to now, you have seen the basic concepts of Google Cloud Deployment Manager. At this point, we can start to define our own IaC.

As you learned, we can define our infrastructure by creating a configuration file, and we can create a deployment via a command line. This essentially deploys our resource in our infrastructure. The first step is to define our configuration file. For starters, we create a simple virtual image with a Debian image. The file is like that shown in Listing 8-11 .

Listing 8-11. The First Configuration File for Our Google Cloud

resources:
- type: compute.v1.instance
name: gcp-first-configuration-vm
properties:

```
zone: us-east1-b
machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-central1-f/machineTypes/fl-micro
disks:
- deviceName: boot
  type: PERSISTENT
  boot: true
  autoDelete: true
  initializeParams:
    sourceImage:
      https://www.googleapis.com/compute/v1/projects/debiancloud/global/images/family/debian-9
- network: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/global/networks/default accessConfigs:
- name: External NAT
  type: ONE_TO_ONE_NAT
```

Note When we create our infrastructure, we can indicate the family for our VM, as in the preceding code. We can find all the families for our operating system here: <https://cloud.google.com/compute/docs/images#os-computesupport>.

In this configuration file, we indicate some information necessary for defining our VM. We define a zone, in our case, us-east1-b; a machineType, in our case, fl-micro; a boot disk; and a random IP address. These values are necessary to identify and to deploy our VM. We must now deploy our infrastructure. To do so, we must use the command line. The command we use for the deployment is

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

Note The YAML file is a space-sensitive file. At the time of publication of this book, it is possible that some images have been removed. It is recommended, therefore, that you download the file from the Git connected with this book.

The command is essentially what we describe to create a new deployment. We can now construct, or import, the file in our Google Cloud Console and run it. It deploys our resource and shows the result of the operation (Listing 8-12).

Listing 8-12. Result of the Google Deployment of Our Resource

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud deployment-manager
deployments create gcp-first-deployment --config gcp_vm.yaml The
fingerprint of the deployment is 4Sb9EZuc2qx92SWmbnmLig== The
fingerprint of the deployment is 4Sb9EZuc2qx92SWmbnmLig==
d094a48e]...done.
Create operation operation-1534453555625-57393cf1b7c2b-27da25f9-
d094a48e completed successfully.
NAME TYPE STATE ERRORS INTENT gcp-first-configuration-vm
compute.v1.instance COMPLETED []
```

We have now successfully completed the installation of our first piece of the infrastructure. We can see the details of our new infrastructure, and we can describe it. The command we must use to describe the infrastructure is

```
gcloud deployment-manager deployments describe gcp-first-deployment
This command returns detailed information about our deployment (Listing
8-13 ).
```

Listing 8-13. Command Used to Describe the Deployment

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud deployment-manager
deployments describe gcp-first-deployment
--
fingerprint: 4Sb9EZuc2qx92SWmbnmLig==
id: '4465263803798936028'
insertTime: '2018-08-16T14:05:55.750-07:00'
manifest: manifest-1534453555753
name: gcp-first-deployment
operation:

endTime: '2018-08-16T14:06:14.117-07:00'
name: operation-1534453555625-57393cf1b7c2b-27da25f9-d094a48e
```

```
operationType: insert
progress: 100
startTime: '2018-08-16T14:05:56.070-07:00'
status: DONE
user: pierluigi.riti@gmail.com
```

```
NAME TYPE STATE INTENT gcp-first-configuration-vm
compute.v1.instance COMPLETED
```

This command shows us that our VM has been created and is ready to use. We can check and connect to the VM directly via the Compute Engine page, if we go to Compute Engine and connect to it (Figure 8-1).



Figure 8-1. The Compute Engine page, showing our first configuration

Because we created the infrastructure using GDM, we can also inspect the configuration using the GDM page. We can find the resource under the tool's menu (Figure 8-2).

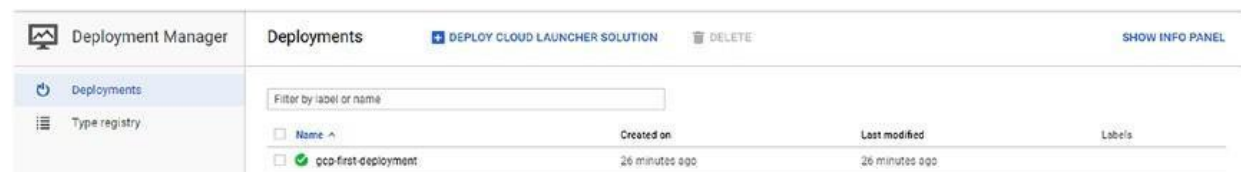


Figure 8-2. The Deployment Manager page

Now we can see that we have our first piece of infrastructure defined by the code. To complete the life cycle, we must learn how we can destroy the infrastructure. The command we use to destroy our infrastructure is

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

This shows a confirmation question about the deletion of our infrastructure. We can confirm the operation by pressing y. This deletes our infrastructure.

The operation can take several minutes to complete. The result is a message advising us that the infrastructure has been deleted (Listing 8-14).

Listing 8-14. Command to Delete an Infrastructure

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud deployment-manager
deployments delete gcp-first-deployment
The following deployments will be deleted:
- gcp-first-deployment
Do you want to continue (y/N)? y
Do you want to continue (y/N)? y
d3b921ed]...done.
Delete operation operation-1534453886807-57393e2d8ebd8-e48703b3-
d3b921ed completed successfully.
```

You have learned how to create basic IaC, but our example is really simple. Real infrastructure is more complex than a simple VM. Let's turn our attention to how to create more complex infrastructure and, more important, how to use a template to define our infrastructure.

Upgrading Our Infrastructure

When we define an infrastructure, we don't have only one VM. Our infrastructure can have more than one single resource. For example, we can have different VMs, a load balancer, etc. In the preceding section, you defined a configuration file with only one VM. Now you will see how it is possible to create a configuration file, using our example, with more than one resource. Let's suppose we want to create two simple VMs for our infrastructure. To do that, we add another VM to the previous one. The file for that follows (Listing 8-15).

Listing 8-15. The Configuration File with Two Different VMs

```
resources:
- name: gcp-first-vm
  type: compute.v1.instance
properties:
  zone: us-east1-b
```

```
machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/f1-micro disks:
- deviceName: boot
type: PERSISTENT
boot: true
autoDelete: true
initializeParams:
sourceImage:
https://www.googleapis.com/compute/v1/projects/debiancloud/global/image
s/family/debian-8
networkInterfaces:
- network: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/global/networks/default
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
- name: gcp-second-vm
type: compute.v1.instance
properties:
zone: us-east1-b
machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/g1-small disks:
- deviceName: boot
type: PERSISTENT
boot: true
autoDelete: true
initializeParams:
sourceImage: https://www.googleapis.com/compute/v1/projects/ debian-
cloud/global/images/family/debian-8
networkInterfaces:
- network: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/global/networks/default
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
```

In this file, we created a second VM of a different size. To create a new resource, in this case, a virtual machine, we can add another resource to our

deployment. In our case, we started with

```
- name: gcp-second-vm
type: compute.v1.instance
properties:
```

```
zone: us-east1-b
machineType: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/g1-small disks:
- deviceName: boot
type: PERSISTENT
boot: true
autoDelete: true
initializeParams:
```

```
sourceImage: https://www.googleapis.com/compute/v1/projects/debian-
cloud/global/images/family/debian-8
networkInterfaces:
- network: https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/global/networks/default
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
```

You can see that we restart a new section to define a new resource. We don't have to reopen another resource section. This is because, under one resource, we can have more than one single resource. We can now deploy our resource, to see the two instances created on the system. The command for deploy is

```
gcloud deployment-manager deployments create gcp-second-deployment --
config gcp-two-vm.yaml
```

The result shows the status of both VMs (Listing 8-16).

Listing 8-16. Result When We Deploy Both VM Images

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud deployment-manager
deployments create gcp-vm-deployment --config gcp-two-vm.yaml The
fingerprint of the deployment is tReLCx9Y7aa7QzguA54V2Q== The
```

fingerprint of the deployment is tReLCx9Y7aa7QzguA54V2Q==
2b46debe]...done.

Create operation operation-1534452546303-5739392f27419-af9ff154-
2b46debe completed successfully.

NAME TYPE

gcp-first-vm compute.v1.instance gcp-second-vm compute.v1.instance

STATE ERRORS INTENT COMPLETED []

COMPLETED []

Expanding and Customizing Our Deployment

You've learned how to create a simple configuration file with one or two resources. That is good, but still not sufficient to create our deployment. When we create more than one resource in our configuration file, we might have issues related to dependencies across the resource, for example, when we use the first network we created. By default, GDM creates all the resources in parallel, which means we are not sure to have the resource in place when we need it.

GDM helps us to define the order in which to use the resource. We can create a hierarchy for the resource, and GDM decides the order and the parallelism to install the resource. We can't push the order of deployment, but we can define the dependencies of the types in a dynamic way. We can do that by defining a reference.

The reference is used to force Deployment Manager on the dependencies and then create a specific resource only when all the dependencies are available. Using a reference in our deployment offers us some advantages.

- *We can give the deployment manager an order to resolve the resource .*

Imagine, for example, that we must create a VM with a connected network. Deployment Manager first creates the network and then, when the network is available, creates the VM.

- *We can use the reference for self-reference to an internal resource of the configuration .* We can use a selfLink to indicate an internal resource. For example, we can add a reference to a specific IP network and use it for defining our network

It is possible to indicate a resource in our configuration file using the syntax \$(ref.RESOURCE_NAME.PROPERTY)

Imagine that we want to specify a family of IP addresses for our previous VM. To do this, we first must create the resource in the configuration file, then reference our VM to this new resource. To create the network resource, we first must add that to the end of the configuration file. What we add is

```
- name: gcp-network
type: compute.v1.network
properties:
```

```
IPv4Range: 10.10.0.1/16
```

Now that we've created the new network resource, we need to connect the network interface on our previous configuration file and update it with the following code (Listing 8-17).

Listing 8-17. Configuration File with the Two VMs and the Network Configuration

```
networkInterfaces:
```

```
- network: $(ref.gcp-network.selfLink)
```

How we can see we use the reference for use the previous configuration network we had created, the final file is like that:

```
resources:
```

```
- name: gcp-first-vm
```

```
type: compute.v1.instance
```

```
properties:
```

```
zone: us-east1-b
```

```
machineType: https://www.googleapis.com/compute/v1/projects/
```

```
practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/f1-micro
```

```
disks:
```

```
- deviceName: boot
```

```
type: PERSISTENT
```

```
boot: true
```

```
autoDelete: true
```

```
initializeParams:
```

```

sourceImage:
https://www.googleapis.com/compute/v1/projects/debiancloud/global/images/family/debian-8
networkInterfaces:
- network: $(ref.gcp-network.selfLink)
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
- name: gcp-second-vm
type: compute.v1.instance
properties:
zone: us-east1-b
machineType: https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/zones/us-east1-b/machineTypes/g1-small
disks:
- deviceName: boot
type: PERSISTENT
boot: true
autoDelete: true
initializeParams:
sourceImage:
https://www.googleapis.com/compute/v1/projects/debiancloud/global/images/family/debian-8
networkInterfaces:
- network: $(ref.gcp-network.selfLink)
accessConfigs:
- name: External NAT
type: ONE_TO_ONE_NAT
- name: gcp-network
type: compute.v1.network
properties:
IPv4Range: 10.10.0.1/16

```

We now try our new configuration file. The command to deploy the configuration file is `gcloud deployment-manager deployments create gcp-two-vm-network --config gcp-two-vm-network.yaml`

The result of the operations is shown in Listing 8-18 .

Listing 8-18. Resource Created with the Network Reference

```

pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud deployment-manager
deployments create gcp-two-vm-network --config gcp-two-vm-
network.yaml The fingerprint of the deployment is
FTlZdQbTWsmDfmCGHfbZZQ== The fingerprint of the deployment is
FTlZdQbTWsmDfmCGHfbZZQ== 5dc88e00]...done.
Create operation operation-1534455595680-5739448b44101-0e70d587-
5dc88e00 completed successfully.

```

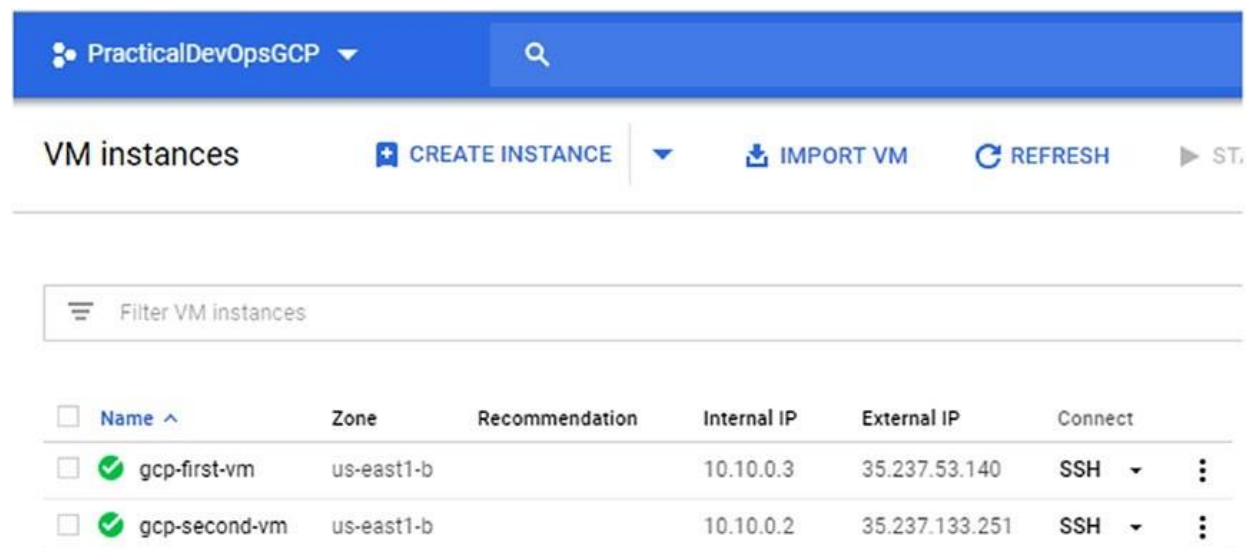
NAME

```

gcp-first-vm gcp-network gcp-second-vm TYPE STATE ERRORS INTENT
compute.v1.instance COMPLETED []
compute.v1.network COMPLETED []
compute.v1.instance COMPLETED []

```

We can see that the new IP assigned to the VM simply opens the Google Cloud Console and goes to the Google Compute Engine section. This indicates both the instance and the new IP assigned (see Figure 8-3).



Name	Zone	Recommendation	Internal IP	External IP	Connect
gcp-first-vm	us-east1-b		10.10.0.3	35.237.53.140	SSH
gcp-second-vm	us-east1-b		10.10.0.2	35.237.133.251	SSH

Figure 8-3. The two VM images with the address assigned by the new network resource

The new internal IP is now one of the family we previously defined in the network. Because we have defined the resource, the VM and the network are in only one configuration file. If we go to the Deployment Manager page, we see only one configuration (Figure 8-4).



Figure 8-4. The Deployment Manager page, showing the new configuration file

Now we can see that there is only one deployment file, but inside it, we can create more than one resource. This is helpful when we construct our infrastructure. We can create more than one configuration file and deploy with different resources inside it. We can now clean the environment and learn how to build a template for creating a more dynamic configuration, using Python. To clean the environment, we can use the following command:

```
gcloud deployment-manager deployments delete gcp-two-vm-network
```

Creating a Template for Our Deployment

With GDM, it is easy to create IaC, using a simple configuration file. We can use the reference to manage the dependencies across the resource, but if we want to have more flexibility, we must use a template.

Using a template gives us the flexibility of using a programming language to define and create our resource. A template is a separate file write using the Jinja syntax or the Python language. In our example, we use the Python language. What we want to do is re-create the configuration file we developed for the two VMs with the network, using the Python syntax. When we define a Python template, we must respect two simple rules.

- We must define a `GenerateConfig()` or `generate_config()` file. In case we define both, Python uses `generate_config()`.
- We must return a valid YAML file.

When we use a template for our configuration, we can create a module, which we can reuse to define pieces of code. We can use the template to define a resource we use in our configuration.

To show how to create a template for defining both VMs in a separate file, the first file looks like Listing 8-19 .

Listing 8-19. A Template Resource Created with Python

```
COMPUTE_URL_BASE = 'https://www.googleapis.com/compute/v1/'
def GenerateConfig(unused_context): """Creates the first virtual
machine."""
```

```
resources = [{ 'name': 'gcp-first-template',
'type': 'compute.v1.instance',
'properties': {

'zone': 'us-east1-b',

'machineType': ".join([COMPUTE_URL_BASE, 'projects/
practicaldevopsgcp-197023', '/zones/us-east1-b/',

'machineTypes/fl1-micro'])), 'disks': [{
'deviceName': 'boot',
'type': 'PERSISTENT',
'boot': True,
'autoDelete': True,
'initializeParams': {

'sourceImage': ".join([COMPUTE_URL_BASE, 'projects/', 'debian-
cloud/global/', 'images/family/debian-8'])

}
}],
'networkInterfaces': [{

'network': '$(ref.a-new-network.selfLink)',
'accessConfigs': [{
'name': 'External NAT',
'type': 'ONE_TO_ONE_NAT'
}]
}]
}]
}
```

```
}]  
return {'resources': resources}
```

The Python code defines in the method `GenerateConfiguration()` the YAML file for the resource we want to define. The interesting part in this code is the line

```
'network': '${ref.a-new-network.selfLink}',
```

We essentially use a reference to define the configuration file inside our template. This starts to reveal the true power of the template.

The template is essentially a piece of configuration code that we can reuse. When we use a template, our configuration file imports the template and then uses it to define the resource. This allowed us, for example, to create a common component, such as a VM, load balancer, Docker configuration, etc., and reuse it across the project.

Because the file is saved in a Python file, it can be easily saved and versioned in our code repo. We can define different versions of our infrastructure and release the version we need and, of course, roll back the infrastructure. For testing, we can use one VM for now, and we can create the first version for our configuration file, which would look like [Listing 8-20](#).

Listing 8-20. The First Version of Our Configuration File Using the Template imports:

```
- path: gcp-first-template.py  
resources:
```

```
- name: gcp-vm-1  
type: gcp-first-template.py  
- name: gcp-network  
type: compute.v1.network  
properties:  
IPv4Range: 10.10.0.1/16
```

The configuration file created in this case is very compact and easy to read. The configuration file first imports the template. This is done with

the imports section. imports:

- path: gcp-first-template.py

Under the import, we indicate the path. This indicates the file with the path we want to import in our configuration file. When we have imported the file, we can use it in the resources section. As with any other resource, we define the name and the type of the resource. In this case, the type is the template file we had imported previously.

- name: gcp-vm-1

type: gcp-first-template.py

This line defines and uses our template. We can define as many templates as we want in our configuration file. These can be used to define very complex configuration files and to reuse the resource we defined previously.

This file uses resources defined in the template file. This is not exactly more reusable, because we give the same type for all the resources. For a complete reusable template, we must modularize it. We can do that by using the environment variable in the template.

Defining the Template with the Environment Variable

With the environment variable, we can create a template that we can reuse across zones, areas, and projects. The environment variable is defined in the configuration file and then can be personalized for the specific project we want to create. We can refer to an environment variable using the following code in the template: `context.properties["property-name"]`. If we want to define the property in the configuration file, we can use `context.env["environment name"]`, in case we want to use the environment variable of our Google Cloud. We can now change our previous template and define an environment variable for our resource (Listing 8-21).

Listing 8-21. Adding an Environment Variable to the Template def GenerateConfig(context):

```
"""Creates the virtual machine with environment variables."""
```

```

resources = [{
  'name': context.env['name'],
  'type': 'compute.v1.instance',
  'properties': {

    'zone': context.properties['zone'],
    'machineType': ".join(['https://www.googleapis.com/compute/v1/',
    'projects/practicaldevopsgcp-197023/ zones/',
    context.properties['zone'], '/ machineTypes/',
    context.properties['machineType'])),
    'disks': [{
      'deviceName': 'boot',
      'type': 'PERSISTENT',
      'boot': True,
      'autoDelete': True,
      'initializeParams': {
        'sourceImage': ".join(['https://www.googleapis.com/ compute/v1/', 'projects/',
        'debian-cloud/global/', 'images/family/debian-8']) }
    }],
    'networkInterfaces': [{
      'network': '$(ref.' + context.properties['network'] + '.selfLink)',
      'accessConfigs': [{
        'name': 'External NAT',
        'type': 'ONE_TO_ONE_NAT'
      }]
    }]
  }
}]
return {'resources': resources}

```

In this template, we add the following environment variables:

- context.properties['network']
- context.properties['zone']
- context.properties['machineType']

These variables are used to create a different value in our template. The different value, in this case, enables us to deploy the resource in a different zone, with a different machine type and different network. We need to adapt

the configuration file, to use the environment variable. In the configuration file, we must add the variable and define the value we want to assign to it. The new configuration file is shown in Listing 8-22 .

Listing 8-22. New Template with the Environment and Properties Files Defined in the Context

```
COMPUTE_URL_BASE = 'https://www.googleapis.com/compute/v1/'
def GenerateConfig(context):

    """Creates the virtual machine with environment variables.""" resources = [{
    'name': context.env['name'],
    'type': 'compute.v1.instance',
    'properties': {

    'zone': context.properties['zone'],
    'machineType': ".join([COMPUTE_URL_BASE, 'projects/',
    context.env['project'], '/zones/', context.properties['zone'], '/machineTypes/',
    context.properties['machineType'])),
    'disks': [{
    'deviceName': 'boot',
    'type': 'PERSISTENT',
    'boot': True,
    'autoDelete': True,
    'initializeParams': {
    'sourceImage': ".join([COMPUTE_URL_BASE, 'projects/', 'debian-
    cloud/global/', 'images/family/debian-8']) }
    }],
    'networkInterfaces': [{
    'network': '$(ref.' + context.properties['network'] + '.selfLink)',
    'accessConfigs': [{
    'name': 'External NAT',
    'type': 'ONE_TO_ONE_NAT'
    }]
    }]
    }
    ]
    return {'resources': resources}
```

We need to update the configuration file to send the new variable to the template file. The new configuration file is shown in Listing 8-23 .

Listing 8-23. New Configuration File Created imports:

```
- path: gcp-first-template.py
resources:
```

```
- name: gcp-vm-1
  type: gcp-first-template.py
- name: gcp-network
  type: compute.v1.network
properties:
  machineType: f1-micro # Sets the 'machineType' template property zone: us-
  east1-b # Sets the 'zone' template property
  network: gcp-network # Sets the 'network' template property IPv4Range:
  10.10.0.1/16
```

We can use the same command to deploy the resource. Using the environment and property variables helps to define a fully reusable template. We can define our variable and reuse it across zones, regions, and, of course, the project.

Conclusion

In this chapter, we saw how to define IaC, using GDM. GDM is a very powerful tool for creating infrastructure, using a definition file. Using a file to define our infrastructure allows versioning of our infrastructure. In GDM, we can define the infrastructure, using a template. A template is a reusable piece of code that defines the resource of our infrastructure. Templates can be modularized, adding some environment and property value. The property values are defined in the configuration files. Conversely, the environment variables are gotten directly from GCP. IaC is one of the most important practices of DevOps, and it is crucial to learn, in order to define, develop, and maintain our infrastructure.

CHAPTER 9

Identity and Access Management with Google Cloud Platform

Today, identifying a user is important to guaranteeing a correct level of privacy for our application. In this context, identity and access management (IAM) becomes increasingly important. Google Cloud Platform (GCP) offers a good solution for defining our IAM and guaranteeing the privacy requirements of our resource.

What Is Identity and Access Management?

IAM is referred to as a framework for identifying users and allowing access to a resource only to the authorized user. IAM is an umbrella term to describe a set of software and rules that define the level of access a user has. With IAM, we define the level of access given to every user.

The user can be a customer, in which case, we can talk about customer identity management (CIM). For an employer, we talk about employer identity management (EIM). IAM, CIM, and EIM all have in common the same core objective: to uniquely identify the user and ensure that the user is exactly who he/she is.

When we think of an IAM framework, we must consider two main components:

- Software
- Governance

The software is all the tools we need to put our framework in place, from an IT point of view. Governance is the rules we define to authorize user access to a specific resource.

When we put in place a new IAM framework, it is because we want to achieve some specific goal. An IAM system is used to *identify* the user and then *authenticate* the user on the system when the user is authenticated. Based on the rules defining the governance, we can identify what IT resources the user is allowed to use. This creates a *digital entity* for the user.

Defining a Digital Entity

Every day, we learn about new cyber fraud or cyberattacks. This raises some critical imperatives. We must be sure to know the people we interact with. Using only an e-mail address and a password to identify a user is not sufficient, because a hacker can steal this information and fake the login.

When we create a digital entity, we want to be sure who exactly our users will be. For this, we collect a set of digital entities directly from the browser or mobile phone. We use all this data to identify the user. Having collected all the data, when a user tries an action, for example, a new login, the system compares the old data against the new data, using specific convergence patterns. This checks common points and, using specific risk patterns, for example, risk-based authentication (RBA), returns a number. This number is used to uniquely identify the user.

When we create a digital entity, we associate with the entity one or more *identity attributes*. These attributes are connected with the entity. For example, this can include medical data, e-mail, telephone numbers, usernames, bank transactions, etc. All this data is collected and grows every time we use our digital entity. All these transactions are used by an algorithm to identify the user. Imagine, for example, that we have only online bank transactions from the same browser, at the same time and from the same location. If we change the browser, time, and location one time, the system can *challenge* us, by asking, for example, that we introduce another code or respond to some security questions. A basic example of challenging a user is 2FA, or two-factor authentication. When we use a Google Authenticator for access to a site such as Facebook, the system challenges us by asking for another code after the password. This is used to authenticate our identity and allow us access to the site.

The Importance of Digital Identity

Today, almost everyone has a digital life —e-mail, a social media account, a smartphone, and probably an online banking account. All these define our “digital life,” a life in which we are identified only by our e-mail and, possibly, a password. But are these really sufficient for recognizing us?

In the United States alone in just the past three years, the incidence of identity theft has increased rapidly. A study conducted by Javelin Strategy & Research shows that since 2012, nearly 60 million US citizens were victims of identity theft, representing a total of more than \$100 billion stolen, and in 2017, the number of American victims of identity theft rose to 16.7 million.

Note The complete report is available at www.javelinstrategy.com/coverage-area/2018-identity-fraud-fraud-enters-new-era-complexity#.

Based on research conducted by Cisco, every person will have an average of four devices connected to a network in 2021. What that number makes clear is why digital identity will become increasingly important.

If we think about what we do daily, we can estimate how much data we move across a network, primarily the Internet, in the conduct of our daily activities. Examples might include using Google Maps to find a faster or alternative route to work, or using a credit card to purchase an ice-cold Frappuccino at our favorite coffee shop, or simply checking for e-mail on our smartphone. For these basic activities, we move masses of data across a network. All this data forms an “entity” that is connected to our digital life, because it identifies our tastes, locations, and banking details.

In this context, defining a secure digital identity is crucial. This is because when we make a transaction, the transactee must be sure we are who we are supposed to be.

Now we know that a digital entity is a set of attributes that define us. Every time we order an ice-cold Frappuccino, log in to our e-mail account, or post on our preferred social media site, we are adding an entity to our digital life, and this grows our digital entity. Our digital entity is essentially something similar to our DNA. When we think of developing a cloud application, we must put in place a set of rules to uniquely identify the user and protect his/her digital entity.

IAM and Digital Entity

IAM assumes a very important role in defining and protecting digital entity. An IAM system can involve four basic functions:

- Pure identification
- User access
- Authorization
- Federated identification

These four functions cover every aspect of IAM. The first and most important function for defining our digital entity is *pure identification*. This can be built using small sets of axioms that bring everything together to define the unique identity of the user. Because we can define the user without constraints, we define a “pure identity.” Because there are no constraints, the user is not specifically defined for a specific system. When we define a pure identity, we essentially define a set of attributes used for a new digital entity.

We can see from the graph in Figure 9-1 how we create our digital entity. We can see how our identity corresponds to a digital entity for a specific user. When we have identified the user, we can use the same digital identity for different accounts, for example, mail accounts, bank accounts, etc. We can see that our identity is built using different *attributes*. These attributes are used to uniquely identify us. In particular, if we think of an RBA system, attributes give us the code for uniquely identifying the user.



Figure 9-1. How we define and create our digital identity

Note *Risk-based authentication* is a non-static authentication system. With this type of authentication, a system collects specific information about the user and assigns to every piece of information a “weight.” Based on the weight of every attribute, the system makes a decision about whether to allow a user to enter the system. We can see the result of this decision, when, for example, Google or our bank asks us to respond to security questions when we try to log in from another country.

The other functionality of an IAM system is *user access* . User access connects a specific user to his/her digital entity. We use the pure identification to define whether the user can access the system. In the event of access, we create a digital entity, used across the system.

User access involves another functionality of the IAM system: *authorization* . When we talk about authorization, we must think of all the operations, groups, and resources a user is allowed to access in a system.

The last function of an IAM system is *federated identification* . Federated identification is normally referred to by the term single sign-on (SSO). When we think of federated identification, we use our digital entity to access an external resource of our system. This means we use our digital entity without entering a password.

An IAM system is one born to manage the user. This means we can create, delete, and update the identity of the user. These functionalities create an entity and allow the user to execute an operation on the system. The main components and functionality of an IAM system are

- Authentication
- Authorization
- Roles

All these operations and functionality are connected and managed by the IAM. Our digital entity becomes our digital DNA, a unique quality that allows us to move and navigate within the system.

Authentication

Authentication is the first step in the IAM system. Authentication is the phase at which we recognize the entity and verify what/who it claims to be. This phase occurs when the user tries to enter a system, for example, inserting a username and password, and the system authenticates the user. In this phase, we define our digital entity.

Authorization

Authorization is the second step in every IAM system. Authorization is the phase during which we define what the digital entity can do. This is defined in the context of every specific application. For example, the user can be authorized to run the financial report but can't modify it. The authorization is defined by a set of rules. These rules define what our entity is allowed to do.

Roles

Roles are the basis for defining what the user is authorized to do. Roles are a set of groups and operations. The roles are used to manage in a central way the operations users perform. Every entity has some roles attached to it, and this defines the operations or groups the entity has authorized to execute or access.

IAM in Google Cloud Platform

Having a correctly defined IAM in the cloud is essential, if we want to have an effective IAM. This is because of the nature of the application. In the cloud, we have all the user access essentially on the same database and the same shared resource. In this context, having a correct granularity defined for the resource is essential for defining access to it.

GCP offers a very easy and advanced system to manage the IAM. It is possible to have very fine granularity, and this allows an administrator to describe a personalized access for the resource in the cloud and in the Google ecosystem. With GCP, it is possible to define an IAM that lets us manage *who*, in terms of identity, has *what* access, in terms of roles, on *which* resource. Google Cloud IAM allows us to grant access to specific *members*, assigning to every member one or more roles. The assigned member and roles are used to answer the three questions of the IAM:

- Who
- What
- Which

The members of the Google IAM can be one of the following types:

- Google Account

- Service Account
- Google Group
- G Suite Domain
- Cloud Identity Domain

Google Account can be made up of any members with an e-mail address connected with a Google account. It can be an identity associated with google.com or any other domain. This includes a developer, an administrator, or any other person interacting with GCP.

The second members are the *Service Account*. This is essentially an account belonging to specific applications. It is used for calling the API instead of the user. It is possible to create as many service accounts as we require. These are used to logically segment the different applications.

The *Google Group* is a named collection of Google Accounts and Service Accounts. Every group is identified by a unique e-mail. Google Group allows us to create a group and associate more than one user to it. With the group, it is possible for an administrator to associate a role to a number of users at the same time.

The *G Suite Domain* is the grouping for all the Google Accounts. This represents the Internet domain name of an organization. When we create a G Suite Domain, we can create as many users as we need in the domain. This is a convenient way to group users under specific roles.

The last members are the *Cloud Identity Domain*. This represents a virtual group of users. As with the G Suite Domain, we can use this group to easily give access to a group of people.

In addition to the preceding members, we have some special identifiers. These are used to represent the user connected with the Google Account. These identifiers are:

- allAuthenticatedUsers: This is a special identifier and represents anyone who is authenticated by a Google Account or a Service Account. This identifier doesn't represent all the anonymous users or unauthenticated users.

- `allUsers`: This is a special identifier for grouping all users, authenticated or not. Some GCP APIs restrict access to users with the identifier `allUsers`. In this case, the group for the users requires only authenticated users.

When Google IAM must authenticate a user, it indicates what operation the user is allowed to execute, based on a specific set of rules defined. These rules are basically composed of three elements:

- Resource
- Permissions
- Roles

Resource is used to grant access to a specific GCP resource. Examples of these include projects, Compute Engine, and Cloud Storage. Generally, when we assign a permission to a project, all the resources under the project have the same permission. Some resources are different, for example, the Cloud Pub/Sub. In this case, we use a more detailed granularity to assign different permissions to a publisher but not to a subscriber.

An important concept we must keep in mind is *permissions*. This defines what operation a user is allowed to execute in a specific resource. In the Cloud IAM, we can define a permission using the following syntax:

`<service>.<resource>.<verb>`

For example, `pubsub.subscription.consume`, when we define a permission, is normally, but not always, connected with a 1:1 REST method. This means that, normally, every GCP service is exposed to a REST service that is itself exposed.

The last element defined in the GCP IAM is *roles*. A role is a collection of permissions. In GCP IAM, we can assign a permission directly to a user, but we can assign a role to a specific user and, with that, define all the sets of permissions assigned to the role.

In Google IAM, there are three primary types of roles:

- Primitive roles
- Predefined roles

- Custom roles

Primitive roles are the historical roles present in GCP. They are

- Owner
- Editor
- Viewer

These roles existed before the introduction of the Google Cloud IAM and are concentric. The owner roles include the roles of editor and viewer. The primitive roles and the permissions associated with them are summarized in Table 9-1 .

Table 9-1. *The three types of roles in IAM*

Role Name	Role Title	Permission
roles/viewer	Viewer	This role allows the user to read only actions but doesn't modify the viewer existing data.
roles/editor	Editor	This role has all Viewer permissions, plus the permission to modify editor the state of the data.
roles/owner	Owner	This role also has the Editor permission. In addition, it can execute

the following:

- Manage roles and permission
- Set up the billing project

Another family of roles is *predefined roles* . These expand the primitive roles and allow us to create finer granularity for the specific service functionality. With predefined roles, we can assign a user-specific role to a specific application. A user can be assigned more than one role. Predefined roles allow us to create roles for all projects in GCP, for example:

- Project
- App Engine
- BigQuery
- Cloud BigTable
- Cloud Billing
- Cloud Dataflow

- Cloud Dataproc
- Cloud Datastore
- Dialogflow

For every project, we can define a set of specific roles. This allows, for example, a user to have the admin roles for the Kubernetes Engine and the Viewer roles for the Compute Engine.

The last type of role we can define on GCP IAM is *custom roles* . This includes all the roles we define, tailored to our system. When we create a custom role, we can mix one or more roles or get a specific permission from one role and mix it with another. When we create a custom role, we define a new specific role that is designed to satisfy our governance needs.

Note Custom roles are very powerful but require attention. First, we must be sure that the roles we want to create can be applied to the object we want to associate with them. Not all objects can have the same types of permissions. This is because not all objects are the same. You will see how to create a custom role later in the chapter.

Google IAM Policy

First, to learn how to create the roles in GCP, we must understand how the IAM policy works. To create a new role, we must articulate an *IAM policy* . This is a collection of statements used to define who has what type of access. A policy is connected to a resource and is used to enforce control of the access to the resource.

The graph in Figure 9-2 shows the structure of the Google IAM. Every IAM in Google is represented by the IAM policy object. This object is created by a list of bindings. Every bind is a list of members and their associated roles. An example of an IAM policy is shown in Listing 9-1 .

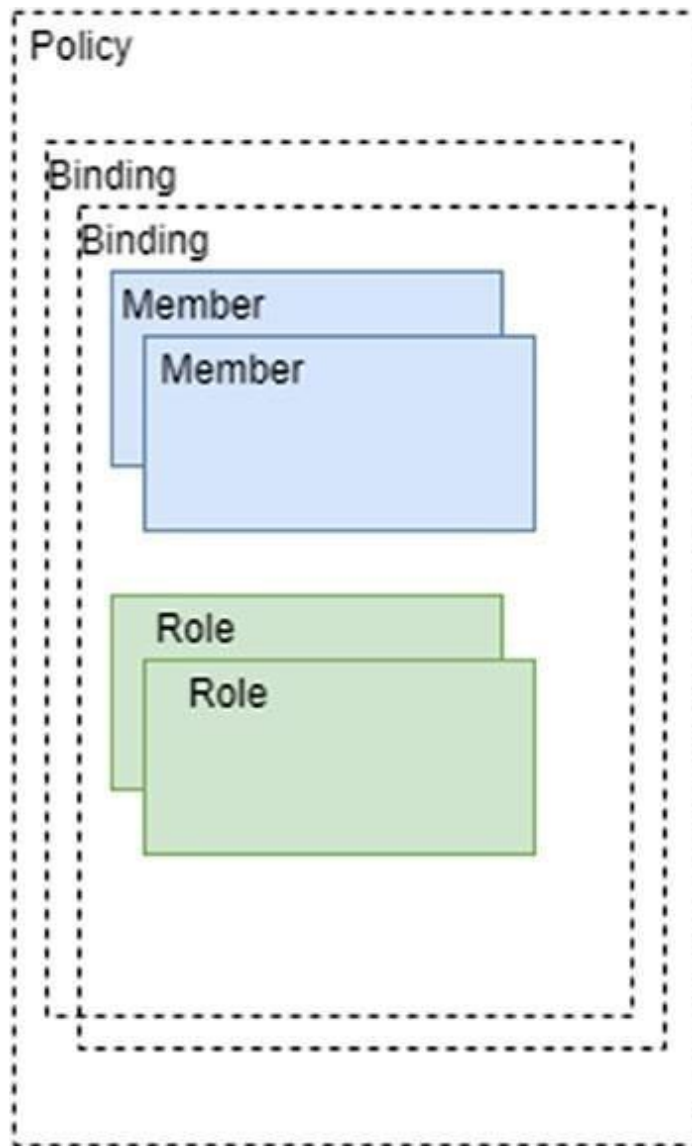


Figure 9-2. The Google IAM

structure

Listing 9-1. A Sample of an IAM Role

```
{
  "bindings": [

    {
      "role": "roles/storage.objectAdmin",
      "members": [
        "user:cgpdevops@devops.com",

        "serviceAccount:my-other-app@appspot.gserviceaccount.com",
        "group:admin@devops.com",
```

```
"domain:gcpdevops.com" ]
},
{ "role": "roles/storage.objectViewer", "members":
["user:test@devops.com"] }
]
}
```

The code to create a new IAM policy is very simple. First, we define the role: "role": "roles/storage.objectAdmin". The format for the role is roles/<name of the role>. When we have to define which role we want to use, we can define the member with which the role is associated.

```
"members": [
"user:cgpdevops@devops.com",

"serviceAccount:my-other-app@appspot.gserviceaccount.com",
"group:admin@devops.com",
"domain:gcpdevops.com" ]
```

This section associates a Google member to the role. In the preceding code, we associate all four members with the same role. This is because we can associate one or more members to one or more roles.

Creating and Managing the IAM Policy

The goal of an IAM policy is to allow a user to have a certain level of access in the project. Until now, I have discussed the general concept of an IAM policy. It's now time to learn how to create an IAM policy to manage the security of our cloud. The IAM policy is used to grant access by a specific user to a specific resource. The IAM policy can be created in two ways: using the GCP console and sending the JSON file with the policy or by the `setIamPolicy()` method.

A common pattern used to create the policy is to *read* the actual policy and then *update* the policy. This can create a problem. Imagine, for example, the following basic scenario. Two users must update the IAM policy to an object

at the same time. User1 sends the request, and with a little delay, User2 sends the same request. User1 receives the policy in return and then updates the policy. User2 has the older policy and then decides to update the IAM policy, which overrides the policy set by User1.

This scenario is more common than you might think. To solve this problem in Google IAM, we can use etag. This value allows us to change the IAM policy only if set. It is used to prevent the concurrency scenario described previously. Google compares the etag associated with the policy with the new one. If the etag is different, the role is not updated. First, to update the policy, we must collect the actual policy using the `getIamPolicy()` method. This gives in return the policy with the actual etag. We can use the etag we gather to update the policy. In case the previous policy doesn't have the etag, it doesn't send an etag. This is because Google doesn't update the policy.

Note The etag value is very useful when we want to manage roles, but not all roles have an etag. First, to update a policy, we must read the actual policy and, if required, get the etag value, but not all policies have the etag set. In case the policy doesn't have an etag, we can't set an etag for the policy. This is because Google rejects the update. If we want the entire policy to have an etag, we can destroy and re-create the policy with the correct etag value.

We can update the policy in some different ways.

- JSON file via gcloud
- API via REST
- API via Java code
- Console

Now you will see how to create roles using the different means allowed by Google.

Creating a JSON File

The first way to creating a new IAM policy is to use a JSON file to define it. This way is probably the most common and fastest, because we have only to create the JSON file and use the GCP console. Using this approach, we can easily integrate the IaC. We can create our JSON file and store it where we

store the file to define the infrastructure and, with every release we make, release the new IAM policy. For modifying the actual IAM policy, we can connect to the gcloud or Google Console and execute the command for reading the actual policy and write in a JSON file. The command is

`gcloud projects get-iam-policy <your project name> --format json > iam.json` This command creates a new JSON file in our path, with the actual policy inside. We can open the file and read what we actually have. In my case, it is like that in Listing 9-2 .

Listing 9-2. The Actual IAM Policy Connected with the Project {

```
"bindings": [  
  
  {  
    "members": [  
      "serviceAccount:service-152799671751@compute-system.iam.  
gserviceaccount.com"  
    ],  
    "role": "roles/compute.serviceAgent"  
  },  
  {  
    "members": [  
      "serviceAccount:service-152799671751@container-engine-robot.iam.  
gserviceaccount.com"  
    ],  
    "role": "roles/container.serviceAgent"  
  },  
  {  
    "members": [  
      "serviceAccount:152799671751-compute@developer.gserviceaccount.com",  
      "serviceAccount:152799671751@cloudservices.gserviceaccount.com",  
      "serviceAccount:service-152799671751@containerregistry.iam.  
gserviceaccount.com",  
      "serviceAccount:service-152799671751@containerregistry.iam.  
gserviceaccount.com"  
    ],  
    "role": "roles/editor"  
  },  
]
```



```
{
  "members": [
    "user:pierluigi.riti@gmail.com"
  ],
  "role": "roles/owner" }
],
"etag": "BwVt1uqAKmk=", "version": 1
}
```

You can see that the IAM policy defines an etag and the version. The version is read-only. We don't have to update the version. To create a new policy, we open the IAM file with an editor and add new members or remove members to the policy.

If we want to add another user account to be the owner of our GCP, we must create new members and associate them with the role. We must add the following to our IAM code:

```
{
  "members": [

    "user:example1@gmail.com" ],
  "role": "roles/viewer"

}
```

When we add this section, we associate with the mail `example1@gmail.com` the roles of the viewer. We can update our IAM file and then execute it. To execute the file, we must use this command:

```
gcloud projects set-iam-policy <your project name> iam.json
```

The result of the command shows the user added in Google Cloud (Listing 9-3).

Listing 9-3. The Cloud IAM Policy Result

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud projects set-iam-policy
practicaldevopsgcp-197023 iam.json
Updated IAM policy for project [practicaldevopsgcp-197023].
```

bindings:

- members:

- serviceAccount:service-152799671751@compute-system.iam.gserviceaccount.com role: roles/compute.serviceAgent

- members:

- serviceAccount:service-152799671751@container-engine-robot.iam.gserviceaccount.com

role: roles/container.serviceAgent

- members:

- serviceAccount:152799671751-compute@developer.gserviceaccount.com

- serviceAccount:152799671751@cloudservices.gserviceaccount.com

- serviceAccount:service-152799671751@containerregistry.iam.gserviceaccount.com

- user:example.1@gmail.com

role: roles/editor

- members:

- user:pierluigi.riti@gmail.com

role: roles/owner

- members:

- user:example.1@gmail.com

role: roles/viewer

etag: BwVzp_aDURQ=

version: 1

Note With the JSON file, it is not possible to create a user-owner. This is possible only using the *setIamPolicy* method. This is because when we invite a new owner, he or she must accept the invitation in the e-mail and join the Google Cloud account.

In addition to using a JSON file, it is possible to use the command line to add a single member with the role. This is possible using the `gcloud projects add-iam-policy-binding` command. This adds a single user to the binding. The command for that is the following:

```
gcloud projects add-iam-policy-binding practicaldevopsgcp-197023 \
--member user:example2@gmail.com --role roles/editor
```

The command inserts another user with the existing member but doesn't create a new member. To create a new member, we must use an existing feature in the Cloud IAM.

Using the API via REST

Another way to modify the Google IAM is to call the `setIamPolicy`. This is a RESTful web service. We must invoke it and send the body to update the IAM. The end point for the REST call is

`https://cloudresourcemanager.googleapis.com/v1/projects/<add your project name>:setIamPolicy`

We must create the body for the request. In this case, we must create a policy, because by setting a new policy, the format is similar to the previous one used for the JSON (Listing 9-4).

Listing 9-4. The `setIamPolicy` Body

```
{
  "policy": {

    "bindings": [ {
      "role": "roles/owner", "members": [

        "user:email1@gmail.com", "user:email2@gmail.com",
        "user:email3@gmail.com"

      ]
    },
    {

      "role": "roles/editor",
      "members": [
        "serviceAccount:example1app@appspot.gserviceaccount.com" ]
    }
  ]
}
```

When we send the request, Google IAM parses it and creates a specific response. The result of the response is shown in Listing 9-5 .

Listing 9-5. The Response from the Google IAM API { "bindings":[

```
{
  "role":"roles/owner", "members":[
    "user:email1@gmail.com", "user:email2@gmail.com",
    "user:email3@gmail.com"
  ],
  {
    "role":"roles/editor",
    "members":[
      "serviceAccount:my-other-app@appspot.gserviceaccount.com" ]
    }
  ]
}
```

Note When we use the API, we must be sure to integrate the OAuth 2.0 in our system. This is because in another case, the request was rejected. This is another layer of security managed by Google to improve security related to the cloud.

Creating the API via Java Code

GCP offers an API to create an IAM. This library is used in our Java project and allows us to create or modify an IAM policy. The library is

```
import com.google.api.services.cloudresourcemanager.model.Policy; import
com.google.api.services.cloudresourcemanager.model.SetIamPolicyRequest;
import
com.google.api.services.cloudresourcemanager.model.GetIamPolicyRequest
; import com.google.api.services.cloudresourcemanager.model.Binding;
```

This object allows us to create the IAM policy and the binding necessary to manage it. Following is a snippet of the code we can create using this object

(Listing 9-6).

Listing 9-6. Code Snippet for Using the Java Code

```
import com.google.api.services.cloudresourcemanager.model.Policy; import
com.google.api.services.cloudresourcemanager.model.
SetIamPolicyRequest;
import com.google.api.services.cloudresourcemanager.model.
GetIamPolicyRequest;
import com.google.api.services.cloudresourcemanager.model.Binding;
import java.util.LinkedList;
import java.util.Arrays;
```

```
...
```

```
String[] myViewers = new String[] {"user:testviewer1@gmail.com",
"user:testviewer2@gmail.com"};
// The name of the role, using the format `roles/<role-name>`. String
targetRole = "roles/viewer";
```

```
Policy policy =
client.projects().getIamPolicy(projectId,
new GetIamPolicyRequest()).execute();
```

```
Binding targetBinding = null;
// Make a local copy of the bindings for modification.
```

```
LinkedList<Binding> bindings =
new LinkedList<Binding>(policy.getBindings());
// Search for the existing binding having role name of
// targetRole.
for (Binding binding : bindings) {

    if (binding.getRole().equals(targetRole)) {
        targetBinding = binding;
        break;
    }
}
```

```
// If no matching targetBinding is found, construct a new Binding object //
and add it to the bindings list.
if (targetBinding == null) {

    targetBinding = new Binding();
    targetBinding.setRole(targetRole);
    bindings.add(targetBinding);
}

// Finally, set the list of members as the members of targetBinding.
targetBinding.setMembers(Arrays.asList(myViewers));
// Write the policy back into the project by calling SetIamPolicy.

SetIamPolicyRequest setIamPolicyRequest = new SetIamPolicyRequest();
setIamPolicyRequest.setPolicy(policy);
client.projects().setIamPolicy(projectId,
setIamPolicyRequest).execute();
...
```

Using the Console

The last way to create a new policy is to use the GCP Console. This is the simplest way to manage the members and policy.

To manage the role and policy via the console, follow these simple steps: 1. Open the Google Console and select IAM & admin from the menu (see [Figure 9-3](#)).

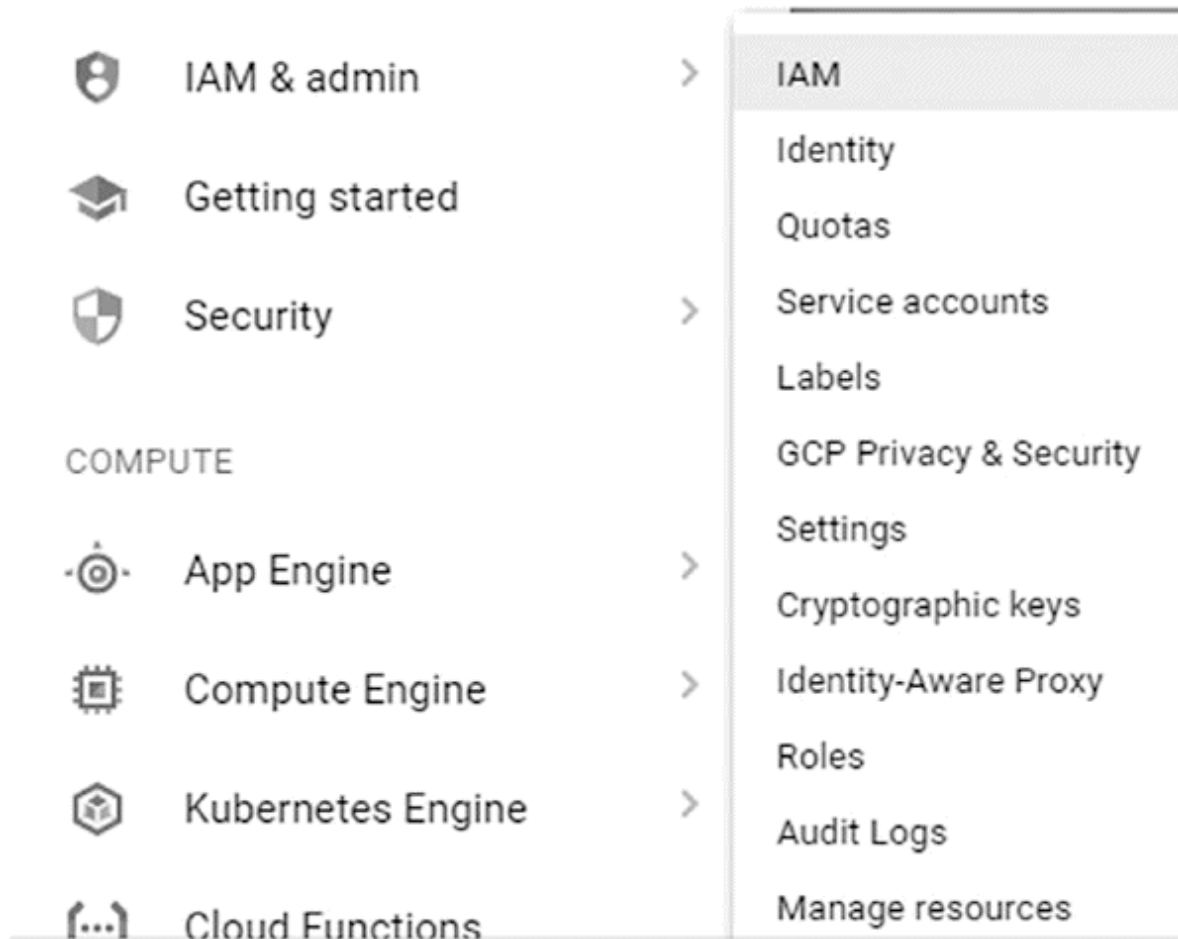


Figure 9-3. *The IAM menu on the GCP Console*

2. Then select IAM. This opens the IAM board, from which it is possible to manage the Google IAM. From the board, it is possible to add or remove members or roles, manage identity, etc. (Figure 9-4).

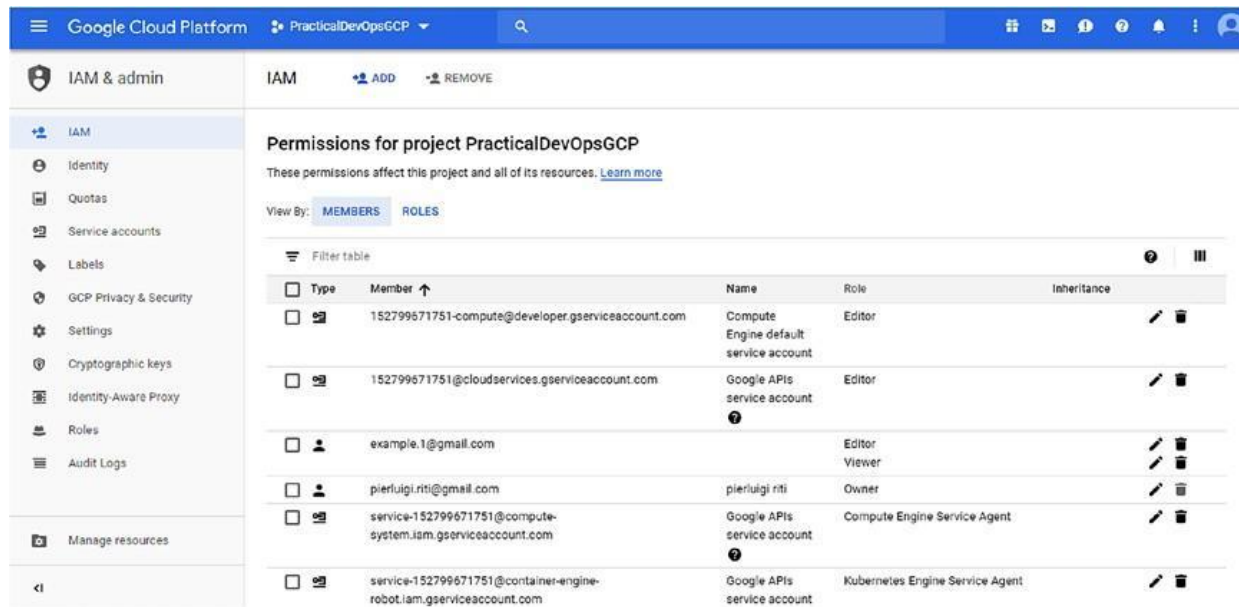


Figure 9-4. The IAM permission board

The board allows us to create and maintain all the roles in the cloud. We can navigate the roles, permissions, and identity in a simple way. Using the board allows us to manage the role and the other entities involved in the IAM policy efficiently.

Conclusion

This chapter discussed the Google IAM policy. First, I introduced what IAM is and why it is important in a cloud environment. After I presented different ways to create the IAM in GCP. Google IAM offers a very flexible and powerful tool for managing the security of an entire system and the flexibility to initiate governance based on a specific project level. We can easily design our governance model and create and maintain it via a JSON file. This allows us to improve our IaC system and integrate it with our security policy. Combining the Google IAM policy with IaC can drastically reduce the security issues in the system, which means more security and fewer SOC security exceptions.

CHAPTER 10

Network Configuration and Management in GCP

When we refer to a cloud environment, we are talking about resources shared over a network. These resources must be configured and managed over a network, so we must ensure the level of security required to expose and maintain them. When we create a cloud application, we must think of having the application up 24/7. This is possible only if we design the architecture to function reliably on the network. In this chapter, you will see how we can design and maintain good network architecture in Google Cloud Platform (GCP).

Fundamentals of Networking in GCP

First, to define and manage our network, we must identify and understand how the network is organized in GCP. In GCP, the network is divided into two main components:

- Regions
- Zones

A region is a geographical zone in which we run our resources. One region may contain one or more zones. A resource that lives in a zone, such as a persistent disk, is called a *zonal resource* . Other resources, such as an external Internet Protocol (IP), is called a *regional resource* . A regional resource can be used by every resource in the same zone. Not all resources are zonal or regional. GCP’s regional and zonal resources are listed in Table 10-1 .

Table 10-1. Regional and Zonal Resources in GCP

	Regional	Regional	Addresses	Subnets
--	----------	----------	-----------	---------

Regional				
----------	--	--	--	--

Regional Regional Managed Instance Groups

Regional Persistent Disks

Zonal Instances

Zonal Persistent Disks

Zonal Machine Types A collection of any static external IP defined in the project A regional subnet is used to split addresses in the segment. It is used to identify the subnetwork in the primary network. These resources are collections of homogeneous resources spanning multiple zones in the same regions.

This are used to provide durable storage and replication of data across different zones. In case of failover, we can attach the resource to another region, to migrate the data to the cloud. Every instance must live in a zone and can access the global resource or resources in the same zone.

A persistent disk can be accessed by all the resources in the same zone. It is possible to attach the disk only to a live resource in the same zone.

These are pre-zone resources. An instance and disk can only use machine types in the same zones.

In thinking about our architecture, we must define where to place our resource, choosing our zone and resource carefully. This is because correct planning can help to achieve two major goals for the network.

- *Handling failures* : If we distribute the resource across zones and regions, we can use Google Cloud to design the zones to be independent of one another in the event of failure. Every zone will be completely independent, and a failure in one zone will not cause a failure in the other zones. When we have a failure in one zone, we can move the data to another zone and reach the high-availability of the cloud.

- *Decreased network latency* : To reduce the latency of the network, we must place the resource in the location nearest to us. This means that if we want to have our application to be global, we have to reproduce the resource in different regions.

In addition to zonal and regional resources, we have some *global* resources. This type of resource can be used globally across the cloud. Global resources include

- Images
- Snapshots
- VPC Network
- Firewalls
- Routes

Global resources are important because they can be used in every zone and every region. The basic requirement for creating a network is a virtual private cloud (VPC). This is the basic resource for exposing and creating the network architecture.

Introduction to the Virtual Private Cloud

A VPC gives us the flexibility to scale and control the workload of the resource on global and regional levels. When we connect our resource to GCP, we can have access to our VPC without having to replicate the connectivity in every region. This is because the VPC does that for us. A single Google VPC doesn't have to access the Internet to share the resource across different regions. A single connection point can share the resource across regions. This reduces the complexity of our organization.

In Figure [10-1](#) , you can see that with the Google VPC, we have different subnets. Every subnet has an application with the internal IP. The VPC shares the IP with the outside world, thereby sharing the network with the outside world as well.

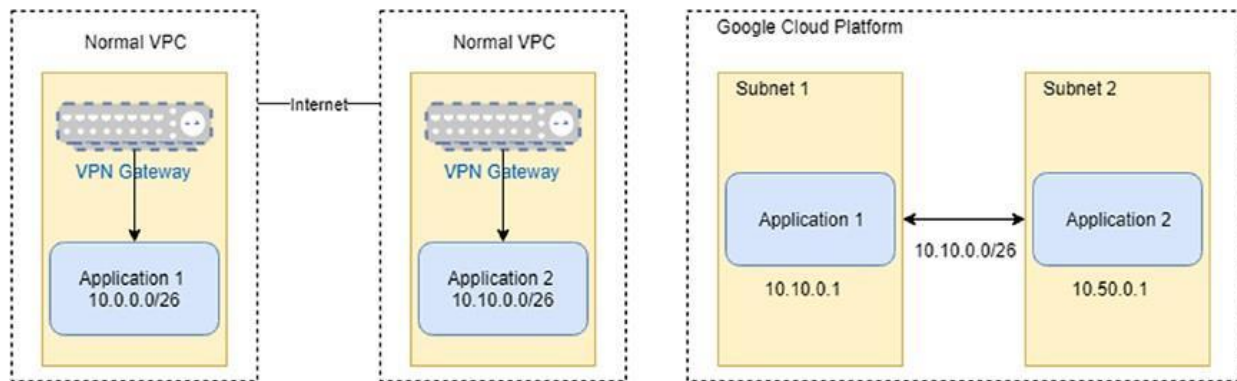


Figure 10-1. The difference between a normal VPC and Google Cloud Platform VPC

Another important attribute of a VPC is its ability to isolate a project. With a VPC, we can easily separate projects and have different billing for each project. However, we can share information via the *cloud interconnect* component. The VPC network is normally referred to simply as the “network.” Every time we create a new project, it is associated with a *default* network. This is essentially a virtual representation of a physical network and provides the connectivity to all other components, such as Compute Engine or Kubernetes Engine. Every VPC has some specific properties.

- The VPC network and all its related components, such as firewall rules and routes, are global. They are not associated with a specific zone or region.
- Subnets are a regional resource. Each subnet defines a range of IP addresses.
- Traffic to and from different instances can be controlled by the firewall rules.
- The resource can communicate across different subnets, using the internal private IPv4 addresses. The communication respects the firewall rules defined for the VPC.
- If we define a private IP for an instance, it can communicate with the Google API only if we enable the Private Google Access for the subnet in which the instance resides.
- It is possible to use the IAM roles to define who can administrate the network.

- A network can be connected to another VPC using *VPC Network Peering*.
- A VPC Network can be securely connected in a hybrid environment, using Cloud VPN or Google Cloud Interconnect.
- A VPC network supports only IPv4 unicast traffic. It doesn't support multicast and broadcast or IPv6 traffic. It is possible to use the IPv6 to reach resources in the network.
- A VPC network has a maximum transmission unit (MTU) of 1460 bytes. This MTU value is offered to the instance via DHCP option 26, per RFC 2132. Configuring the MTU for packets larger than 1460 bytes can lead to packet loss.

These properties help to understand how a VPC works and what its limitations are. Now that we know that the VPC allows subnets to talk to each other, it is important to understand how the subnets work in GCP.

Network and Subnets

Every VPC network consists of one or more IP ranges. These IP ranges are called subnetworks, or subnets. Every subnet is associated with a region. A VPC network can have one or more subnets in one or more regions. This means that we can have our VPC network span different geographical locations, to ensure the high availability of the cloud. This means that, in case of failure in one region, another region is available, and we can ensure our customers of the cloud's functionality. We can define the VPC network in two ways:

- *Auto mode* : The network is created automatically across different regions.
- *Custom mode* : The network is created without any associated subnet, which makes it possible for us to define all the characteristics of our network.

When we create an auto mode network, GCP creates a subnet for every region. The subnet is associated with a predefined set of IP ranges in the 0.128.0.0/9 CIDR (classless inter-domain routing) block. When a new GCP region becomes available, a new subnet is added to this region, and the

network uses the IP network associated with the region. In addition to adding a subnet by GCP, it is possible to add a subnet to a region manually and use an IP outside of the automatic subnet. When we create a custom mode network, GCP doesn't provide a default subnet. This type of network leaves it to users to determine the power to define their network. We can use every IP range we want and create the subnet in every region we decide to use.

When we plan our network, we must consider the strengths and weaknesses of every type of network. This is useful for understanding what type of network is best suited to our architecture. The auto mode network is easy to set up and use. It lends itself best to the following use cases:

- For a subnet that is automatically created in every region
- When the predefined IP ranges don't overlap with any IP addresses we have configured for our architecture

On the other hand, the custom mode network offers more flexibility and better suits a production environment. This is because we can define our ranges of IP addresses and we plan more carefully where to create the subnet network. We can identify some basic use cases in which the custom mode network is highly recommended.

- When we don't have to create a subnet network automatically in every region
- When we want to decide the range of the IP assigned to our subnet, because the predefined IP ranges overlap with other IPs on the network
- When we plan to connect our subnet using VPC Network, Peering, or Cloud VPC.

Defining the IP Range for the Subnet

When we create a custom mode network, we must design our IP ranges according to certain rules. These rules are used to help us correctly implement the IP ranges for our subnets.

- A subnet must have a *primary address range* . This range must be a valid RFC 1918 CIDR block.
- Subnets in the same network must have a unique IP range, but subnets in a different network, even if part of the same project, can use the same IP range and reuse it for another network.
- When we define an RFC 1918 CIDR block, we must impose some limitations on the IP ranges we can create.
- Subnets in the same network must have a unique IP address.
- When we connect VPC using VPC Network Peering or Cloud VPC, the subnet must have a unique IP address.
- When we define IP ranges for an on-premise network connected via Cloud VPN or Google Cloud Interconnect, they should not conflict with any other IP range defined in the GCP.
- It is possible to assign one or more IP address ranges to a subnet. These ranges are reserved for the Virtual Machine (VM) instances. These can use the RFC 1918 CIDR block and must respect the same rules defined previously.
- We don't have to define a contiguous IP range for the subnet we define in the same network.
- The minimum size for the CIDR for a subnet is /29.

Note CIDR, sometimes called supernetting, is a way to allow more flexibility in allocating an Internet Protocol (IP). This slows down exhaustion of the IPv4 address. CIDR introduces the concept of the *variable-length subnet masking (VLSM)* . This technique allows us to specify an arbitrary length for the prefix. CIDR introduces a new method to represent the IP address, called *CIDR notation* . With CIDR notation, the address or routing prefix is written with a suffix indicating the number of bits of the prefix, for example, 192.10.0.2/24 for the IPv4.

When we define a new subnet, we must be sure to have four IP addresses reserved on its primary IP range. These reserved addresses are listed in Table 10-2 .

Table 10-2. Reserved IP Addresses for Subnets

Reserved Address	Network
------------------	---------

Default Gateway

Second-to-Last Reservation Broadcast

Description Example First IP address in the primary IP range of the subnet 10.10.1.0 in 10.10.1.0/24 Second IP address in the primary IP range of the subnet 10.10.1.1 in 10.10.1.0/24 Second-to-last IP address in the primary IP range of the 10.10.1.254 in subnet 10.10.1.0/24 Last IP address in the primary IP range of the subnet 10.10.1.255 in 10.10.1.0/24

The reserved IP addresses exist only in the primary IP address. This means that when we define a secondary IP address, we don't have a reserved address in the secondary address.

Routes and Firewalls

Routes and firewalls are some of the basic components of every network. A *route* is a defined pair of addresses that represent the *destination* and the *gateway*. A route is used when we try to send a packet to a specific destination, using a specific gateway. We can identify three types of destinations.

- Individual hosts
- Subnets
- Default

There are also three types of gateways.

- Individual hosts
- Interface or links
- Ethernet hardware address (MAC)

When we create a VPC GCP to generate a *system-generated route* automatically, this route is created and managed within it, as follows:

- GCP creates a route whose destination is 0.0.0.0/0 and whose next hop is the default Internet gateway.

- For each subnet we create, a route is created to define the path for its resource. These subnet routes show the next hop of the virtual network in the GCP Console. The permit for the communication is defined by the *firewall rules*.

- The priority of a subnet's route is fixed at 1000.
- An auto mode network automatically creates a route for every subnet we create.
- For every subnet we create, a subnet route is automatically created.
- When we delete a subnet, all the routes are automatically deleted.

We can identify the route generated by the system, because the name begins with the default-route prefix. This helps to identify what routes are added manually and those generated by the system. For example, if we open GCP and move to the Route section, we can see something such as Figure 10-2 .

<input type="checkbox"/> Name ^	Destination IP ranges	Priority	Instance tags	Next hop	Network
<input type="checkbox"/> default-route-1d7c378c35a58bf1	10.142.0.0/20	1000	None	VPC network	default
<input type="checkbox"/> default-route-2733f6b435adfb8	10.146.0.0/20	1000	None	VPC network	default
<input type="checkbox"/> default-route-27ee2255201b7609	10.152.0.0/20	1000	None	VPC network	default

Figure 10-2. Example of a default route

As with the default rules applied to routes, in GCP, we can find some generic firewall rules. These rules allow us to define what traffic is allowed or denied to and from a VM.

These rules are always used, and the traffic is controlled even if we stay in our network. This means that when we move from a private internal VM to another private internal VM, the firewall applies the rules we previously defined. Every VPC has two implicit firewall rules. These are not shown in the GCP Console but are used and defined when we create a new VPC.

These rules are:

- *Implied allow egress rule* : This rule allows *egress* traffic. Incoming traffic to the destination 0.0.0.0/0 is allowed, and the priority is the lowest possible, 65535.
- *Implied deny ingress rule* : This rule denies *ingress* traffic, if the source is 0.0.0.0/0 and the priority is the lowest possible, 65535.

These implied rules cannot be removed, but they can be overridden. To override the rules, we must create a new rule with a higher priority, with a priority number less than 65535. This is because both implied rules have a

priority of 65535. The default network adds some additional default rules, which can be modified or deleted, as necessary. These rules are:

- default-allow-internal: This rule allows ingress connections for all protocols and ports among the instances in the network. The priority for this rule is 65534.
- default-allow-sh: This rule allows the connection on TCP with the port 22, from any source or instance to the network. The priority for this rule is 65534.
- default-allow-rdp: This rule allows the connection on TCP with the port 3389. This rule has a priority of 65534 and is used to allow the Windows Remote Desktop.
- default-allow-icmp: This rule allows the ICMP traffic for any source or instance on the network. The priority of this rule is 65534.²⁵³

In GCP, some traffic is blocked by default. It is not possible to create firewall rules to allow this traffic. The blocked traffic is

- GRE traffic
- Protocols other than TCP, UDP, ICMP, IP
- Egress traffic on the TCP and port 25
- Egress traffic on the TCP and port 465 or 587 (SMTP over SSL/TLS)

Having a basic knowledge of routes and firewalls is important for understanding how to design and implement a VPC in GCP.

Tagging Network

One very useful feature is network tagging. A network tag is an attribute we can add in our Compute Engine VM and use with firewalls and routes, applied only on some specific Compute Engine VM. Tags don't need to be unique across multiple VPC networks.

When we create a tag for our network, we can apply firewall rules or routes to a specific set of instances. With a tag, we can

- Apply the firewall rule applicable to a specific instance, using target tags and source tags
- Apply the routes to a specific instance, by using the tag

In GCP, every firewall has a target. By default, all firewalls have the target all instances of the network. We can specify the instances as targets, using the target tags or target service accounts. A *target tag* is used to identify the GCP VM to which the rules are applied. The rule is applied to the primary internal IP address and to all the instances that have the same network tag.

To create a tag in our network, we must to use this command:

```
gcloud compute instances add-tags <INSTANCE-NAME> --zone <ZONE>
--tags <TAGS>
```

This adds a new tag to the instance, which means existing tags are not deleted or modified by the command. The result of the command is shown in Listing 10-1 .

Listing 10-1. Result of the add-tags Command

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute instances
add-tags test-tag --zone us-east1-b --tags test-tag
Updated [https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/zones/us-east1-b/instances/test-tag].
```

We can remove the tag, using the command `remove-tags`. The syntax is

```
gcloud compute instances remove-tags <INSTANCE-NAME> --zone
<ZONE>
--tags <TAGS>
```

The tag is very useful when we want to apply the firewall rules or routes to a specific set of VM instances. This allows us to have a “logical” representation of the network, and we can easily identify what rules are applied to the specific VM.

Implementing a VPC Network

Up to now, I have presented the theory and the basis for a VPC network, but to understand these better, you must get your hands dirty. So, let’s start to create a VPC network for our Google Cloud Platform. The simple way to

create a VPC is to create an auto mode network via the GCP Console. We can create a VPC with these simple steps:

1. Connect to the GCP Console.
2. Move to the VPC networks page (Figure 10-3).

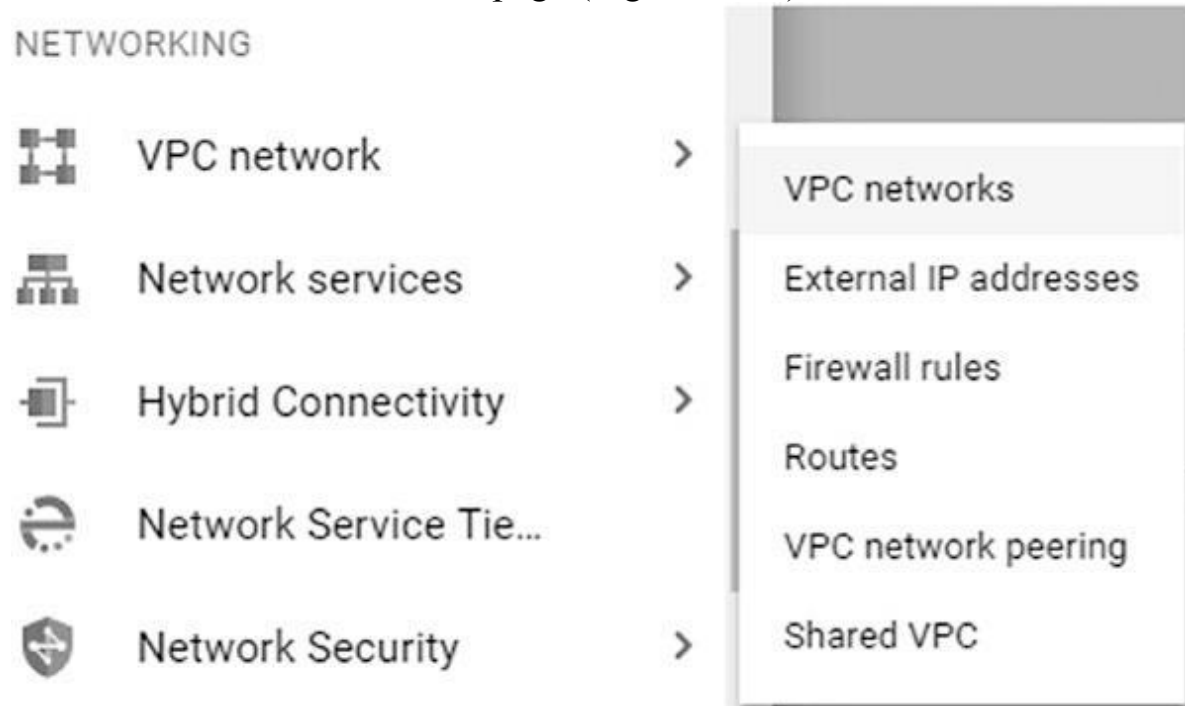


Figure 10-3. The networking section for creating a new VPC network

3. This opens the page from which it is possible to find all the VPC networks and create a new one. To create a new network, just click the CREATE VPC NETWORK button (Figure 10-4).

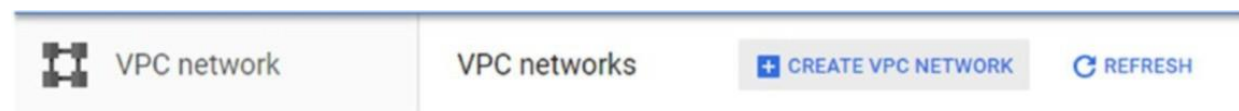



Figure 10-4. The button to create a new VPC network

The page for creating the VPC network is now open. On this page, we can add and indicate all the information we need to create our VPC. First, in the appropriate field, we must indicate the *name* of the network. This must be in all lowercase letters and without spaces. We can use numbers or hyphens in the name of the network. With the name, we can add a description of the network. This can be used to indicate the scope of the network. When we

create the name and add the description, we must create the subnet to connect with the VPC. We can create the subnet in two ways:

- Automatic
- Custom

If we choose an *automatic* mode to create the VPN, we have only to select the region for our subnet, and the GCP assigns the default IP address range for the region (Figure [10-5](#)).

 These IP address ranges will be assigned to each region in your VPC network. When an instance is created for your VPC network, it will be assigned an IP from the appropriate region's address range.

<< Previous

1

2

Next >>

Region	IP address range
us-central1	10.128.0.0/20
eu-west-1	10.132.0.0/20
us-west-1	10.138.0.0/20
asia-east-1	10.140.0.0/20
us-east-1	10.142.0.0/20
asia-northeast-1	10.146.0.0/20
asia-southeast-1	10.148.0.0/20
us-east-4	10.150.0.0/20
australia-southeast-1	10.152.0.0/20
eu-west-2	10.154.0.0/20

<< Previous

1

2

Next >>

Figure 10-5. The automatic subnet creation options

When we opt to create *custom* subnets, we must indicate more values for the subnet. We can indicate the name of the network, respecting the same rules as for the VPC. The name must be entirely in lowercase letters, with only letters, numbers, or hyphens, and without spaces. It is possible to add a

description for the subnet. This is useful for indicating the type and the use of the subnet.

The other parameter we need to specify for a subnet is the *region* where we want to create the VPC. The region is the classic region we can use in Google Cloud. In the *IP address range* section, we can identify the private IP range for our VPC. It is always possible to add a second IP range. In this case, we can indicate a name and the IP range we want to use. The last two options for our subnet are

- *Private Google access* : This option indicates if our VPC can access

other services without assigning an external IP.

- *Flow logs* : This option is used to generate the log of the subnet. It doesn't slow down any service but generates Stackdriver traffic and can improve the cost of the service (Figure [10-6](#)).

New subnet

Name ?

lowercase, no spaces

Add a description

Region ?

IP address range ?

Example: 10.0.0.0/9

Create secondary IP range

Private Google access ?

☐ On

☒ Off

Flow logs

☐ On

☒ Off

Done

Cancel

+ Add subnet

Figure 10-6. The option for generating the custom subnet

The last option is the *dynamic routing mode*. This option is used to determine which subnets are visible in the cloud routers. We can set two types of routing:

- Regional
- Global

If we select *regional* routing, Google Cloud Router advertises and propagates our subnet in the region where the subnet is defined. With *global* routing, Cloud Router advertises all the subnets defined in our VPC to the on-premises network. Cloud Router propagates the learned route across all the regions of our VPC. With all the options set, we can click the Create button, and this creates the VPC. We can see the created network in our Google Console (Figure 10-7).

practicaldevopsgcp	17	Auto ▾	0	Off
us-central1	practicaldevopsgcp	10.128.0.0/20	10.128.0.1	Off
europa-west1	practicaldevopsgcp	10.132.0.0/20	10.132.0.1	Off
us-west1	practicaldevopsgcp	10.138.0.0/20	10.138.0.1	Off
asia-east1	practicaldevopsgcp	10.140.0.0/20	10.140.0.1	Off
us-east1	practicaldevopsgcp	10.142.0.0/20	10.142.0.1	Off
asia-northeast1	practicaldevopsgcp	10.146.0.0/20	10.146.0.1	Off
asia-southeast1	practicaldevopsgcp	10.148.0.0/20	10.148.0.1	Off
us-east4	practicaldevopsgcp	10.150.0.0/20	10.150.0.1	Off
australia-southeast1	practicaldevopsgcp	10.152.0.0/20	10.152.0.1	Off
europa-west2	practicaldevopsgcp	10.154.0.0/20	10.154.0.1	Off
europa-west3	practicaldevopsgcp	10.156.0.0/20	10.156.0.1	Off

Figure 10-7. The Google Console displaying the information about the network we just created

Another way to create a VPC is to use the gcloud command line. To create a network and a subnet, we must perform two different steps. First, we must create the network and then the subnet. To create the network, we must use a command with the following format:

```
gcloud compute --project=<project name> networks create <network name>
--subnet-mode=auto/custom
```

In case we want to create a custom mode network, we must add a subnet. The command for creating the subnet is

```
gcloud compute --project=<project name> networks subnets create <subnet
name> --network=<network name> --region=<region name> --range=<ip
range>
```

```
--secondary-range=<secondary range name>=<ip range> --bgp-routing-mode <routing mode>
```

The routing mode section is optional and used to indicate what type of routing mode we want to use. We can now try the command to create the network. The first step is to create the network. The complete command for this is

```
gcloud compute --project=<your project name> networks create practicaldevopsgcp --subnet-mode=auto
```

The result of the command shows the information about the network we just wanted to create. We can see the result from the command in Listing 10-2 .

Listing 10-2. The Result of the Command Line Network Creation

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute --project=practicaldevopsgcp-197023 networks create practicaldevopsgcp --subnet-mode=auto
```

```
Created [https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp197023/global/networks/practicaldevopsgcp]. NAME SUBNET_MODE BGP_ROUTING_MODE IPV4_RANGE GATEWAY_IPV4 practicaldevopsgcp AUTO REGIONAL
```

Instances on this network will not be reachable until firewall rules are created. As an example, you can allow all internal traffic between instances as well as SSH, RDP, and ICMP by running:

```
$ gcloud compute firewall-rules create <FIREWALL_NAME> --network practicaldevopsgcp --allow tcp,udp,icmp --source-ranges <IP_RANGE> $ gcloud compute firewall-rules create <FIREWALL_NAME> --network practicaldevopsgcp --allow tcp:22,tcp:3389,icmp
```

In the Google Console, it is possible to see the new network created. Because we have created the network in auto mode, the console shows an IP address for any region. Global routing is disabled, because we don't specify any routing for the network, and, of course, we don't have any specific firewall rules. If we want to create our subnet network, we must create the network with a custom mode. The command for doing this is

```
gcloud compute --project=<your project name> networks create practicaldevopsgcp-custom --subnet-mode=custom
```

The result for the command shows the creation of the network and the command for creating the firewall rules (see Listing 10-3).

Listing 10-3. The Result of the Command to Create the Network in Custom Mode

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute
--project=practicaldevopsgcp-197023 networks create
practicaldevopsgcpcustom --subnet-mode=custom
Created [https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/global/networks/practicaldevopsgcp-custom].
NAME SUBNET_MODE BGP_ROUTING_MODE IPV4_RANGE
GATEWAY_IPV4 practicaldevopsgcp-custom CUSTOM REGIONAL
Instances on this network will not be reachable until firewall rules are
created. As an example, you can allow all internal traffic between instances
as well as SSH, RDP, and ICMP by running:
$ gcloud compute firewall-rules create <FIREWALL_NAME> --network
practicaldevopsgcp-custom --allow tcp,udp,icmp --source-ranges
<IP_RANGE> $ gcloud compute firewall-rules create
<FIREWALL_NAME> --network practicaldevopsgcp-custom --allow
tcp:22,tcp:3389,icmp
```

The network now shows up on the Google Cloud Console, but because we don't add any subnet on the network, this is just a name created in custom mode (Figure 10-8).

Name	Region	Subnets	Mode	IP addresses ranges	Gateways	Firewall Rules	Global dynamic routing	Flow logs
practicaldevopsgcp-custom		0	Custom			0	Off	

Figure 10-8. The custom network created without subnet

The next step is to create the subnet associated with the network. The command for creating the subnet associated with the network is

```
gcloud compute --project=<you project name> networks subnets create
practicaldevopsgcp-subnet --network=practicaldevopsgcp-custom
--region=us-central1 --range=10.0.0.0/24
```

The parameter for creating the network requires first the name of the subnet. We associate the new subnet of the network with the parameter--network. The result of the command is shown in Listing 10-4 .

Listing 10-4. The Result of the Subnet Associated with the Network

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute
--project=practicaldevopsgcp-197023 networks subnets create
practicaldevopsgcp-subnet --network=practicaldevopsgcp
-custom --region=us-central1 --range=10.0.0.0/24
Created [https://www.googleapis.com/compute/v1/projects/
practicaldevopsgcp-197023/regions/us-central1/subnetworks/
practicaldevopsgcp-subnet].
NAME REGION NETWORK RANGE practicaldevopsgcp-subnet us-
central1 practicaldevopsgcp-custom 10.0.0.0/24
```

This command associates the new subnet to the previous network we created. We can see the subnet associated with the network in the console (Figure 10-9).

Name	Region	Subnets	Mode	IP addresses ranges	Gateways	Firewall Rules	Global dynamic routing	Flow logs
practicaldevopsgcp-custom		1	Custom			0	Off	
	us-central1	practicaldevopsgcp-subnet		10.0.0.0/24	10.0.0.1			Off

Figure 10-9. The subnet associated with the previous network

We have now created a network and associated it with a subnet network. For the custom network, we must create a subnet for every region in which we want to replicate the VPC.

Creating and Maintaining Firewall Rules

You have learned how to create a new VPC, but creating a VPC is not enough. To ensure security of access, we must configure a firewall. A firewall is one of the basic components of the network's security, and the aim of a firewall is to deny or allow traffic to the network, if the traffic doesn't respect some certain rules.

We see, when we create the VPC, that GCP assigns some default firewall rules to our VPC. This is to define what traffic is allowed or denied on the VPC, but these rules are not enough to manage a network. For that, we must create and manage the firewall rules necessary to manage our VPC. Now you will learn how to create new firewall rules with gcloud. The command

for creating a new firewall rule is

gcloud compute firewall-rules create [NAME] \

[--network [NETWORK]; default="default"] \

[--priority [PRIORITY]; default=1000] \

[--direction (ingress|egress|in|out); default="ingress"] \ [--action (deny |
allow)] \

[--target-tags [TAG][, TAG,...]] \

[--target-service-accounts=[IAM Service Account] \

[--source-ranges [CIDR-RANGE][, CIDR-RANGE...]] \

[--source-tags [TAG][, TAG,...]] \

[--source-service-accounts=[IAM Service Account] \

[--destination-ranges [CIDR-RANGE][, CIDR-RANGE...]] \

[--rules ([PROTOCOL][:PORT[-PORT]], [PROTOCOL[:PORT[-PORT]]], ...)]
| all) \

[--disabled | --no-disabled]

All these parameters are used to define the type of firewall rule we want to create. Following are all the parameters and their role in creating the firewall rule.

- **--network:** This parameter indicates the network on which we create the firewall rule. If we don't indicate any network, the rule is created in the default network.
- **--priority:** This is a numerical value indicating the priority of the rule. The lowest level is 65535 and is normally assigned to the default firewall rule. The lower the number, the higher the priority of the rule. A higher number indicates a lower-level priority of the rule.
- **--direction:** This parameter indicates the direction of the rule. Ingress is the direction used to indicate traffic from a source to a target. Egress is the direction indicating the traffic from a target to a destination.
- **--action:** This parameter indicates what action we want to execute with the firewall rule. We can define one of two actions:

- allow: This permits the connection
- deny: This prevents the connection

A firewall rule must be applied to a target. This target can be a network or a service user. The two different values to define the target are

- --target-tags: This is used to indicate the network on which the firewall rule applies.
- --target-service-accounts: This indicates the service account to which this rule applies.

If we don't define any of the target values, the rule simply applies to the entire network. Another important part of the configuration is the direction of the traffic we want to use to define the direction of the rule. For the ingress rule, we must specify a source. We can specify three types of sources.

- --source-ranges: With this flag, we can specify the range of source addresses in CIDR format.
- --source-tags: With this flag, we apply the rule to all the IP addresses coming from a specifically tagged network.
- --source-service-account: With this flag, we can specify all the IP addresses for a service account. We can use this tag only if we don't configure the source tag.

If we don't specify any source, we apply the rule everywhere, because it is applied to the IP 0.0.0.0/0. We can define an egress rule to use the destination. This has only the following option:

- --destination-range: This flag defines the range of the destination IP using the CIDR format.

In case we omit the destination option, the egress destination is applied everywhere to IP 0.0.0.0/0. The --rules option defines to which protocols and ports the rule applies. If we use All, we apply the rule to all the protocols and ports in the VPC network. An example of a firewall rule is

```
gcloud compute --project=<your project name> firewall-rules create test
--direction=INGRESS --priority=1000 --network=practicaldevopsgcp-
```

custom

--action=deny --rules=tcp

The result for the command is the creation of the rule (see Listing 10-5).

Listing 10-5. The Newly Created Firewall Rule

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute
--project=practicaldevopsgcp-197023 firewall-rules create test
--direction=INGRESS --priority=1000 --network=practical
devopsgcp-custom --action=deny --rules=tcp
Creating firewall...Created [https://www.googleapis.com/compute/v1/
projects/practicaldevopsgcp-197023/global/firewalls/test]. Creating
firewall...done.
NAME NETWORK DIRECTION PRIORITY ALLOW DENY test
practicaldevopsgcp-custom INGRESS 1000 tcp
```

In case we want to update the firewall rule, we can use the same basic syntax, only instead of using `firewall-rules create`, we use `firewall-rules update` command.

Creating and Maintaining Routes in GCP

At the start of this chapter, you learned what a route is. Every network has some system-generated routes from a VPC network and to the Internet. We can't override the routes to another subnet in the same network, but we can override the default route 0.0.0.0/0. We can create routes to a specific destination range. This is to direct the traffic to a VPN tunnel, a specific instance, or other destination.

When we plan our network, we want to be sure not to override any existing route. We can list the actual route we have in the network with this simple command: `gcloud compute routes list`

The result of the command shows all the routes actually configured in the system. We can see an extract of the routes in Listing 10-6 .

Listing 10-6. The Route List Result

```
pierluigi_riti@practicaldevopsgcp-197023:~$ gcloud compute routes list
NAME NETWORK DEST_RANGE NEXT_HOP PRIORITY
default-route-1d7c378c35a58bf1 default 10.142.0.0/20
```

```
default 1000 default-route-2733f6b435addfb8 default 10.146.0.0/20 default 1000 default-route-27ee2255201b7609 default 10.152.0.0/20 default 1000 default-route-2ba825e5b5d3dd2a default 10.160.0.0/20 default 1000 default-route-2eca2b974eff32fb default 0.0.0.0/0 default internet-gateway 1000
```

To see the details of an individual route, we can use this command: `gcloud compute routes describe <route name>`

For example, we can see the details for one of the default routes, using this command line:

```
gcloud compute routes describe default-route-9702087d924fe3ad
```

The result is the description of the route.

```
creationTimestamp: '2018-07-29T14:07:03.497-07:00'
description: Default local route to the subnetwork 10.132.0.0/20. destRange: 10.132.0.0/20
id: '1148425788491797144'
kind: compute#route
name: default-route-9702087d924fe3ad
network: https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/networks/practicaldevopsgcp
nextHopNetwork: https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/networks/practicaldevopsgcp priority: 1000
selfLink: https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/routes/default-route-9702087d924fe3ad
```

The description of the route shows us the information connected with the route, for example, the creation of the route, the kind of route, the network associated with the route, and the NextHop for the network.

The most important operation we want to perform with the route is the creation of a new static route. The syntax for creating a new static route is

```
gcloud compute routes create [ROUTE] \
--destination-range [DEST_RANGE] \
--network [NETWORK] \
[--tags=TAG,[TAG,...]] \
[--next-hop-address=[ADDRESS]] |
[--next-hop-gateway=default-internet-gateway] |
```



```
[--next-hop-instance=[INSTANCE_NAME] |  
--next-hop-vpn-tunnel=[VPN_TUNNEL]]
```

The parameter for creating a route indicates all the information we need to create a new route in GCP.

- `--destination-range`: This value indicates the destination range for the outgoing packet.
- `--network`: This indicates the network on which we define the rule. We must be careful to correctly indicate the network or we might apply the route to another network.
- `--tags`: We can use this option to indicate which VM we want to associate to the route.

The hop parameters can be specified only once. We can have two hop parameters in the same definition. The description of the parameters follows:

- `--next-hop-address`: This is used to specify the IP address where the route wants to send the traffic.
- `--next-hop-gateway`: This flag is used to specify via what subnet we can send the packet to the Internet connection.
- `--next-hop-instance`: This flag is used to specify the instance we want to use to send the packet.
- `--next-hop-vpn-tunnel`: This flag is used to specify through which VPN tunnel we want to send the packet.

The command for generating a new route is

```
gcloud compute --project=<your project name> routes create route-gcp  
--network=default --priority=1000 --destination-range=10.0.0.0/16  
--next-hop-gateway=default-internet-gateway
```

The result of the operation shows the creation of the route (Listing 10-7).

Listing 10-7. The Result of the Route Creation

```
gcloud compute --project=practicaldevopsgcp-197023 routes create route-  
gcp  
--network=default --priority=1000 --destination-range=  
10.0.0.0/16 --next-hop-gateway=default-internet-gateway
```

Created [<https://www.googleapis.com/compute/v1/projects/practicaldevopsgcp-197023/global/routes/route-gcp>].
NAME NETWORK DEST_RANGE NEXT_HOP PRIORITY route-gcp
default 10.0.0.0/16 default-internet-gateway 1000

With GCP, by default, we cannot forward an IP to another instance. This means that we can send a packet to an IP, unless the IP doesn't match the destination, but we must forward the IP when we want to create a new route or use the instance with the route packet. To forward the IP, we must create the instance with the flag `--can-ip-forward`. The command line for this follows:

```
gcloud compute instances create ... --can-ip-forward
```

This creates the instance with the ability to forward the IP and allows the instance to use the help of the route. The last command we must explore is that to delete a route. To delete a route, we can use the following command:

```
gcloud compute routes delete [ROUTE]
```

This command deletes the route indicated in the name.

Conclusion

This chapter covered the basics for creating and maintaining a network and a private cloud resource in GCP. This functionality is very important when we plan our cloud project. We must plan the network carefully, to be sure that it is always reachable and maintains an accurate SLA with the customer. In addition, a carefully planned network helps to prevent data breaches and service interruptions resulting from hacker attacks. Having strong firewall rules in place, in addition to good routing, helps to speed the network and prevent the most common attacks by hackers.

Index

A

Acceptance test, [46](#), [55](#), [104](#)
Amazon Web Services (AWS), [19](#), [22](#), [23](#), [175](#)
Automatic testing, [46](#)

B

Black-box monitoring, [16](#), [167–168](#), [171](#) Blue/green deployment, [59–60](#)
Branching strategies, [44–45](#)

C

Canary deployment, [61](#)
Canary server, [58](#)
Chief technology officer (CTO), [3](#) Classless inter-domain routing
([CIDR](#)), [249](#)

Cloud computing
characteristics, [20](#)
definition, [19–20](#)
deployment models

community, [22](#)
hybrid, [22](#)
[private](#), [22](#)
public, [22](#)

ROI, [23](#)
service-level agreements, [23](#) service models

IaaS, [21](#)
PaaS, [21](#)
SaaS, [21](#)
[TCO](#), [23](#)

Code inspection, [47](#), [49](#)
Code promotion, [46–47](#)
Code repository server, [40](#)
Code review, [55](#)

Continuous database integration, [53–54](#) Continuous delivery (CD)
chain, [43](#)
vs. CI, [44](#)
code inspection, [55–56](#)
continuous testing, [54–55](#)
costs and benefits, [47–48](#)
definition, [42](#)
designing, [49–50](#)
pipelines, [51–52](#)
strategies
automatic testing, [46](#)
branching, [44–45](#)
code inspection, [47](#)
code promotion, [46–47](#)
unit testing, [45–46](#)
Continuous integration (CI)
and CD system
blue/green deployment, [59–60](#) build reports, [57–58](#)
Continuous integration (CI) (*cont.*) canary deployment, [61](#)
canary server, [58](#)
cloud SaaS, [59](#)
differences, [44](#)
identifying code, repository, [57](#) incremental deployment, [62](#) shared location,
[58](#)
software maintenance windows, [59](#) chain, [38](#)
code inspection, [55–56](#)
code repository server, [40](#)
continuous testing, [54–55](#)
costs and benefits, [47–48](#)
definition, [37](#)
designing, [49–50](#)
phases, [37](#)
pipelines, [51–52](#)
scenario, [38–39](#)
server, [40–42](#)
Coordination, DevOps
automatic process, [9](#)

[direct, 10](#)

indirect, [10](#)

nonautomatic process, [9](#)

persistent, [11](#)

Customer identity management
(CIM), [223](#)

D

Database migration, [53](#)

Declarative approach, [195](#)

DevOps

adoption

Agile practices, [7](#)

automating, release process, [6](#) barriers to IT department, [6](#) changes, [3–4](#)

CI/CD policies, [5](#)

communication, [7–9](#)

coordination, [9–11](#)

developer and software, [4](#)

first-class citizens, [4](#)

infrastructure, [8](#)

software quality, [8](#)

Agile conference, [2](#)

chain, [11–13](#)

definition, [1](#)

development pipeline

branch policy, [14](#)

centralizing, building server, [15](#) continuous integration, [14](#)

monitoring, [15–16](#)

operations, [17](#)

unit test, [14](#)

engineer, [2](#)

- Digital entity
 - attributes, [226](#)
 - authentication, [228](#)
 - authorization, [228](#)
 - basic functions, [226](#)
 - definition, [224](#)
 - federated identification, [227](#)
 - pure identification, [226](#)
 - roles, [228](#)

- Direct coordination, [10](#)

- Docker

- application deployment, [64](#)

- components, [64](#)

- Google Compute Engine (*see* Google Compute Engine)

- hypervisor-based virtualization, [63](#) libcontainers, [64](#)

- microservice architecture, [64](#)

- operating-system virtualization, [63–64](#) PaaS cloud, [65](#)

- uses, [65](#)

- virtual machine (VM), [63](#)

E

- Elasticsearch, Logstash, and Kibana (ELK), [16](#)

- Employer identity management (EIM), [223](#)

F

- First-class citizens, [4](#)

G

- GenerateConfiguration() method, [217](#) Google App Engine, [24](#)

- Google BigQuery, [24](#)

- Google Cloud Bigtable, [24](#)

- Google Cloud Datastore, [24](#)

Google Cloud Function, [24](#)
Google Cloud Platform (GCP), [19](#)

areas, [23](#)
billing, [28–30](#)
Compute Engine page, [208](#)
configuration file, [196](#), [205](#), [213–214](#) console, [243](#)
creating, new project, [25–26](#)
deployment, [204](#), [207](#)
Deployment Manager page, [208](#), [215](#) [description](#), [23](#)
Docker (*see* Docker)
environment variable, [219](#)
Google dashboard, [26](#), [28](#)
Google SDK, [32–34](#)
home dashboard, [26](#)
IaC (*see* Infrastructure as code (IaC)) IAM permission board, [243](#)
initial page, [25](#)
Kubernetes (*see* Kubernetes)
manifest, [201](#)
new account, [24](#)
network reference, [214–215](#)
popular services, [23–24](#)
Python/Jinja2 template, [195](#)
resources, [30–31](#), [201](#), [206](#)
screen, [25](#)
selecting project, [27](#)
template, [197](#), [216](#)
types, [201](#)
virtual machine (VM), [209–211](#), [215](#)

Google Compute Engine, [23](#)
components, [66](#)
container application

CoreOS operating system, [75](#) creating instance, [76–77](#)
Docker image, [77–79](#)
Google Console dashboard, [76](#) Remote access selection, [78](#)

high-CPU, [67](#)
high-memory, [67](#)
instances

connections, [72–73](#)
dashboard, [72](#)
enabling API, console, [71](#)
Google SDK configuration, [68–69](#) managed, [74](#)
new project created, command

line, [70](#)
pre-built images, [73](#)
unmanaged, [74](#)

standard, [66](#)
Google Deployment Manager (GDM), [195](#) Google Kubernetes Engine, [24](#)

Google SDK
command line, console, [33](#) configuration, [68–69](#)
downloaded link, [32](#)
initialization, [32](#)
[operations, 33](#)
scripting languages, [34](#)

Google Storage, [24](#)

H

Hypervisor-based virtualization, [63](#)

I

Identity and access management (IAM) custom roles, [232](#)
database and shared resource, [228](#) digital entity (*see* Digital entity) digital
identity, [225](#)
elements, [230](#)
Google Account, [230](#)
Google ecosystem, [228](#)

- Google IAM, [229](#)
- Governance, [224](#)
- level of access, [223](#)
- permissions, [230](#)
- policy

- API via Java Code, [240](#)
- API via REST, [239](#)
- getIamPolicy() method, [235](#)
- Google IAM structure, [233](#)
- IAM Role, [233–234](#)
- JSON file, [235](#)
- level of access, [234](#)
- pattern, [234](#)

- predefined roles, [231–232](#)
- primitive roles, [231](#)
- [resource](#), [230](#)
- roles, [231](#)
- software, [224](#)

- Imperative approach, [195](#)
- Incremental deployment, [62](#)
- Indirect coordination, [10](#)
- Infrastructure as a service (IaaS), [21](#), [66](#)
- Infrastructure as code (IaC)

- architecture, [194–195](#)
- DevOps movement, [191](#)
- goals, [192](#)
- principles, [192–193](#)
- virtual machine (VM), [192](#)

- Integration test, [46](#), [55](#), [103](#)
- Intelligent approach, [195](#)

J

- Jenkins CI/CD pipelines
- branch indexing log, [141–142](#)
- branching strategy, [103–104](#)

- code, [101](#)
- code integration, [101](#)
- credentials
- Global, [138](#)
- Google account, [138–139](#)
- updated, [139](#)

- GitHub hook log, [141](#)
- GitHub hook trigger, [141](#)
- Git repository, [140](#)
- Jenkinsfile, [142–145](#)
- Kubernetes (*see* Kubernetes and

- Jenkins)
- multibranch project configuration, [139–140](#)
- plug-ins, [99](#)
- popularity, [100](#)
- production release, [102](#) stage release, [102](#)
- system testing, [102](#)
- unit test, [101](#)
- user acceptance, [102](#) Jenkins services
- complete file, [118–119](#) deployment file, [120–121](#) discovery service, [118](#)
- Docker image, [122](#)
- kubelet, [122](#)
- quotas, [123](#)
- resources, [123](#)
- service-ui, [117](#)

K, L

- Kubernetes
- cluster, [81](#)
- DaemonSet controller, [80](#)
- definition, [79–80](#)
- etcd, [81](#)
- job controller, [80](#)

- pods, [81](#)
- replication controller, [80](#)
- selectors, [80](#)
- [services, 81](#)

Kubernetes and Jenkins

- benefits, [104](#)
- cluster, [106, 108](#)
- creating repository, [135–138](#) credentials, [110](#)
- deployments, [116–117](#)

- administrative roles, RBAC, [125](#) cluster-admin, [125](#)
- health status, [128](#)
- ingress.yaml, [127](#)
- Jenkins login page, [129](#)
- load balancer, [128–129](#)
- option file, [124](#)
- password, [124](#)
- pods, [126](#)
- status of service, [126](#)
- TLS communication, [127](#)
- volume creation, [125](#)

- environments, [130](#)
- Google Cloud Container Registry, [109](#) Google Cloud Storage, [109](#)
- Jenkins services (*see* Jenkins services) master/slave architecture, [110–111](#)
- namespaces, [112–114](#)
- pods, [114](#)
- production namespace

- application IP, [134–135](#)
- back-end service, [131–132](#)
- front-end service, [132, 134](#)
- RESTful API object, [133](#)

- quotas, [116](#)
- scopes, URI, [106–108](#)

services, [114–115](#)
VPC creation, [105–106](#)

Kubernetes Engine

cluster

Compute Engine instance, [95–96](#) create, [83–84](#)
deleting, [96](#)
information, [94](#)
Kubernetes Compute

Engine, [93–94](#)

node pools, [94–95](#)

command line, [82](#)

dashboard

access UI, [88](#)

Kubernetes Engine (*cont .*)

Cloud SDK, service account activation, [90](#)

cluster information, [87](#)

Create key menu option, [89](#)

gcloud authorization, [90](#)

installing and configuring, [87](#)

login, [92–93](#)

Service Account screen, [88](#)

tokens, [90–92](#)

etcd, [85](#)

Google Cloud Shell button, [82](#) Google SDK, [82](#)

hello-server application, [85–87](#) kubectl version command, [82](#)

M

Microservices architecture

advantage, [148](#)

constraints, [149](#)

creating, [154](#)

Docker and Kubernetes

- characteristics, [152](#)
- designing, [152–153](#)
- monolithic system vs. microservices system, [151](#)
- fine-grained, [147](#)
- image publishing
 - Google Docker registry, [158](#)
 - Google SDK, [158](#)
- tagging, [158–161](#)
- Kubernetes cluster, [161–163](#)
- loosely coupled service, [147](#)
- philosophy, [149](#)
- principles, [148](#)
- pros and cons, [150](#)
- services
 - Dockerfile, [156–157](#)
 - Go code, [155](#)
- SOA, [147](#)
- Microsoft Azure, [19](#)
- Monitoring system
 - alerts, [169](#)
 - broken, [170](#)
 - components, [173](#)
 - data comparison, [169](#)
 - debugging, [169](#)
 - definition, [165](#)
 - error, [172](#)
 - factors, [167](#), [169](#)
 - infrastructure, [167](#)
 - latency, [172](#)
 - predictive analysis, [166](#)
 - real-time, [166](#)
 - requirements, [166](#)
 - sample memory graph, [167](#)
 - saturation, [172](#)
 - software, [165](#)
 - Stackdriver (see Stackdriver) system, [165](#)
 - temporal analysis, [169](#)
 - traffic, [172](#)
 - visualizing status, [169](#)

N

Network configuration and management cloud environment, [245](#)
components, [245](#)
[GCP, 265](#)
global resources, [247](#)
goals, [246](#)
regional resource, [245–246](#)
zonal resource, [245–246](#)

O

Operating-system virtualization, [63–64](#)

P, Q

Persistent coordination, [11](#)
Platform as a service (PaaS), [21](#), [65](#)

R

Real-time monitoring, [166](#)
Regional resource, [245–246](#)
Return on investment (ROI), [23](#) Risk-based authentication (RBA), [224](#) Role-based access control (RBAC), [125](#) Routes and firewalls

default route, [253](#)
destinations, [252](#)
gateways, [252](#)
network's security, [262](#)
network tagging, [254](#)
parameters, [263](#)
system-generated route, [252](#) target, [264](#)
types of sources, [264](#)
client installation, logging, [177](#) configuration command, [175–176](#) creating instance, [178](#)
dashboard

- creating, [187–188](#)
- testing, [189–190](#)
- installation, [176–177](#)
- log analysis, [181–183](#)
- menu from GCP, [174](#)
- monitoring section, [175](#)
- PHP-Apache installation package, [179–180](#)
- policy alert configuration, [185–187](#) projects, [175](#)
- SSH connection, [179](#)
- stackdriverinstance, [178–179](#) Static code analysis, [56, 101, 103](#) System monitoring, [165](#)
- System testing, [102–103](#)

T

- Test driven development (TDD), [45](#) Test pyramid, [46](#)
- Total cost of operation (TCO), [23](#)

S

- Service-oriented architecture (SOA), [147](#) Single sign-on (SSO), [227](#)
- Site reliability engineer (SRE), [2](#)
- Software as a service (SaaS), [21–22, 66](#) Software monitoring, [165](#)
- Stackdriver

- alerts, [183–185](#)
- Apache HTTP server default page, [181](#) applications, [174](#)

U

- Unit testing, [45–46](#)

V

- Verification test, [55](#)
- Virtual private cloud (VPC), [34, 105–106](#) automatic mode, [249, 256](#)
- cloud interconnect component, [248](#) command line network

creation, [260](#)

Virtual private cloud (VPC) (*cont .*) creation, [255](#)

custom mode network, [250](#), [257](#), [259](#), [261](#)

custom subnet, [258](#)

dynamic routing mode, [258](#) firewall rules, [260](#)

flow logs, [258](#)

global routing, [259](#)

Google Cloud Platform, [247](#) IP ranges, subnets, [250](#)

network and subnets, [249](#) Private Google access, [258](#) properties, [248](#)

regional routing, [259](#)

subnet network, [262](#)

W, X, Y

White-box monitoring, [16](#), [167–168](#), [171](#)

Z

Zonal resource, [245–246](#)