



2ND EDITION

Modern DevOps Practices

Implement, secure, and manage applications
on the public cloud by leveraging cutting-edge tools



GAURAV AGARWAL

Modern DevOps Practices

Implement, secure, and manage applications on the public cloud by leveraging cutting-edge tools

Gaurav Agarwal



BIRMINGHAM—MUMBAI

Modern DevOps Practices

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Surbhi Suman

Book Project Manager: Ashwini Gowda

Senior Editor: Isha Singh

Technical Editor: Rajat Sharma

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Tejal Soni

Production Designer: Joshua Misquitta

DevRel Marketing Coordinator: Linda Pearson and Rohan Dobhal

First published: September 2021

Second edition: January 2024

Production reference: 1131223

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 78-1-80512-182-4

www.packtpub.com

To my mother and the memory of my father, for their sacrifices and for exemplifying the power of determination. To my wife, for being my loving partner throughout our joint life journey.

Contributors

About the author

Gaurav Agarwal is a senior cloud engineer at ThoughtSpot, bringing over a decade of experience as a seasoned cloud and DevOps engineer. Previously, Gaurav served as a cloud solutions architect at Capgemini and began his career as a software developer at TCS. With a distinguished list of certifications, including HashiCorp Certified Terraform Associate, Google Cloud Certified Professional Cloud Architect, Certified Kubernetes Administrator, and Security Specialist, he possesses an impressive technical profile. Gaurav's extensive background encompasses roles where he played pivotal roles in infrastructure setup, cloud management, and the implementation of CI/CD pipelines. His technical prowess extends to numerous technical blog posts and a published book, underscoring his commitment to advancing the field.

I want to thank my family, especially my wonderful wife, Deepti, for giving me the space and support I needed to write this book. I'd also like to thank Surbhi for granting me the opportunity to complete this journey and Ashwini for keeping me on track. Special thanks to Aniket for reviewing the book. The whole Packt editing team has helped me immensely, but I'd like to give special thanks to Isha, Rajat, and Joshua for editing most of my work.

About the reviewer

Aniket Mhala is an accomplished and forward-thinking leader known for his unwavering dedication to strategic excellence and achievement. With over 25 years of diverse and multifaceted experience, Aniket possesses a formidable blend of technical prowess and business acumen. He is recognized for his exceptional capabilities in technology thought leadership, crafting and executing business and IT strategies, spearheading digital transformation initiatives, and modernizing DevOps practices. His expertise extends to application modernization, seamlessly integrating multiple cloud platforms (AWS, Azure, and OCI), designing modern data architectures, shaping enterprise architectures, and managing intricate programs and deliveries with precision.

I'd like to thank my family and friends who understand the time and commitment it takes to research and test data, commands, and DevOps code snippets on various clouds that are constantly changing. Working in this field would not be possible without their kind support.

Table of Contents

Part 1: Modern DevOps Fundamentals

1

The Modern Way of DevOps	3
What is DevOps?	4
Introduction to cloud computing	7
Understanding modern cloud-native applications	9
Modern DevOps versus traditional DevOps	10
The need for containers	11
The matrix of hell	13
Virtual machines	14
Containers	15
It works on my machine	15
Container architecture	15
Container networking	18
Containers and modern DevOps practices	20
Migrating from virtual machines to containers	22
Discovery	22
Application requirement assessment	23
Container infrastructure design	23
Containerizing the application	24
Testing	24
Deployment and rollout	25
What applications should go in containers?	26
Breaking the applications into smaller pieces	29
Are we there yet?	29
Summary	30
Questions	30
Answers	32

2

Source Code Management with Git and GitOps		33	
Technical requirements	33	Pulling and rebasing your code	44
What is source code management?	34	Git branches	46
A crash course on Git	34	Creating and managing Git branches	46
Installing Git	35	Working with pull requests	48
Initializing your first Git repository	35	What is GitOps?	51
Staging code changes	36	Why GitOps?	51
Displaying commit history	37	The principles of GitOps	52
Amending the last commit	38	Branching strategies and the GitOps workflow	53
Understanding remote repositories	39	The push model	53
Creating a remote Git repository	40	The pull model	53
Setting up authentication with the remote Git repository	40	Structuring the Git repository	54
Connecting the local repository to the remote repository	41	Git versus GitOps	58
Pushing changes from the local repository to the remote repository	41	Summary	59
		Questions	59
		Answers	60

3

Containerization with Docker		61	
Technical requirements	61	Troubleshooting containers	69
Installing Docker	62	Putting it all together	71
Introducing Docker storage drivers and volumes	64	Restarting and removing containers	72
Docker data storage options	64	Docker logging and logging drivers	73
Mounting volumes	65	Container log management	73
Docker storage drivers	65	Logging drivers	73
Configuring a storage driver	66	Configuring logging drivers	74
Running your first container	68	Typical challenges and best practices to address these challenges with Docker logging	76
Running containers from versioned images	68	Docker monitoring with Prometheus	77
Running Docker containers in the background	69	Challenges with container monitoring	77

Installing Prometheus	78	Deploying a sample application with Docker Compose	83
Configuring cAdvisor and the node exporter to expose metrics	78	Creating the docker-compose file	85
Configuring Prometheus to scrape metrics	79	Docker Compose best practices	87
Launching a sample container application	79	Summary	89
Metrics to monitor	82	Questions	89
Declarative container management with Docker Compose	83	Answers	90

4

Creating and Managing Container Images 91

Technical requirements	91	Multi-stage builds	106
Docker architecture	92	Managing Docker images	108
Understanding Docker images	94	Flattening Docker images	111
The layered filesystem	94	Optimizing containers with distroless images	113
Image history	96	Performance	114
Understanding Dockerfiles, components, and directives	97	Security	114
Can we use ENTRYPOINT instead of CMD?	98	Cost	114
Are RUN and CMD the same?	98	Understanding Docker registries	116
Building our first container	99	Hosting your private Docker registry	116
Building and managing Docker images	104	Other public registries	118
Single-stage builds	105	Summary	119
		Questions	121
		Answers	122

Part 2: Container Orchestration and Serverless

5

Container Orchestration with Kubernetes 125

Technical requirements	125	Kubernetes architecture	128
What is Kubernetes, and why do I need it?	126	Installing Kubernetes (Minikube and KinD)	130

Installing Minikube	131	Ensuring pod reliability	140
Installing KinD	132	Pod multi-container design patterns	143
Understanding Kubernetes pods	134	Summary	159
Running a pod	134	Questions	159
Using port forwarding	137	Answers	160
Troubleshooting pods	138		

6

Managing Advanced Kubernetes Resources 161

Technical requirements	162	Ingress resources	182
Spinning up GKE	162	Horizontal Pod autoscaling	189
The need for advanced Kubernetes resources	162	Managing stateful applications	192
Kubernetes Deployments	163	StatefulSet resources	193
ReplicaSet resources	164	Managing Persistent Volumes	194
Deployment resources	166	Kubernetes command-line best practices, tips, and tricks	203
Kubernetes Deployment strategies	169	Using aliases	203
Kubernetes Services and Ingresses	175	Using kubectl bash autocompletion	205
ClusterIP Service resources	176	Summary	205
NodePort Service resources	179	Questions	206
LoadBalancer Service resources	181	Answers	207

7

Containers as a Service (CaaS) and Serverless Computing for Containers 209

Technical requirements	210	Creating task definitions	215
The need for serverless offerings	210	Scheduling EC2 tasks on ECS	217
Amazon ECS with EC2 and Fargate	211	Scaling tasks	217
ECS architecture	211	Querying container logs from CloudWatch	218
Installing the AWS and ECS CLIs	214	Stopping tasks	218
Spinning up an ECS cluster	214	Scheduling Fargate tasks on ECS	218

Scheduling services on ECS	221	Spinning up GKE	229
Browsing container logs using the ECS CLI	222	Installing Knative	229
Deleting an ECS service	223	Deploying a Python Flask application on Knative	231
Load balancing containers running on ECS	223	Load testing your app on Knative	233
Other CaaS services	225	Summary	234
Open source CaaS with Knative	226	Questions	234
Knative architecture	227	Answers	235

Part 3: Managing Config and Infrastructure

8

Infrastructure as Code (IaC) with Terraform	239		
Technical requirements	239	Terraform modules	253
Introduction to IaC	240	Managing Terraform state	256
Installing Terraform	242	Using the Azure Storage backend	257
Terraform providers	243	Terraform workspaces	260
Authentication and authorization with Azure	243	Inspecting resources	264
Using the Azure Terraform provider	245	Inspecting state files	266
Cleaning up			267
Terraform variables	246	Terraform output, state, console, and graphs	267
Providing variable values	247	terraform output	267
Terraform workflow	248	Managing Terraform state	268
terraform init	249	terraform console	270
Creating the first resource – Azure resource group	249	Terraform dependencies and graphs	270
terraform fmt	250	Cleaning up resources	271
terraform validate	250	Summary	272
terraform plan	251	Questions	272
terraform apply	251	Answers	273
terraform destroy	252		

9

Configuration Management with Ansible	275
Technical requirements	275
Introduction to configuration management	276
Setting up Ansible	279
Setting up inventory	280
Connecting the Ansible control node with inventory servers	280
Installing Ansible in the control node	283
Setting up an inventory file	283
Setting up the Ansible configuration file	285
Ansible tasks and modules	286
Introduction to Ansible playbooks	287
Checking playbook syntax	289
Applying the first playbook	289
Ansible playbooks in action	290
Updating packages and repositories	290
Installing application packages and services	291
Configuring applications	292
Combining playbooks	294
Executing playbooks	294
Designing for reusability	296
Ansible variables	296
Sourcing variable values	298
Jinja2 templates	300
Ansible roles	300
Summary	305
Questions	306
Answers	307

10

Immutable Infrastructure with Packer	309
Technical requirements	309
Immutable infrastructure with HashiCorp's Packer	310
When to use immutable infrastructure	313
Installing Packer	315
Creating the required infrastructure with Terraform	324
Summary	331
Questions	331
Answers	332

Part 4: Delivering Applications with GitOps

11

Continuous Integration with GitHub Actions and Jenkins 335

Technical requirements	336	Configure build reporting	368
The importance of automation	336	Customize the build server size	368
Introduction to the sample microservices-based blogging application – Blog App	338	Ensure that your builds only contain what you need	368
Building a CI pipeline with GitHub Actions	339	Parallelize your builds	368
Creating a GitHub repository	342	Make use of caching	368
Creating a GitHub Actions workflow	343	Use incremental building	368
Scalable Jenkins on Kubernetes with Kaniko	349	Optimize testing	369
Spinning up Google Kubernetes Engine	352	Use artifact management	369
Creating the Jenkins CaC (JCasC) file	353	Manage application dependencies	369
Installing Jenkins	357	Utilize Infrastructure as Code	369
Running our first Jenkins job	361	Use containerization to manage build and test environments	369
Automating a build with triggers	365	Utilize cloud-based CI/CD	369
Building performance best practices	367	Monitor and profile your CI/CD pipelines	369
Aim for faster builds	367	Pipeline optimization	369
Always use post-commit triggers	368	Implement automated cleanup	370
		Documentation and training	370
		Summary	370
		Questions	370
		Answers	371

12

Continuous Deployment/Delivery with Argo CD 373

Technical requirements	373	Complex deployment models	378
The importance of CD and automation	374	The Blog App and its deployment configuration	379
CD models and tools	376	Continuous declarative IaC using an Environment repository	382
Simple deployment model	377		

Creating and setting up our Environment repository	382	Managing sensitive configurations and Secrets	398
Introduction to Argo CD	388	Installing the Sealed Secrets operator	399
Installing and setting up Argo CD	390	Installing kubeseal	401
Terraform changes	391	Creating Sealed Secrets	401
The Kubernetes manifests	393	Deploying the sample Blog App	402
Argo CD Application and ApplicationSet	393	Summary	406
Accessing the Argo CD Web UI	396	Questions	407
		Answers	408

13

Securing and Testing Your CI/CD Pipeline	409		
Technical requirements	409	Merging code and deploying to prod	443
Securing and testing CI/CD pipelines	410	Security and testing best practices for modern DevOps pipelines	445
Revisiting the Blog Application	414	Adopt a DevSecOps culture	446
Container vulnerability scanning	415	Establish access control	446
Installing Anchore Grype	416	Implement shift left	446
Scanning images	417	Manage security risks consistently	446
Managing secrets	420	Implement vulnerability scanning	446
Creating a Secret in Google Cloud Secret Manager	421	Automate security	447
Accessing external secrets using External Secrets Operator	421	Test automation within your CI/CD pipelines	447
Setting up the baseline	422	Manage your test data effectively	447
Installing external secrets with Terraform	424	Test all aspects of your application	447
Testing your application within the CD pipeline	431	Implement chaos engineering	448
CD workflow changes	431	Monitor and observe your application when it is being tested	448
Binary authorization	434	Effective testing in production	448
Setting up binary authorization	436	Documentation and knowledge sharing	448
Release gating with pull requests and deployment to production	441	Summary	449
		Questions	449
		Answers	450

Part 5: Operating Applications in Production

14

Understanding Key Performance Indicators (KPIs) for Your Production Service	453
Understanding the importance of reliability	453
Understanding SLIs, SLOs, and SLAs	456
SLIs	456
SLOs	458
SLAs	459
Error budgets	460
Disaster recovery, RTO, and RPO	462
Running distributed applications in production	463
Summary	464
Questions	465
Answers	466

15

Implementing Traffic Management, Security, and Observability with Istio	467
Technical requirements	468
Setting up the baseline	468
Revisiting the Blog App	471
Introduction to service mesh	472
Introduction to Istio	475
Traffic management	475
Security	476
Observability	476
Developer-friendly	476
Understanding the Istio architecture	476
The control plane architecture	478
The data plane architecture	478
Installing Istio	480
Enabling automatic sidecar injection	484
Using Istio ingress to allow traffic	485
Securing your microservices using Istio	487
Creating secure ingress gateways	489
Enforcing TLS within your service mesh	491
Managing traffic with Istio	497
Traffic shifting and canary rollouts	502
Traffic mirroring	504
Observing traffic and alerting with Istio	507
Accessing the Kiali dashboard	507
Monitoring and alerting with Grafana	509
Summary	513
Questions	514
Answers	515

Appendix: The Role of AI in DevOps	517
What is AI?	517
The role of AI in the DevOps infinity loop	518
Code development	519
Software testing and quality assurance	520
Continuous integration and delivery	521
Software operations	522
Summary	524
Index	525
Other Books You May Enjoy	542

Preface

The new and improved Second Edition of this book goes beyond just the fundamentals of DevOps tools and their deployments. It covers practical examples to get you up to speed with containers, infrastructure automation, serverless container services, continuous integration and delivery, automated deployments, deployment pipeline security, and operating your services in production, all using containers and GitOps as a special focus.

Who this book is for

If you are a software engineer, system administrator, or operations engineer looking to step into the world of DevOps within public cloud platforms, this book is for you. Current DevOps engineers will also find this book useful, as it covers the best practices, tips, and tricks to implement DevOps with a cloud-native mindset. Although no containerization experience is necessary, a basic understanding of the software development life cycle and delivery will help you get the most out of the book.

What this book covers

Chapter 1, The Modern Way of DevOps, delves into the realm of modern DevOps, emphasizing its distinctions from traditional DevOps. We'll explore the core technologies propelling modern DevOps, with a special emphasis on the central role played by containers. Given that containers are a relatively recent development, we'll delve into the essential best practices and techniques to develop, deploy, and secure container-based applications.

Chapter 2, Source Code Management with Git and GitOps, introduces us to Git, the leading source code management tool, and its application in managing software development and delivery through GitOps.

Chapter 3, Containerization with Docker, initiates our journey into Docker, encompassing installation, Docker storage configuration, launching initial containers, and monitoring Docker via Journald and Splunk.

Chapter 4, Creating and Managing Container Images, dissects Docker images, a critical component in Docker usage. We'll understand Docker images, the layered model, Dockerfile directives, image flattening, image construction, and image-building best practices. Additionally, we'll explore distroless images and their relevance from a DevSecOps standpoint.

Chapter 5, Container Orchestration with Kubernetes, introduces Kubernetes. We'll install Kubernetes using minikube and kinD, delve into Kubernetes' architectural underpinnings, and explore fundamental Kubernetes building blocks such as Pods, containers, ConfigMaps, secrets, probes, and multi-container Pods.

Chapter 6, Managing Advanced Kubernetes Resources, advances into intricate Kubernetes concepts, encompassing networking, DNS, Services, Deployments, the HorizontalPodAutoscaler, and StatefulSets.

Chapter 7, Containers as a Service (CaaS) and Serverless Computing for Containers, explores the hybrid nature of Kubernetes, bridging IaaS and PaaS paradigms. Additionally, we will examine serverless container services such as AWS ECS, alongside alternatives such as Google Cloud Run and Azure Container Instances. We will conclude with a discussion on Knative, an open source, cloud-native, and serverless technology.

Chapter 8, Infrastructure as Code (IaC) with Terraform, introduces IaC using Terraform, elucidating its core principles. We will proceed with hands-on examples, creating a resource group and virtual machine from scratch on Azure using Terraform, while grasping essential Terraform concepts.

Chapter 9, Configuration Management with Ansible, acquaints us with configuration management through Ansible and its foundational principles. We will explore key Ansible concepts by configuring a MySQL and Apache application on Azure Virtual Machines.

Chapter 10, Immutable Infrastructure with Packer, delves into immutable infrastructure using Packer. We will integrate this with insights from *Chapter 8, Infrastructure as Code (IaC) with Terraform*, and *Chapter 9, Configuration Management with Ansible*, to launch an IaaS-based Linux, Apache, MySQL, and PHP (LAMP) stack on Azure.

Chapter 11, Continuous Integration with GitHub Actions and Jenkins, explains continuous integration from a container-centric perspective, evaluating various tools and methodologies to continuously build container-based applications. We will examine tools such as GitHub Actions and Jenkins, discerning when and how to employ each one while deploying an example microservices-based distributed application, the Blog app.

Chapter 12, Continuous Deployment/Delivery with Argo CD, delves into continuous deployment/delivery, employing Argo CD. As a contemporary GitOps-based continuous delivery tool, Argo CD streamlines the deployment and management of container applications. We will harness its power to deploy our example Blog App.

Chapter 13, Securing and Testing the Deployment Pipeline, explores multiple strategies to secure a container deployment pipeline, encompassing container image analysis, vulnerability scanning, secrets management, storage, integration testing, and binary authorization. We will integrate these techniques to enhance the security of our existing CI/CD pipelines.

Chapter 14, Understanding Key Performance Indicators (KPIs) for Your Production Service, introduces site reliability engineering and investigates a range of key performance indicators, vital for effectively managing distributed applications in production.

Chapter 15, Operating Containers in Production with Istio, acquaints you with the widely adopted service mesh technology Istio. We will explore various techniques for day-to-day operations in production, including traffic management, security measures, and observability enhancements for our example Blog app.

To get the most out of this book

For this book, you will need the following:

- An Azure subscription to perform some of the exercises: Currently, Azure offers a free trial for 30 days with \$200 worth of free credits; sign up at <https://azure.microsoft.com/en-in/free>.
- An AWS subscription: Currently, AWS offers a free tier for some products. You can sign up at <https://aws.amazon.com/free>. The book uses some paid services, but we will try to minimize how many we use as much as possible during the exercises.
- A Google Cloud Platform subscription: Currently, Google Cloud Platform provides a free \$300 trial for 90 days, which you can go ahead and sign up for at <https://console.cloud.google.com/>.
- For some chapters, you will need to clone the following GitHub repository to proceed with the exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>

Software/hardware covered in the book	Operating system requirements
Google Cloud Platform	Windows, macOS, or Linux
AWS	Windows, macOS, or Linux
Azure	Windows, macOS, or Linux
Linux VM	Ubuntu 18.04 LTS or later

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Let’s try to raise a pull request for merging our code from the `feature/feature1` branch to the `master` branch.”

A block of code is set as follows:

```
import os
import datetime
from flask import Flask
app = Flask(__name__)
@app.route('/')
def current_time():
    ct = datetime.datetime.now()
    return 'The current time is : {}!\n'.format(ct)
if __name__ == "__main__":
    app.run(debug=True,host='0.0.0.0')
```

Any command-line input or output is written as follows:

```
$ cp ~/modern-devops/ch13/install-external-secrets/app.tf \
terraform/app.tf
$ cp ~/modern-devops/ch13/install-external-secrets/\
external-secrets.yaml manifests/argocd/
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Click on the **Create pull request** button to create the pull request.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share your thoughts

Once you've read *Modern DevOps Practices*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805121824>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1:

Modern DevOps

Fundamentals

This part will introduce you to the world of modern DevOps and containers and build a strong foundation of knowledge regarding container technologies. In this section, you will learn how containers help organizations build distributed, scalable, and reliable systems in the cloud.

This part has the following chapters:

- *Chapter 1, The Modern Way of DevOps*
- *Chapter 2, Source Code Management with Git and GitOps*
- *Chapter 3, Containerization with Docker*
- *Chapter 4, Creating and Managing Container Images*



1

The Modern Way of DevOps

This first chapter will provide some background knowledge of DevOps practices, processes, and tools. We will understand modern DevOps and how it differs from traditional DevOps. We will also introduce containers and understand in detail how containers within the cloud change the entire IT landscape so that we can build on this book's base. While this book does not entirely focus on containers and their orchestration, modern DevOps practices heavily emphasize it.

In this chapter, we're going to cover the following main topics:

- What is DevOps?
- Introduction to cloud computing
- Understanding modern cloud-native applications
- Modern DevOps versus traditional DevOps
- The need for containers
- Container architecture
- Containers and modern DevOps practices
- Migrating to containers from virtual machines

By the end of this chapter, you should understand the following key aspects:

- What DevOps is and what role it plays in the modern IT landscape
- What cloud computing is and how it has changed IT services
- What a modern cloud-native application looks like and how it has changed DevOps
- Why we need containers and what problems they solve
- The container architecture and how it works
- How containers contribute to modern DevOps practices
- The high-level steps of moving from a virtual machine-based architecture to containers

What is DevOps?

As you know, software development and operations were traditionally handled by separate teams with distinct roles and responsibilities. Developers focused on writing code and creating new features, while operations teams focused on deploying and managing the software in production environments. This separation often led to communication gaps, slow release cycles, and inefficient processes.

DevOps bridges the gap between development and operations by promoting a culture of collaboration, shared responsibilities, and continuous feedback using automation throughout the software development life cycle.

It is a set of principles and practices, as well as a philosophy, that encourage the participation of the development and operations teams in the entire software development life cycle, including software maintenance and operations. To implement this, organizations manage several processes and tools that help automate the software delivery process to improve speed and agility, reduce the cycle time of code release through **continuous integration and continuous delivery (CI/CD)** pipelines, and monitor the applications running in production.

A DevOps team should ensure that instead of having a clear set of siloed groups that do development, operations, and QA, they have a single team that takes care of the entire SDLC life cycle – that is, the team will build, deploy, and monitor the software. The combined team owns the whole application instead of certain functions. That does not mean that people don't have specialties, but the idea is to ensure that developers know something about operations and that operations engineers know something about development. The QA team works hand in hand with developers and operations engineers to understand the business requirements and various issues faced in the field. Based on these learnings, they need to ensure that the product they are developing meets business requirements and addresses problems encountered in the field.

In a traditional development team, the source of the backlog is the business and its architects. However, for a DevOps team, there are two sources of their daily backlog – the business and its architects and the customers and issues that they face while they're operating their application in production. Therefore, instead of following a linear path of software delivery, DevOps practices generally follow an infinity loop, as shown in the following figure:



Figure 1.1 – DevOps infinity loop

To ensure smooth interoperability between people of different skill sets, DevOps focuses heavily on automation and tools. DevOps aims to ensure that we try to automate repeatable tasks as much as possible and focus on more important things. This ensures product quality and speedy delivery. DevOps focuses on *people, processes, and tools*, giving the most importance to people and the least to tools. We generally use tools to automate processes that help people achieve the right goals.

Some of the fundamental ideas and jargon that a DevOps engineer generally encounters are as follows. We are going to focus heavily on each throughout this book:

- **Continuous integration (CI)**

CI is a software development practice that involves frequently merging code changes from multiple developers into a shared repository, typically several times a day. This ensures that your developers regularly merge code into a central repository where automated builds and tests run to provide real-time feedback to the team. This reduces cycle time significantly and improves the quality of code. This process aims to minimize bugs within the code early within the cycle rather than later during the test phases. It detects integration issues early and ensures that the software always remains in a releasable state.

- **Continuous delivery (CD)**

CD is all about shipping your tested software into your production environment whenever it is ready. So, a CD pipeline will build your changes into packages and run integration and system tests on them. Once you have thoroughly tested your code, you can automatically (or on approval) deploy changes to your test and production environments. So, CD aims to have the latest set of tested artifacts ready to deploy.

- **Infrastructure as Code (IaC)**

IaC is a practice in software development that involves managing and provisioning infrastructure resources, such as servers, networks, and storage, using code and configuration files rather than

manual processes. IaC treats infrastructure as software, enabling teams to define and manage infrastructure resources in a programmable and version-controlled manner. With the advent of virtual machines, containers, and the cloud, technology infrastructure has become virtual to a large extent. This means we can build infrastructure through API calls and templates. With modern tools, we can also build infrastructure in the cloud declaratively. This means that you can now build IaC, store the code needed to build the infrastructure within a source code repository such as Git, and use a CI/CD pipeline to spin and manage the infrastructure.

- **Configuration as code (CaC)**

CaC is a practice in software development and system administration that involves managing and provisioning configuration settings using code and version control systems. It treats configuration settings as code artifacts, enabling teams to define, store, and manage configuration in a programmatic and reproducible manner. Historically, servers used to be built manually from scratch and seldom changed. However, with elastic infrastructure in place and an emphasis on automation, the configuration can also be managed using code. CaC goes hand in hand with IaC for building scalable, fault-tolerant infrastructure so that your application can run seamlessly.

- **Monitoring and logging**

Monitoring and logging are essential practices in software development and operations that involve capturing and analyzing data about the behavior and performance of software applications and systems. They provide insights into the software's health, availability, and performance, enabling teams to identify issues, troubleshoot problems, and make informed decisions for improvement. Monitoring and logging come under observability, which is a crucial area for any DevOps team – that is, knowing when your application has issues and exceptions using monitoring and triaging them using logging. These practices and tools form your eye, and it is a critical area in the DevOps stack. In addition, they contribute a lot to building the backlog of a DevOps team.

- **Communication and collaboration**

Communication and collaboration are crucial aspects of DevOps practices. They promote effective teamwork, knowledge sharing, and streamlined workflows across development, operations, and other stakeholders involved in the software delivery life cycle. Communication and collaboration make a DevOps team function well. Gone are the days when communication used to be through emails. Instead, modern DevOps teams manage their backlog using ticketing and Agile tools, keep track of their knowledge articles and other documents using a wiki, and communicate instantly using chat and **instant messaging (IM)** tools.

While these are just a few core aspects of DevOps practices and tools, there have been recent changes with the advent of containers and the cloud – that is, the modern cloud-native application stack. Now that we've covered a few buzzwords in this section, let's understand what we mean by the cloud and cloud computing.

Introduction to cloud computing

Traditionally, software applications used to run on servers that ran on in-house computers (servers), known as **data centers**. This meant that an organization would have to buy and manage physical computer and networking infrastructure, which used to be a considerable capital expenditure, plus they had to spend quite a lot on operating expenses. In addition, servers used to fail and required maintenance. This meant smaller companies who wanted to try things would generally not start because of the huge **capital expenditure (CapEx)** involved. This suggested that projects had to be well planned, budgeted, and architected well, and then infrastructure was ordered and provisioned accordingly. This also meant that quickly scaling infrastructure with time would not be possible. For example, suppose you started small and did not anticipate much traffic on the site you were building. Therefore, you ordered and provisioned fewer resources, and the site suddenly became popular. In that case, your servers won't be able to handle that amount of traffic and will probably crash. Scaling that quickly would involve buying new hardware and then adding it to the data center, which would take time, and your business may lose that window of opportunity.

To solve this problem, internet giants such as Amazon, Microsoft, and Google started building public infrastructure to run their internet systems, eventually leading them to launch it for public use. This led to a new phenomenon known as **cloud computing**.

Cloud computing refers to delivering on-demand computing resources, such as servers, storage, databases, networking, software, and analytics, over the internet. Rather than hosting these resources locally on physical infrastructure, cloud computing allows organizations to access and utilize computing services provided by **cloud service providers (CSPs)**. Some of the leading public CSPs are **Amazon Web Services (AWS)**, **Microsoft Azure**, and **Google Cloud Platform**.

In cloud computing, the CSP owns, maintains, and manages the underlying infrastructure and resources, while the users or organizations leverage these resources for their applications and services.

Simply put, cloud computing is nothing but using someone else's data center to run your application, which should be on demand. It should have a control panel through a web portal, APIs, and so on over the internet to allow you to do so. In exchange for these services, you need to pay rent for the resources you provision (or use) on a pay-as-you-go basis.

Therefore, cloud computing offers several benefits and opens new doors for businesses like never before. Some of these benefits are as follows:

- **Scalability:** Resources on the cloud are scalable. This means you can add new servers or resources to existing servers when needed. You can also automate scaling with traffic for your application. This means that if you need one server to run your application, and suddenly because of popularity or peak hours, you need five, your application can automatically scale to five servers using cloud computing APIs and inbuilt management resources. This gives businesses a lot of power as they can now start small, and they do not need to bother much about future popularity and scale.

- **Cost savings:** Cloud computing follows a **pay-as-you-go** model, where users only pay for the resources and services they consume. This eliminates the need for upfront CapEx on hardware and infrastructure. It is always cheaper to rent for businesses rather than invest in computing hardware. Therefore, as you pay only for the resources you need at a certain period, there is no need to overprovision resources to cater to the future load. This results in substantial cost savings for most small and medium organizations.
- **Flexibility:** Cloud resources are no longer only servers. You can get many other things, such as simple object storage solutions, network and block storage, managed databases, container services, and more. These provide you with a lot of flexibility regarding what you do with your application.
- **Reliability:** Cloud computing resources are bound by **service-level agreements (SLAs)**, sometimes in the order of 99.999% availability. This means that most of your cloud resources will never go down; if they do, you will not notice this because of built-in redundancy.
- **Security:** Since cloud computing companies run applications for various clients, they often have a stricter security net than you can build on-premises. They have a team of security experts manning the estate 24/7, and they have services that offer encryption, access control, and threat detection by default. As a result, when architected correctly, an application running on the cloud is much more secure.

There are a variety of cloud computing services on offer, including the following:

- **Infrastructure-as-a-Service (IaaS)** is similar to running your application on servers. It is a cloud computing service model that provides virtualized computing resources over the internet. With IaaS, organizations can access and manage fundamental IT infrastructure components, such as virtual machines, storage, and networking, without investing in and maintaining physical hardware. In the IaaS model, the CSP owns and manages the underlying physical infrastructure, including servers, storage devices, networking equipment, and data centers. Users or organizations, on the other hand, have control over the **operating systems (OSs)**, applications, and configurations running on the virtualized infrastructure.
- **Platform-as-a-Service (PaaS)** gives you an abstraction where you can focus on your code and leave your application management to the cloud service. It is a cloud computing service model that provides a platform and environment for developers to build, deploy, and manage applications without worrying about underlying infrastructure components. PaaS abstracts the complexities of infrastructure management, allowing developers to focus on application development and deployment. In the PaaS model, the CSP offers a platform that includes OSs, development frameworks, runtime environments, and various tools and services needed to support the application development life cycle. Users or organizations can leverage these platform resources to develop, test, deploy, and scale their applications.
- **Software-as-a-Service (SaaS)** provides a pre-built application for your consumption, such as a monitoring service that's readily available for you to use that you can easily plug and play

with your application. In the SaaS model, the CSP hosts and manages the software application, including infrastructure, servers, databases, and maintenance. Users or organizations can access the application through a web browser or a thin client application. They typically pay a subscription fee based on usage, and the software is delivered as a service on demand.

The advent of the cloud has led to a new buzzword in the industry called cloud-native applications. We'll look at them in the next section.

Understanding modern cloud-native applications

When we say cloud-native, we talk about applications built to run natively on the cloud. A cloud-native application is designed to run in the cloud taking full advantage of the capabilities and benefits of the cloud using cloud services as much as possible.

These applications are inherently **scalable**, **flexible**, and **resilient** (fault-tolerant). They rely on cloud services and automation to a large extent.

Some of the characteristics of a modern cloud-native application are as follows:

Microservices architecture: Modern cloud-native applications typically follow the microservices architecture. Microservices are applications that are broken down into multiple smaller, loosely coupled parts with independent business functions. Independent microservices can be written in different programming languages based on the need or specific functionality. These smaller parts can then independently scale, are flexible to run, and are resilient by design.

Containerization: Microservices applications typically use containers to run. Containers provide a **consistent**, **portable**, and **lightweight** environment for applications to run, ensuring that they have all the necessary dependencies and configurations bundled together. Containers can run the same on all environments and cloud platforms.

DevOps and automation: Cloud-native applications heavily use modern DevOps practices and tools and therefore rely on automation to a considerable extent. This streamlines development, testing, and operations for your application. Automation also brings about **scalability**, **resilience**, and **consistency**.

Dynamic orchestration: Cloud-native applications are built to scale and are inherently meant to be fault tolerant. These applications are typically **ephemeral** (**transient**); therefore, replicas of services can come and go as needed. Dynamic orchestration platforms such as **Kubernetes** and **Docker Swarm** are used to manage these services. These tools help run your application under changing demands and traffic patterns.

Use of cloud-native data services: Cloud-native applications typically use managed cloud data services such as **storage**, **databases**, **caching**, and **messaging** systems to allow for communication between multiple services.

Cloud-native systems emphasize DevOps, and modern DevOps has emerged to manage them. So, now, let's look at the difference between traditional and modern DevOps.

Modern DevOps versus traditional DevOps

DevOps' traditional approach involved establishing a DevOps team consisting of **Dev**, **QA**, and **Ops** members and working toward creating better software faster. However, while there would be a focus on automating software delivery, automation tools such as **Jenkins**, **Git**, and others were installed and maintained manually. This led to another problem as we now had to manage another set of IT infrastructure. It finally boiled down to infrastructure and configuration, and the focus was to automate the automation process.

With the advent of containers and the recent boom in the public cloud landscape, DevOps' modern approach came into the picture, which involved automating everything. From provisioning infrastructure to configuring tools and processes, there is code for everything. So, now, we have **IaC**, **CaC**, **immutable infrastructure**, and **containers**. I call this approach to DevOps modern DevOps, and it will be the focus of this book.

The following table describes some of the key similarities and differences between modern DevOps and traditional DevOps:

Aspect	Modern DevOps	Traditional DevOps
Software Delivery	Emphasis on CI/CD pipelines, automated testing, and deployment automation.	Emphasis on CI/CD pipelines, automated testing, and deployment automation.
Infrastructure management	IaC is commonly used to provision and manage infrastructure resources. Cloud platforms and containerization technologies are often utilized.	Manual provisioning and configuration of infrastructure is done, often relying on traditional data centers and limited automation.
Application deployment	Containerization and container orchestration technologies, such as Docker and Kubernetes, are widely adopted to ensure application portability and scalability.	Traditional deployment methods are used, such as deploying applications directly on virtual machines or physical servers without containerization.
Scalability and resilience	Utilizes the auto-scaling capabilities of cloud platforms and container orchestration to handle varying workloads. Focuses on high availability and fault tolerance.	Scalability is achieved through vertical scaling (adding resources to existing servers) or manual capacity planning. High availability is achieved by adding redundant servers manually. Elasticity is non-existent, and fault tolerance is not a focus.

Monitoring and logging	Extensive use of monitoring tools, log aggregation, and real-time analytics to gain insights into application and infrastructure performance.	Limited monitoring and logging practices, with fewer tools and analytics available.
Collaboration and culture	Emphasizes collaboration, communication, and shared ownership between development and operations teams (DevOps culture).	Emphasizes collaboration, communication, and shared ownership between development and operations teams (DevOps culture).
Security	Security is integrated into the development process with the use of DevSecOps practices. Security testing and vulnerability scanning are automated.	Security measures are often applied manually and managed by a separate security team. There is limited automated security testing in the SDLC.
Speed of deployment	Rapid and frequent deployment of software updates through automated pipelines, enabling faster time-to-market.	Rapid application deployments, but automated infrastructure deployments are often lacking.

Table 1.1 – Key similarities and differences between modern DevOps and traditional DevOps

It's important to note that the distinction between modern DevOps and traditional DevOps is not strictly binary as organizations can adopt various practices and technologies along a spectrum. The modern DevOps approach generally focuses on leveraging cloud technologies, automation, containerization, and DevSecOps principles to enhance collaboration, agility, and software development and deployment efficiency.

As we discussed previously, containers help implement modern DevOps and form the core of the practice. We'll have a look at containers in the next section.

The need for containers

Containers are in vogue lately and for excellent reason. They solve the computer architecture's most critical problem – *running reliable, distributed software with near-infinite scalability in any computing environment*.

They have enabled an entirely new discipline in software engineering – *microservices*. They have also introduced the *package once deploy anywhere* concept in technology. Combined with the cloud

and distributed applications, containers with container orchestration technology have led to a new buzzword in the industry – *cloud-native* – changing the IT ecosystem like never before.

Before we delve into more technical details, let's understand containers in plain and simple words.

Containers derive their name from shipping containers. I will explain containers using a shipping container analogy for better understanding. Historically, because of transportation improvements, a lot of stuff moved across multiple geographies. With various goods being transported in different modes, loading and unloading goods was a massive issue at every transportation point. In addition, with rising labor costs, it was impractical for shipping companies to operate at scale while keeping prices low.

Also, it resulted in frequent damage to items, and goods used to get misplaced or mixed up with other consignments because there was no isolation. There was a need for a standard way of transporting goods that provided the necessary isolation between consignments and allowed for easy loading and unloading of goods. The shipping industry came up with shipping containers as an elegant solution to this problem.

Now, shipping containers have simplified a lot of things in the shipping industry. With a standard container, we can ship goods from one place to another by only moving the container. The same container can be used on roads, loaded on trains, and transported via ships. The operators of these vehicles don't need to worry about what is inside the container most of the time. The following figure depicts the entire workflow graphically for ease of understanding:

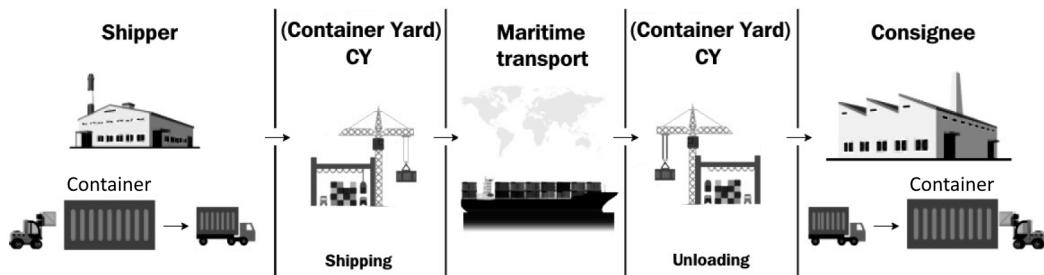


Figure 1.2 – Shipping container workflow

Similarly, there have been issues with software portability and compute resource management in the software industry. In a standard software development life cycle, a piece of software moves through multiple environments, and sometimes, numerous applications share the same OS. There may be differences in the configuration between environments, so software that may have worked in a development environment may not work in a test environment. Something that worked in test may also not work in production.

Also, when you have multiple applications running within a single machine, there is no isolation between them. One application can drain compute resources from another application, and that may lead to runtime issues.

Repackaging and reconfiguring applications is required in every step of deployment, so it takes a lot of time and effort and is sometimes error-prone.

In the software industry, containers solve these problems by providing isolation between application and compute resource management, which provides an optimal solution to these issues.

The software industry's biggest challenge is to provide application isolation and manage external dependencies elegantly so that they can run on any platform, irrespective of the OS or the infrastructure. Software is written in numerous programming languages and uses various dependencies and frameworks. This leads to a scenario called the **matrix of hell**.

The matrix of hell

Let's say you're preparing a server that will run multiple applications for multiple teams. Now, assume that you don't have a virtualized infrastructure and that you need to run everything on one physical machine, as shown in the following diagram:

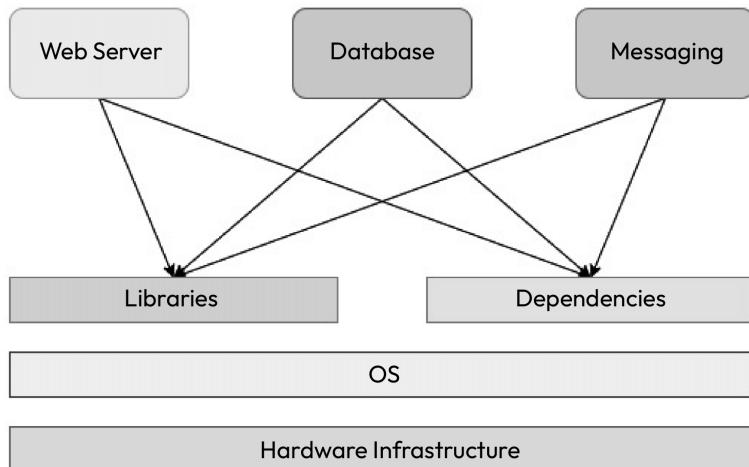


Figure 1.3 – Applications on a physical server

One application uses one particular version of a dependency, while another application uses a different one, and you end up managing two versions of the same software in one system. When you scale your system to fit multiple applications, you will be managing hundreds of dependencies and various versions that cater to different applications. It will slowly turn out to be unmanageable within one physical system. This scenario is known as the **matrix of hell** in popular computing nomenclature.

Multiple solutions come out of the matrix of hell, but there are two notable technological contributions – *virtual machines* and *containers*.

Virtual machines

A **virtual machine** emulates an OS using a technology called a **hypervisor**. A hypervisor can run as software on a physical host OS or run as firmware on a bare-metal machine. Virtual machines run as a virtual guest OS on the hypervisor. With this technology, you can subdivide a sizeable physical machine into multiple smaller virtual machines, each catering to a particular application. This has revolutionized computing infrastructure for almost two decades and is still in use today. Some of the most popular hypervisors on the market are **VMware** and **Oracle VirtualBox**.

The following diagram shows the same stack on virtual machines. You can see that each application now contains a dedicated guest OS, each of which has its own libraries and dependencies:

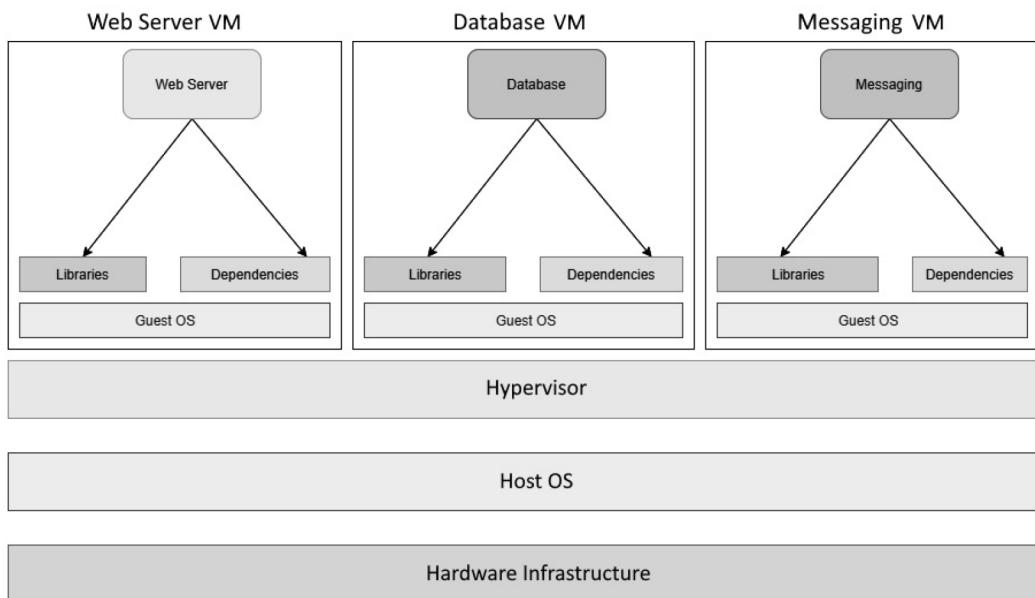


Figure 1.4 – Applications on virtual machines

Though the approach is acceptable, it is like using an entire ship for your goods rather than a simple container from the shipping container analogy. Virtual machines are heavy on resources as you need a heavy guest OS layer to isolate applications rather than something more lightweight. We need to allocate dedicated CPU and memory to a virtual machine; resource sharing is suboptimal since people tend to overprovision virtual machines to cater to peak load. They are also slower to start, and virtual machine scaling is traditionally more cumbersome as multiple moving parts and technologies are involved. Therefore, automating horizontal scaling (handling more traffic from users by adding more machines to the resource pool) using virtual machines is not very straightforward. Also, sysadmins now have to deal with multiple servers rather than numerous libraries and dependencies in one. It is better than before, but it is not optimal from a compute resource point of view.

Containers

This is where containers come into the picture. Containers solve the matrix of hell without involving a heavy guest OS layer between them. Instead, they isolate the application runtime and dependencies by encapsulating them to create an abstraction called containers. Now, you have multiple containers that run on a single OS. Numerous applications running on containers can share the same infrastructure. As a result, they do not waste your computing resources. You also do not have to worry about application libraries and dependencies as they are isolated from other applications – a win-win situation for everyone!

Containers run on container runtimes. While **Docker** is the most popular and more or less the de facto container runtime, other options are available on the market, such as **Rkt** and **Containerd**. They all use the same Linux kernel **cgroups** feature, whose basis comes from the combined efforts of Google, IBM, OpenVZ, and SGI to embed **OpenVZ** into the main Linux kernel. OpenVZ was an early attempt at implementing features to provide virtual environments within a Linux kernel without using a guest OS layer, which we now call containers.

It works on my machine

You might have heard this phrase many times in your career. It is a typical situation where you have erratic developers worrying your test team with “*But, it works on my machine*” answers and your testing team responding with “*We are not going to deliver your machine to the client.*” Containers use the *Build once, run anywhere* and the *Package once, deploy anywhere* concepts and solve the *It works on my machine* syndrome. As containers need a container runtime, they can run on any machine in the same way. A standardized setup for applications also means that the sysadmin’s job is reduced to just taking care of the container runtime and servers and delegating the application’s responsibilities to the development team. This reduces the admin overhead from software delivery, and software development teams can now spearhead development without many external dependencies – a great power indeed! Now, let’s look at how containers are designed to do that.

Container architecture

In most cases, you can visualize containers as mini virtual machines – at least, they seem like they are. But, in reality, they are just computer programs running within an OS. So, let’s look at a high-level diagram of what an application stack within containers looks like:

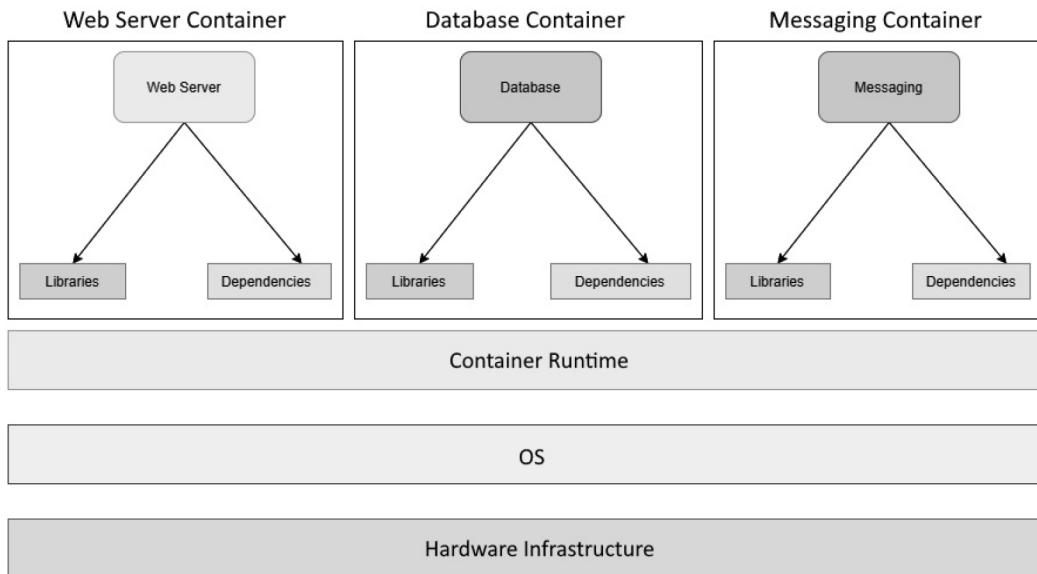


Figure 1.5 – Applications on containers

As we can see, we have the compute infrastructure right at the bottom, forming the base, followed by the host OS and a container runtime (in this case, Docker) running on top of it. We then have multiple containerized applications using the container runtime, running as separate processes over the host operating system using *namespaces* and *cgroups*.

As you may have noticed, we do not have a guest OS layer within it, which is something we have with virtual machines. Each container is a *software program* that runs on the Kernel userspace and shares the same OS and associated runtime and other dependencies, with only the required libraries and dependencies within the container. Containers do not inherit the OS environment variables. You have to set them separately for each container.

Containers replicate the filesystem, and though they are present on disk, they are isolated from other containers. This makes containers run applications in a secure environment. A separate container filesystem means that containers don't have to communicate to and fro with the OS filesystem, which results in faster execution than virtual machines.

Containers were designed to use Linux *namespaces* to provide isolation and *cgroups* to offer restrictions on CPU, memory, and disk I/O consumption.

This means that if you list the OS processes, you will see the container process running alongside other processes, as shown in the following screenshot:

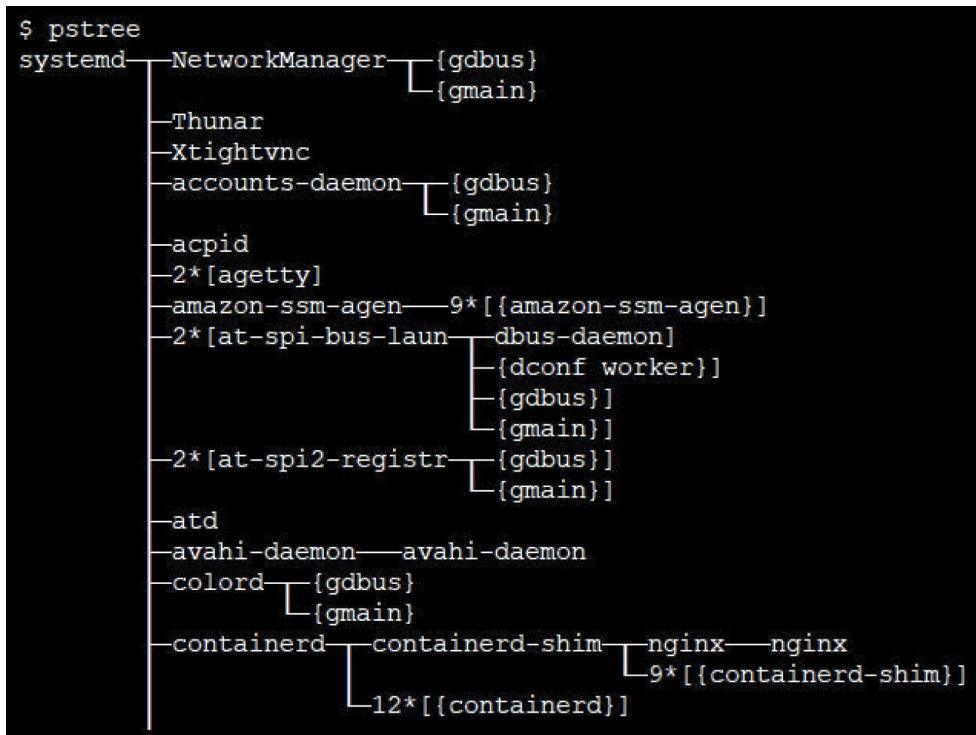


Figure 1.6 – OS processes

However, when you list the container's processes, you will only see the container process, as follows:

```
$ docker exec -it mynginx1 bash
root@4ee264d964f8:/# pstree
nginx---nginx
```

This is how namespaces provide a degree of isolation between containers.

Cgroups play a role in limiting the amount of computing resources a group of processes can use. For example, if you add processes to a cgroup, you can limit the CPU, memory, and disk I/O the processes can use. In addition, you can measure and monitor resource usage and stop a group of processes when an application goes astray. All these features form the core of containerization technology, which we will see later in this book.

Once we have independently running containers, we also need to understand how they interact. Therefore, we'll have a look at container networking in the next section.

Container networking

Containers are separate network entities within the OS. Docker runtimes use network drivers to define networking between containers, and they are software-defined networks. **Container networking** works by using software to manipulate the *host iptables*, connect with external network interfaces, create tunnel networks, and perform other activities to allow connections to and from containers.

While there are various types of network configurations you can implement with containers, it is good to know about some widely used ones. Don't worry too much if the details are overwhelming – you will understand them while completing the hands-on exercises later in this book, and it is not a hard requirement to know all of this to follow the text. For now, let's look at various types of container networks that you can define:

- **None:** This is a fully isolated network, and your containers cannot communicate with the external world. They are assigned a loopback interface and cannot connect with an external network interface. You can use this network to test your containers, stage your container for future use, or run a container that does not require any external connection, such as batch processing.
- **Bridge:** The bridge network is the default network type in most container runtimes, including Docker, and uses the `docker0` interface for default containers. The bridge network manipulates IP tables to provide **Network Address Translation (NAT)** between the container and host network, allowing external network connectivity. It also does not result in port conflicts, enabling network isolation between containers running on a host. Therefore, you can run multiple applications that use the same container port within a single host. A bridge network allows containers within a single host to communicate using the container IP addresses. However, they don't permit communication with containers running on a different host. Therefore, you should not use the bridge network for clustered configuration (using multiple servers in tandem to run your containers).
- **Host:** Host networking uses the network namespace of the host machine for all the containers. It is similar to running multiple applications within your host. While a host network is simple to implement, visualize, and troubleshoot, it is prone to port-conflict issues. While containers use the host network for all communications, it does not have the power to manipulate the host network interfaces unless it is running in privileged mode. Host networking does not use NAT, so it is fast and communicates at bare-metal speeds. Therefore, you can use host networking to optimize performance. However, since it has no network isolation between containers, from a security and management point of view, in most cases, you should avoid using the host network.
- **Underlay:** Underlay exposes the host network interfaces directly to containers. This means you can run your containers directly on the network interfaces instead of using a bridge network. There are several underlay networks, the most notable being MACvlan and IPvlan. MACvlan allows you to assign a MAC address to every container so that your container looks like a physical device. This is beneficial for migrating your existing stack to containers, especially when your application needs to run on a physical machine. MACvlan also provides complete isolation to your host networking, so you can use this mode if you have a strict security requirement.

MACvlan has limitations as it cannot work with network switches with a security policy to disallow MAC spoofing. It is also constrained to the MAC address ceiling of some network interface cards, such as Broadcom, which only allows 512 MAC addresses per interface.

- **Overlay:** Don't confuse overlay with underlay – even though they seem like antonyms, they are not. Overlay networks allow communication between containers on different host machines via a networking tunnel. Therefore, from a container's perspective, they seem to interact with containers on a single host, even when they are located elsewhere. It overcomes the bridge network's limitations and is especially useful for cluster configuration, especially when using a container orchestrator such as Kubernetes or Docker Swarm. Some popular overlay technologies container runtimes and orchestrators use are **flannel**, **calico**, and **VXLAN**.

Before we delve into the technicalities of different kinds of networks, let's understand the nuances of container networking. For this discussion, we'll talk about Docker in particular.

Every Docker container running on a host is assigned a unique IP address. If you `exec` (open a shell session) into the container and run `hostname -I`, you should see something like the following:

```
$ docker exec -it mynginx1 bash
root@4ee264d964f8:/# hostname -I
172.17.0.2
```

This allows different containers to communicate with each other through a simple TCP/IP link. The Docker daemon acts as the DHCP server for every container. Here, you can define virtual networks for a group of containers and club them together to provide network isolation if you desire. You can also connect a container to multiple networks to share it for two different roles.

Docker assigns every container a unique hostname that defaults to the container ID. However, this can be overridden easily, provided you use unique hostnames in a particular network. So, if you `exec` into a container and run `hostname`, you should see the container ID as the hostname, as follows:

```
$ docker exec -it mynginx1 bash
root@4ee264d964f8:/# hostname
4ee264d964f8
```

This allows containers to act as separate network entities rather than simple software programs, and you can easily visualize containers as mini virtual machines.

Containers also inherit the host OS's DNS settings, so you don't have to worry too much if you want all the containers to share the same DNS settings. If you're going to define a separate DNS configuration for your containers, you can easily do so by passing a few flags. Docker containers do not inherit entries in the `/etc/hosts` file, so you must define them by declaring them while creating the container using the `docker run` command.

If your containers need a proxy server, you must set that either in the Docker container's environment variables or by adding the default proxy to the `~/.docker/config.json` file.

So far, we've discussed containers and what they are. Now, let's discuss how containers are revolutionizing the world of DevOps and how it was necessary to spell this outright at the beginning.

Containers and modern DevOps practices

Containers and modern DevOps practices are highly complementary and have transformed how we approach software development and deployment.

Containers have a great synergy with modern DevOps practices as they provide the necessary infrastructure encapsulation, portability, scalability, and agility to enable rapid and efficient software delivery. With modern DevOps practices such as CI/CD, IaC, and microservices, containers form a powerful foundation for organizations to achieve faster time-to-market, improved software quality, and enhanced operational efficiency.

Containers follow DevOps practices right from the start. If you look at a typical container build and deployment workflow, this is what you'll get:

1. First, code your app in whatever language you wish.
2. Then, create a **Dockerfile** that contains a series of steps to install the application dependencies and environment configuration to run your app.
3. Next, use the Dockerfile to create container images by doing the following:
 - a) Build the container image.
 - b) Run the container image.
 - c) Unit test the app running on the container.
4. Then, push the image to a container registry such as **DockerHub**.
5. Finally, create containers from container images and run them in a cluster.

You can embed these steps beautifully in the CI/CD pipeline example shown here:

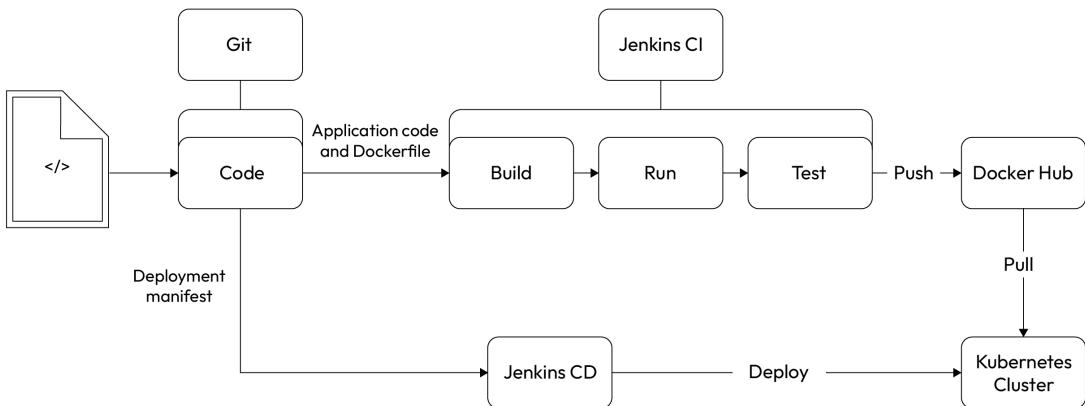


Figure 1.7 – Container CI/CD pipeline example

This means your application and its runtime dependencies are all defined in the code. You follow configuration management from the very beginning, allowing developers to treat containers like ephemeral workloads (ephemeral workloads are temporary workloads that are dispensable, and if one disappears, you can spin up another one without it having any functional impact). You can replace them if they misbehave – something that was not very elegant with virtual machines.

Containers fit very well within modern CI/CD practices as you now have a standard way of building and deploying applications, irrespective of the language you code in. You don't have to manage expensive build and deployment software as you get everything out of the box with containers.

Containers rarely run on their own, and it is a standard practice in the industry to plug them into a container orchestrator such as **Kubernetes** or use a **Container-as-a-Service (CaaS)** platform such as **AWS ECS** and **EKS**, **Google Cloud Run** and **Kubernetes Engine**, **Azure ACS** and **AKS**, **Oracle OCI** and **OKE**, and others. Popular **Function-as-a-Service (FaaS)** platforms such as **AWS Lambda**, **Google Functions**, **Azure Functions**, and **Oracle Functions** also run containers in the background. So, though they may have abstracted the underlying mechanism from you, you may already be using containers unknowingly.

As containers are lightweight, you can build smaller parts of applications into containers to manage them independently. Combine that with a container orchestrator such as Kubernetes, and you get a distributed microservices architecture running with ease. These smaller parts can then scale, auto-heal, and get released independently of others, which means you can release them into production quicker than before and much more reliably.

You can also plug in a **service mesh** (infrastructure components that allow you to discover, list, manage, and allow communication between multiple components (services) of your microservices application) such as **Istio** on top, and you will get advanced Ops features such as traffic management, security, and observability with ease. You can then do cool stuff such as **blue/green deployments** and **A/B testing**, operational tests in production with **traffic mirroring**, **geolocation-based routing**, and much more.

As a result, large and small enterprises are embracing containers quicker than ever, and the field is growing exponentially. According to [businesswire.com](https://www.businesswire.com), the application container market shows a compounded growth of 31% per annum and will reach \$6.9 billion by 2025. The exponential growth of 30.3% per annum in the cloud, expected to reach over \$2.4 billion by 2025, has also contributed to this.

Therefore, modern DevOps engineers must understand containers and the relevant technologies to ship and deliver containerized applications effectively. This does not mean that virtual machines are unnecessary, and we cannot completely ignore the role of IaaS-based solutions in the market, so we will also cover some config management with **Ansible** in further chapters. Due to the advent of the cloud, IaC has been gaining much momentum recently, so we will also cover **Terraform** as an IaC tool.

Migrating from virtual machines to containers

As we see the technology market moving toward containers, DevOps engineers have a crucial task – *migrating applications running on virtual machines so that they can run on containers*. Well, this is in most DevOps engineers' job descriptions and is one of the most critical things we do.

While, in theory, containerizing an application is as simple as writing a few steps, practically speaking, it can be a complicated beast, especially if you are not using config management to set up your virtual machines. Virtual machines that run on current enterprises these days were created from a lot of manual labor by toiling sysadmins, improving the servers piece by piece, and making it hard to reach out to the paper trail of hotfixes they might have made until now.

Since containers follow config management principles from the very beginning, it is not as simple as picking up the virtual machine image and using a converter to convert it into a Docker container.

Migrating a legacy application running on virtual machines requires numerous steps. Let's take a look at them in more detail.

Discovery

First, we start with the discovery phase:

- Understand the different parts of your applications
- Assess what parts of the legacy applications you can containerize and whether it is technically possible to do so
- Define a migration scope and agree on the clear goals and benefits of the migration with timelines

Application requirement assessment

Once the discovery phase is complete, we need to do the application requirement assessment:

- Assess if it is a better idea to break the application into smaller parts. If so, then what would the application parts be, and how will they interact with each other?
- Assess what aspects of the architecture, its performance, and its security you need to cater to regarding your application, and think about the container world's equivalent.
- Understand the relevant risks and decide on mitigation approaches.
- Understand the migration principle and decide on a migration approach, such as what part of the application you should containerize first. Always start with the application with the least amount of external dependencies first.

Container infrastructure design

Container infrastructure design involves creating a robust and scalable environment to support the deployment and management of containerized applications.

Designing a container infrastructure involves considering factors such as scalability, networking, storage, security, automation, and monitoring. It's crucial to align the infrastructure design with the specific requirements and goals of the containerized applications and to follow best practices for efficient and reliable container deployment and management.

Once we've assessed all our requirements, architecture, and other aspects, we can move on to container infrastructure design:

- Understand the current and future scale of operations when you make this decision. You can choose from many options based on your application's complexity. The right questions include; how many containers do we need to run on the platform? What kind of dependencies do these containers have on each other? How frequently are we going to deploy changes to the components? What is the potential traffic the application can receive? What is the traffic pattern on the application?
- Based on the answers you get to the preceding questions, you need to understand what sort of infrastructure you will run your application on. Will it be on-premises or the cloud, and will you use a managed Kubernetes cluster or self-host and manage one? You can also look at options such as CaaS for lightweight applications.
- How will you monitor and operate your containers? Will it require installing specialist tools? Will it require integrating with the existing monitoring tool stack? Understand the feasibility and make an appropriate design decision.
- How will you secure your containers? Are there any regulatory and compliance requirements regarding security? Does the chosen solution cater to them?

Containerizing the application

Containerizing an application involves packaging the application and its dependencies into a container image, which can be deployed and run consistently across different environments.

Containerizing an application offers benefits such as improved portability, scalability, and reproducibility. It simplifies the deployment process and allows for consistent application behavior across different environments.

Once we've considered all aspects of the design, we can now start containerizing the application:

- This is where we look into the application and create a Dockerfile containing the steps to create the container just as it is currently. This requires a lot of brainstorming and assessment, mostly if config management tools don't build your application by running on a virtual machine such as Ansible. It can take a long time to figure out how the application was installed, and you need to write the exact steps for this.
- If you plan to break your application into smaller parts, you may need to build your application from scratch.
- You must decide on a test suite that works on your parallel virtual machine-based application and improve it over time.

Testing

Testing containerized applications is an important step to ensure their functionality, performance, and compatibility.

By implementing a comprehensive testing strategy, you can ensure the reliability, performance, and security of your containerized application. Testing at various levels, integrating automation, and closely monitoring the application's behavior will help you identify and resolve issues early in the development life cycle, leading to a more robust and reliable containerized application.

Once we've containerized the application, the next step in the process is testing:

- To prove whether your containerized application works exactly like the one in the virtual machine, you need to do extensive testing to prove that you haven't missed any details or parts you should have considered previously. Run an existing test suite or the one you created for the container.
- Running an existing test suite can be the right approach, but you also need to consider the software's non-functional aspects. Benchmarking the original application is a good start, and you need to understand the overhead the container solution is putting in. You also need to fine-tune your application so that it fits the performance metrics.

- You also need to consider the importance of security and how you can bring it into the container world. Penetration testing will reveal a lot of security loopholes that you might not be aware of.

Deployment and rollout

Deploying and rolling out a containerized application involves deploying the container images to the target environment and making the application available for use.

Once we've tested our containers and are confident enough, we can roll out our application to production:

- Finally, we roll out our application to production and learn from there if further changes are needed. We then return to the discovery process until we have perfected our application.
- You must define and develop an automated runbook and a CI/CD pipeline to reduce cycle time and troubleshoot issues quickly.
- Doing A/B testing with the container applications running in parallel can help you realize any potential issues before you switch all the traffic to the new solution.

The following diagram summarizes these steps, and as you can see, this process is cyclic. This means that you may have to revisit these steps from time to time based on what you learned from the operating containers in production:

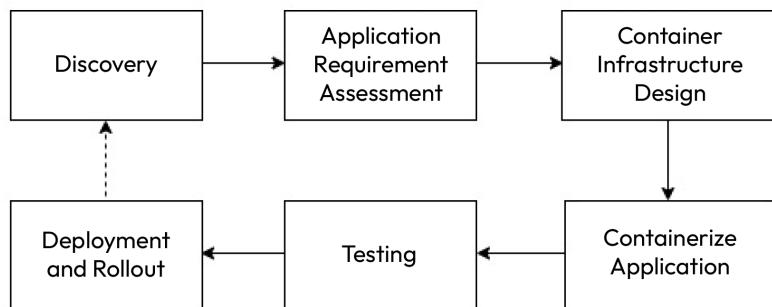


Figure 1.8 – Migrating from virtual machines to containers

Now, let's understand what we need to do to ensure that we migrate from virtual machines to containers with the least friction and also attain the best possible outcome.

What applications should go in containers?

In your journey of moving from virtual machines to containers, you first need to assess what can and can't go in containers. Broadly speaking, there are two kinds of application workloads you can have – **stateless** and **stateful**. While stateless workloads do not store state and are computing powerhouses, such as APIs and functions, stateful applications, such as databases, require persistent storage to function.

Though it is possible to containerize any application that can run on a Linux virtual machine, stateless applications become the first low-hanging fruits you may want to look at. It is relatively easy to containerize these workloads because they don't have storage dependencies. The more storage dependencies you have, the more complex your application becomes in containers.

Secondly, you also need to assess the form of infrastructure you want to host your applications on. For example, if you plan to run your entire tech stack on Kubernetes, you would like to avoid a heterogeneous environment wherever possible. In that scenario, you may also wish to containerize stateful applications. With web services and the middleware layer, most applications rely on some form of state to function correctly. So, in any case, you would end up managing storage.

Though this might open up Pandora's box, there is no standard agreement within the industry regarding containerizing databases. While some experts are naysayers for its use in production, a sizeable population sees no issues. The primary reason is insufficient data to support or disprove using a containerized database in production.

I suggest that you proceed with caution regarding databases. While I am not opposed to containerizing databases, you must consider various factors, such as allocating proper memory, CPU, disk, and every dependency you have on virtual machines. Also, it would help if you looked into the behavioral aspects of the team. If you have a team of DBAs managing the database within production, they might not be very comfortable dealing with another layer of complexity – containers.

We can summarize these high-level assessment steps using the following flowchart:

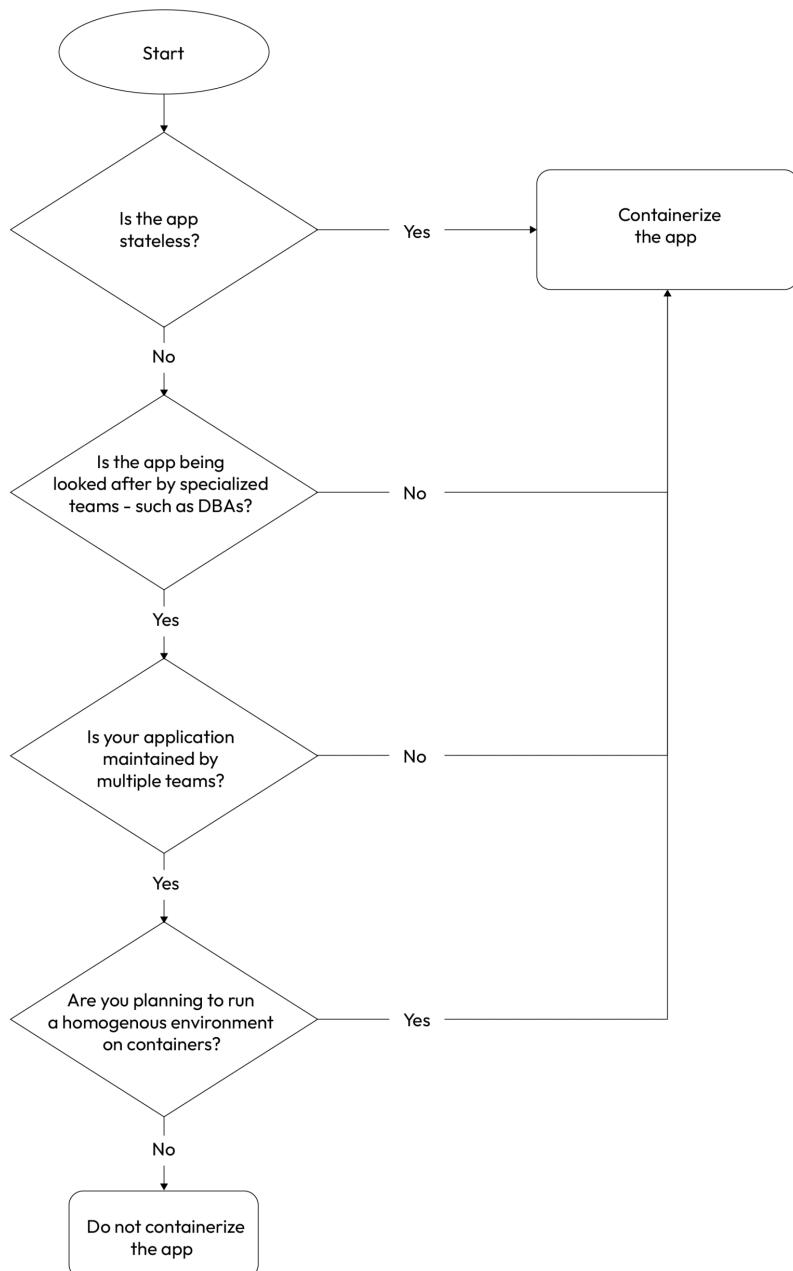


Figure 1.9 – Virtual machine to container migration assessment

This flowchart accounts for the most common factors that are considered during the assessment. You also need to factor in situations that are unique to your organization. So, it is a good idea to take those into account as well before making any decisions.

Let's look at some use cases that are suitable for containerization to get a fair understanding. The following types of applications are commonly deployed using containers:

- **Microservices architecture:** Applications that follow a microservices architecture, where the functionality is divided into small, independent services, are well-suited for containerization. Each microservice can be packaged as a separate container, enabling easier development, deployment, scaling, and management of the individual services.
- **Web applications:** Web applications, including frontend applications, backend APIs, and web services, can be containerized. Containers provide a consistent runtime environment, making it easier to package and deploy web applications across different environments, such as development, testing, and production.
- **Stateful applications:** Containers can also be used to run stateful applications that require persistent data storage. By leveraging container orchestration platforms' features, such as persistent volumes or stateful sets, stateful applications such as databases, content management systems, or file servers can be containerized and managed effectively.
- **Batch processing or scheduled jobs:** Applications that perform batch processing tasks or scheduled jobs, such as data processing, periodic backups, or report generation, can benefit from containerization. Containers provide a controlled and isolated environment for running these jobs, ensuring consistent execution and reproducibility.
- **CI/CD tools:** Containerizing CI/CD tools such as Jenkins, GitLab CI/CD, or CircleCI allows for consistent and reproducible build, test, and deployment pipelines. Containers make it easier to manage dependencies, isolate build environments, and enable rapid deployment of CI/CD infrastructure.
- **Development and testing environments:** Containers are valuable for creating isolated and reproducible development and testing environments. Developers can use containers to package their applications along with the required dependencies, libraries, and development tools. This enables consistent development and testing experiences across different machines and team members.
- **Internet of Things (IoT) applications:** Containers can be used to deploy and manage applications in IoT scenarios. They provide lightweight and portable runtime environments for IoT applications, enabling easy deployment across edge devices, gateways, or cloud infrastructures.
- **Machine learning and data analytics applications:** Containerization is increasingly used to deploy machine learning models and data science applications. Containers encapsulate the necessary dependencies, libraries, and runtime environments, allowing for seamless deployment and scaling of data-intensive applications.

It's important to note that not all applications are ideal candidates for containerization. Applications with heavy graphical interfaces, legacy monolithic architectures tightly coupled to the underlying infrastructure, or applications that require direct hardware access may not be suitable for containerization. Virtual machines or other deployment approaches may be more appropriate in such cases.

Breaking the applications into smaller pieces

You get the most out of containers if you run parts of your application independently of others.

This approach has numerous benefits, as follows:

- You can release your application more often as you can now change a part of your application without this impacting something else; your deployments will also take less time to run.
- Your application parts can scale independently of each other. For example, if you have a shopping app and your *orders* module is jam-packed, it can scale more than the *reviews* module, which may be far less busy. With a monolith, your entire application would scale with traffic, and this would not be the most optimized approach from a resource consumption point of view.
- Something that impacts one part of the application does not compromise your entire system. For example, customers can still add items to their cart and check out orders if the *reviews* module is down.

However, you should also not break your application into tiny components. This will result in considerable management overhead as you will not be able to distinguish between what is what. In terms of the shopping website example, it is OK to have an *order* container, a *reviews* container, a *shopping cart* container, and a *catalog* container. However, it is not OK to have *create order*, *delete order*, and *update order* containers. That would be overkill. Breaking your application into logical components that fit your business is the right way.

But should you break your application into smaller parts as the very first step? Well, it depends. Most people will want to get a **return on investment (ROI)** out of their containerization work. Suppose you do a lift and shift from virtual machines to containers, even though you are dealing with very few variables, and you can go into containers quickly. In that case, you don't get any benefits out of it – especially if your application is a massive monolith. Instead, you would add some application overhead because of the container layer. So, rearchitecting your application to fit in the container landscape is the key to going ahead.

Are we there yet?

So, you might be wondering, are we there yet? Not really! Virtual machines are to stay for a very long time. They have a good reason to exist, and while containers solve most problems, not everything can be containerized. Many legacy systems are running on virtual machines that cannot be migrated to containers.

With the advent of the cloud, *virtualized infrastructure* forms its base, and virtual machines are at its core. Most containers run on virtual machines within the cloud, and though you might be running containers in a cluster of nodes, these nodes would still be virtual machines.

However, the best thing about the container era is that it sees virtual machines as part of a standard setup. You install a container runtime on your virtual machines and do not need to distinguish between them. You can run your applications within containers on any virtual machine you wish. With a container orchestrator such as *Kubernetes*, you also benefit from the orchestrator deciding where to run the containers while considering various factors – resource availability is among the most critical.

This book will look at various aspects of modern DevOps practices, including managing cloud-based infrastructure, virtual machines, and containers. While we will mainly cover containers, we will also look at config management with equal importance using Ansible and learn how to spin up infrastructure with Terraform.

We will also look into modern CI/CD practices and learn how to deliver an application into production efficiently and error-free. For this, we will cover tools such as **Jenkins** and **Argo CD**. This book will give you everything you need to undertake a modern DevOps engineer role in the cloud and container era.

Summary

In this chapter, we understood modern DevOps, the cloud, and modern cloud-native applications. We then looked at how the software industry is quickly moving toward containers and how, with the cloud, it is becoming more critical for a modern DevOps engineer to have the required skills to deal with both. Then, we took a peek at the container architecture and discussed some high-level steps in moving from a virtual machine-based architecture to a containerized one.

In the next chapter, we will look at source code management with **Git**, which will form the base of everything we will do in the rest of this book.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Cloud computing is more expensive than on-premises. (True/False)
2. Cloud computing requires more **Capital Expenditure (CapEx)** than **Operating Expenditure (OpEx)**. (True/False)
3. Which of the following is true about cloud-native applications? (Choose three)
 - A. They typically follow the microservices architecture
 - B. They are typically monoliths
 - C. They use containers

- D. They use dynamic orchestration
 - E. They use on-premises databases
4. Containers need a hypervisor to run. (True/False)
 5. Which of the following statements regarding containers is *not* correct? (Choose one)
 - A. Containers are virtual machines within virtual machines
 - B. Containers are simple OS processes
 - C. Containers use cgroups to provide isolation
 - D. Containers use a container runtime
 - E. A container is an ephemeral workload
 6. All applications can be containerized. (True/False)
 7. Which of the following is a container runtime? (Choose two)
 - A. Docker
 - B. Kubernetes
 - C. Containerd
 - D. Docker Swarm
 8. What kind of applications should you choose to containerize first?
 - A. APIs
 - B. Databases
 - C. Mainframes
 9. Containers follow CI/CD principles out of the box. (True/False)
 10. Which of the following is an advantage of breaking your applications into multiple parts? (Choose four)
 - A. Fault isolation
 - B. Shorter release cycle time
 - C. Independent, fine-grained scaling
 - D. Application architecture simplicity
 - E. Simpler infrastructure
 11. While breaking an application into microservices, which aspect should you consider?
 - A. Breaking applications into as many tiny components as possible
 - B. Breaking applications into logical components

12. What kind of application should you containerize first?
 - A. Stateless
 - B. Stateful
13. Which of the following are examples of CaaS? (Choose three)
 - A. Azure Functions
 - B. Google Cloud Run
 - C. Amazon ECS
 - D. Azure ACS
 - E. Oracle Functions

Answers

1. False
2. False
3. A, C, D
4. False
5. A
6. False
7. A, C
8. A
9. True
10. A, B, C, E
11. B
12. A
13. B, C, D

2

Source Code Management with Git and GitOps

In the previous chapter, we looked at the core concepts of modern DevOps, had an introduction to the cloud, and got a fair understanding of containers. In this chapter, we will understand source code management and one of the modern ways of enabling DevOps with **GitOps**.

In this chapter, we're going to cover the following main topics:

- What is source code management?
- A crash course on Git
- What is GitOps?
- The principles of GitOps
- Why GitOps?
- Branching strategies and GitOps workflow
- Git versus GitOps

Technical requirements

To follow this chapter, you will need access to a Linux-based command line. If you are using macOS, you can use the inbuilt Terminal for all tasks. If you're a Windows user, you must install **GitBash** from <https://git-scm.com/download/win>. We will cover the installation instructions for this in the following sections.

Now, let's start by understanding source code management.

What is source code management?

Software development involves writing code. Code is the only tangible aspect of the software, allowing the software to function. Therefore, you need to store code somewhere to write and make changes to existing software. There are two kinds of code – **source code**, which is written in a high-level language, and **binaries**, which are compiled from the source code. Generally, binaries are nothing but functional applications that execute when we run the software, and source code is the human-readable code written to generate the binary, which is why source code is named as such.

A software development team has multiple members writing software features, so they must collaborate on code. They cannot just write code on silos without understanding how the application works. Sometimes, more than one developer works on the same feature, so they need some place to share their code with their peers. Source code is an asset in itself; therefore, we want to store it securely in a central location while still readily providing access to developers without hampering their work. You will also want to track changes and version them as you might want to know what caused a problem and immediately roll them back. You will also need to persist the history of code to understand what changes were made by whom, and you will want to have a mechanism for source code peer reviews.

As you can see, you would want to manage multiple aspects of source code, and therefore you would use a source code management tool to do so.

A source code management tool helps you manage all aspects of source code. It provides a central location to store your code, version changes and allows multiple developers to collaborate on the same source code. It also keeps a record of all changes through a version history and everything else that we've talked about before. Effective source code management practices improve collaboration; enable efficient development workflows; provide version control, repository management, branching and merging, change tracking, and auditing; and enhance the overall quality and maintainability of software projects. Some popular SCM tools are **Git**, **Subversion**, **Mercurial**, and **CVS**. However, the most popular and de facto standard for SCM is Git. So, let's go ahead and learn about it in the next section.

A crash course on Git

Git is the most popular source code management system available these days, and it has now become mandatory for all developers to learn Git, at least the basic stuff. In this crash course, we will learn about all basic Git operations and build from them in the subsequent chapters.

Git is a distributed version control system. This means that every Git repository is a copy of the original, and you can replicate that to a remote location if needed. In this chapter, we will create and initialize a local Git repository and then push the entire repository to a remote location.

A Git repository in a remote central location is also known as a **remote repository**. From this central repository, all developers sync changes in their local repository, similar to what's shown in the following diagram:

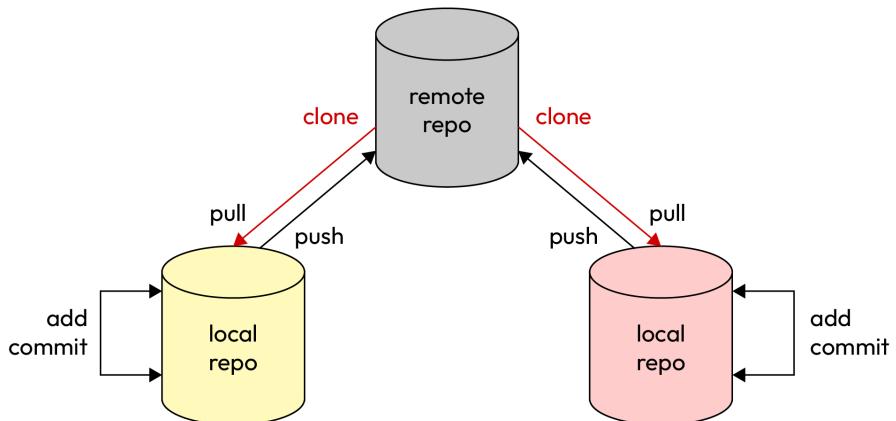


Figure 2.1 – Git distributed repository model

First, let's install Git locally and initialize a local repository. We will look at a remote repository later.

Installing Git

Depending on your platform and workstation, there are different ways to install Git. To install Git on **Ubuntu**, run the following command:

```
$ sudo apt install -y git-all
```

For other OSs and platforms, you can follow the steps at the following link: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

To check if Git has been installed successfully, run the following command:

```
$ git --version  
git version 2.30.2
```

Now, let's initialize our first Git repository.

Initializing your first Git repository

To create a Git repository, you need to create a directory and run the `git init` command, as shown here:

```
$ mkdir first-git-repo && cd first-git-repo/  
$ git init  
Initialized empty Git repository in ~/first-git-repo/.git/
```

You are now ready to use your Git repository. You can also see that when you initialized the Git repository, Git created a hidden directory, `.git`, which it uses to keep track of all changes and commits. Whatever changes you make in your repo, Git keeps them as a delta of changes, which it depicts using `+` and `-` signs. We will look at these in detail in the subsequent sections. For now, let's create a new file within our Git repository and stage it for changes.

Staging code changes

Git allows developers to stage their changes before they commit them. This helps you prepare what you want to commit to the repository. The staging area is a temporary holding area for your changes, and you can add and remove files from the staging area by using the `git add` and `git restore` commands.

Let's create our first file within the local Git repository and stage the changes:

```
$ touch file1
```

Alternatively, you can create a blank file in the `first-git-repo` directory.

Now, we will check if Git can detect the new file that we've created. To do so, we need to run the following command:

```
$ git status
On branch master
No commits yet
Untracked files: (use "git add <file>..." to include in what will be committed)
  file1
nothing added to commit but untracked files present (use "git add" to track)
```

So, as we can see, Git has detected `file1` and is telling us that it is not tracking the file currently. To allow Git to track the file, let's run the following command:

```
$ git add file1
```

Now, let's run `git status` again to see what has changed:

```
$ git status
On branch master
No commits yet
Changes to be committed: (use "git rm --cached <file>..." to unstage)
  new file:   file1
```

As we can see, Git now shows `file1` as a new file in the staging area. You can continue making changes, and when you are done, you can commit the changes using the following command:

```
$ git commit -m "My first commit"
[master (root-commit) cecfb61] My first commit
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 file1
```

Git has now recorded a commit with your changes. Now, let's look at its status again using the following command:

```
$ git status  
On branch master  
nothing to commit, working tree clean
```

Git is now reporting that the working tree is clean, and there is nothing to commit. It also shows that there are no untracked files. Now, let's change `file1` and add some text to it:

```
$ echo "This is first line" >> file1  
$ cat file1  
This is first line
```

`file1` now contains the first line. Let's go ahead and commit this change:

```
$ git add file1  
$ git commit -m "My second commit"  
[master 4c55cf5] My second commit  
1 file changed, 1 insertion(+)
```

As we can see, Git is now reporting that one file has changed, and there is one insertion. Remember when we discussed that Git only tracks the delta changes between commits? That is what is happening here.

In the introduction, we mentioned that Git provides a history of all commits. Let's look at how we can display this history.

Displaying commit history

Git keeps a history of all commits. To see a list of all commits that you've done, you can use the following command:

```
$ git log  
commit 275d24c62a0e946b8858f562607265c269ec5484 (HEAD -> master)  
Author: Gaurav Agarwal <example@gmail.com>  
Date:   Wed Apr 19 12:27:13 2023 +0530  
      My second commit  
commit cecfb61b251f9966f50a4d8bb49742b7af014da4  
Author: Gaurav Agarwal <example@gmail.com>  
Date:   Wed Apr 19 12:20:02 2023 +0530  
      My first commit
```

As we can see, Git has displayed the history of both our commits. Notice that Git marks every commit with a commit ID. We can also delve into what changes were made in the commit by using the `git diff <first_commit_id> <second_commit_id>` command, as follows:

```
$ git diff cecfb61b251f9966f50a4d8bb49742b7af014da4 \  
275d24c62a0e946b8858f562607265c269ec5484  
diff --git a/file1 b/file1
```

```
index e69de29..0cbcf32 100644
--- a/file1
+++ b/file1
@@ -0,0 +1 @@
+This is first line
```

The output clearly shows that the second commit has added `This is first line` within `file1`.

You've suddenly realized that you needed to add another line to `file1` and wanted to do so in the same commit. We can do this by amending the commit. We'll look at this in the next section.

Amending the last commit

It is a best practice to have a single commit for your changes to a particular feature. This helps you track the changes better and makes it easier for the reviewer to review them. In turn, it is cleaner to visualize and manage. However, committing frequently is also a best practice so that your changes are not lost. Fortunately, Git allows you to add changes to the last commit.

To demonstrate this, let's change `file1` and add another line:

```
$ echo "This is second line" >> file1
$ cat file1
This is first line
This is second line
```

Now, let's add the changes to the previous commit using the following commands:

```
$ git add file1
$ git commit --amend
```

Once you run this command, Git will show you a prompt, allowing you to amend the commit message if you like. It will look something like the following:

```
My second commit
# Please enter the commit message for your changes. Lines
# starting with # will be ignored and an empty message aborts the commit
# Date Wed Apr 19 12:27:13 2023 +0530
# on branch master
# Changes to be committed
#   modified: file1
#
```

Save this file (use `ESC :wq` for Vim). This should amend the last commit with the changes. You should get the following output:

```
Date: Wed Apr 19 12:27:13 2023 +0530
1 file changed, 2 insertions(+)
```

When Git amends a commit, you can no longer refer to the previous commit with the same commit ID. Instead, Git generates a separate SHA-1 id for the amended commit. So, let's look at the logs to see this for ourselves:

```
$ git log
commit d11c13974b679b1c45c8d718f01c9ef4e96767ab (HEAD -> master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 12:27:13 2023 +0530

    My second commit
commit cecfb61b251f9966f50a4d8bb49742b7af014da4
Author: Gaurav Agarwal <example@gmail.com>
Date:   Wed Apr 19 12:20:02 2023 +0530

    My first commit
```

Now, let's run the `diff` command again and see what it is reporting:

```
$ git diff cecfb61b251f9966f50a4d8bb49742b7af014da4 \
d11c13974b679b1c45c8d718f01c9ef4e96767ab
diff --git a/file1 b/file1
index e69de29..655a706 100644
--- a/file1
+++ b/file1
@@ -0,0 +1,2 @@
+This is first line
+This is second line
```

The output clearly shows that the second commit has added `This is first line`, as well as `This is second line`, within `file1`. With that, you've successfully amended a commit.

Local repositories are as good as keeping files on your system. However, since you need to share your code with others and keep it secure from laptop OS crashes, theft, physical damage, and more, you need to push your code into a remote repository. We'll look at remote repositories in the next section.

Understanding remote repositories

Remote repositories are replicas of the Git repository at a central location for multiple people to access. This allows your developers to work on the same code base simultaneously and provides you with a backup of your code. There are various tools you can use to host your remote repositories. Notable ones include **GitHub**, **Bitbucket**, and **Gerrit**. You can install them on your on-premises or cloud servers or use a **Software-as-a-Service (SaaS)** platform to store them online. In this book, we are going to focus on GitHub.

GitHub is a web-based platform that helps developers collaborate on code. It is based on Git and allows you to host remote Git repositories. It was founded in 2008 and was acquired by Microsoft in 2018. It is one of the most popular open-source SaaS-based Git repository services and contains almost all open-source code available worldwide.

Before we can create our first remote repo, we must go to <https://github.com/signup> to create an account.

Once you've created an account, we can go ahead and create our first remote Git repository.

Creating a remote Git repository

Creating a remote Git repository is simple on GitHub. Go to <https://github.com/new>, set **Repository Name** to `first-git-repo`, keep the rest of the fields as-is, and click the **Create Repository** button.

Once you've done that, GitHub will provide you with some steps that you can follow to connect with your remote repository. Before we go into any of that, we want to configure some authentication for our local Git command line to interact with the remote repository. Let's take a look.

Setting up authentication with the remote Git repository

Some of the ways you can authenticate with your remote Git repository are as follows:

- **HTTPS:** In this mode, Git uses HTTPS to connect with the remote Git repository. We need to create an HTTPS token within our GitHub account and use this token as a password to authenticate with the remote repository. This process requires you to key in your token every time you authenticate with Git; therefore, it is not a convenient option.
- **SSH:** In this mode, Git uses the SSH protocol to connect with the remote Git repository. While using SSH, we do not need to use a password to authenticate; instead, we must add the **public key** of an **SSH key pair** we can generate from the Linux (or Windows if you're using Git Bash) command line to the GitHub account. This process is more secure as well as convenient.

So, let's set up SSH-based authentication with our remote Git repository.

First, we must generate the SSH key pair within our local system. Go to your Terminal and run the following command to generate an SSH key pair:

```
$ ssh-keygen -t rsa  
Generating public/private rsa key pair.
```

You will be prompted for other details. Keep pressing *Enter* until you reach the prompt again.

Once the key pair has been generated, copy the public key present in the `~/.ssh/id_rsa.pub` file.

Then, go to <https://github.com/settings/ssh/new>, paste the public key in the **Key** field, and click the **Add SSH Key** button. We are now ready to connect with the remote Git repository. Now, let's look at the configurations we must do on our local repository to connect with the remote repository.

Connecting the local repository to the remote repository

You will need to add a remote entry using the following command to connect with the remote repository from the local repository:

```
$ git remote add origin git@github.com:<your-github-username>/first-git-repo.git
```

You can also find these details on the **Quick Setup** page of your GitHub repository.

Now that we've set up the connection, let's look at how we can push our changes to the remote repository.

Pushing changes from the local repository to the remote repository

To push the changes from the local repository to the remote repository, use the following command:

```
$ git push -u origin master
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 474 bytes | 474.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com: <your-github-username>/first-git-repo.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Now, refresh the page on your remote repository. You should see that the code was synced, as shown in the following screenshot:

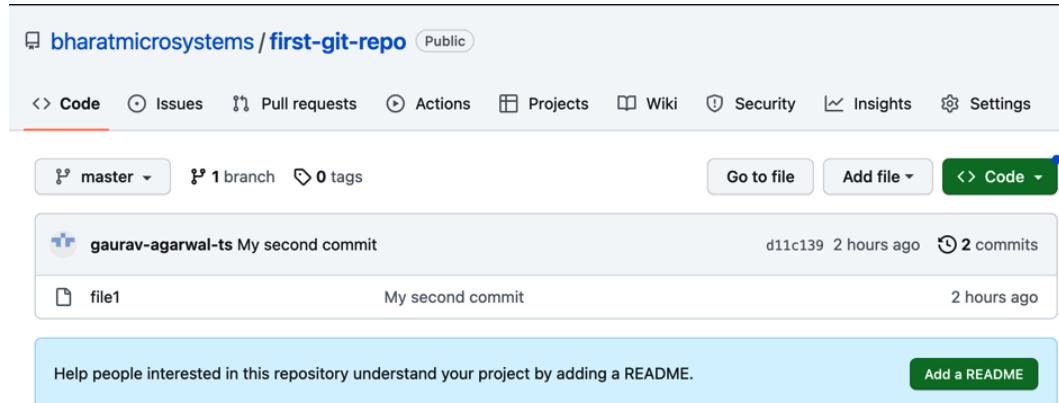


Figure 2.2 – Code synced in the remote repository

You can also use the inline editor to make further changes to the file using the GitHub web portal. While this is not recommended, we'll do this to simulate a situation where another developer changed the same file you were working on.

Click on **file1** and then click on the **pencil** icon to edit the file, as shown in the following screenshot:

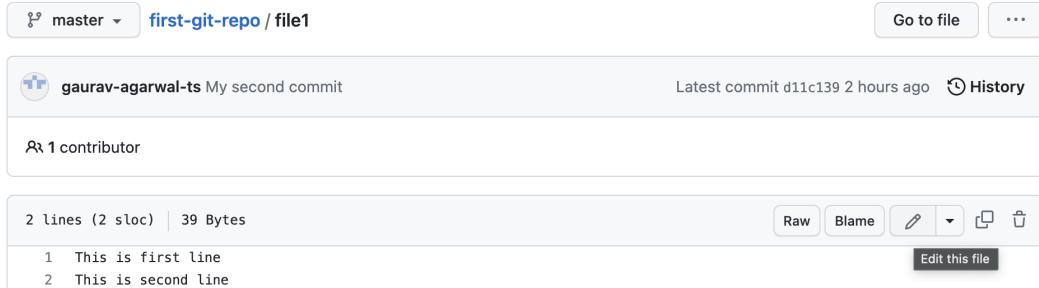


Figure 2.3 – Editing the file in the remote repository

Upon doing this, an editing window will open where you can make changes to the file. Let's add `This is third line` within the file, as shown in the following screenshot:



Figure 2.4 – Adding a new line

Scroll down – you should see a **Commit changes** section, where you can add a commit message field and click on the **Commit** button, as shown in the following screenshot:

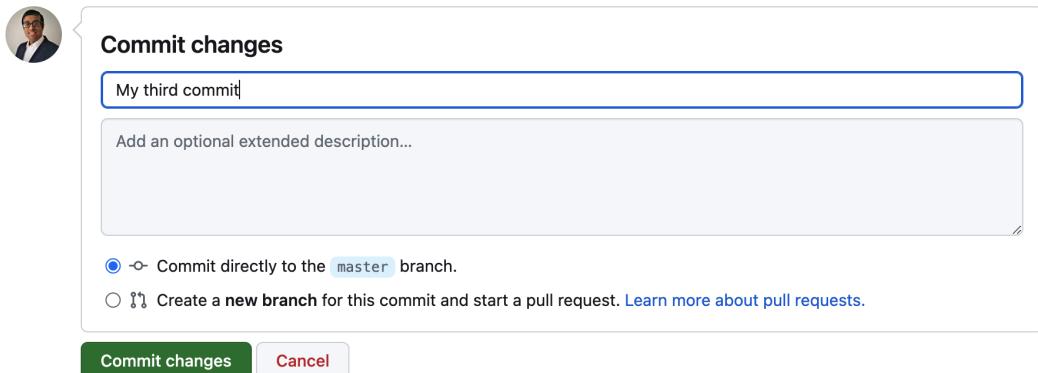


Figure 2.5 – The Commit changes section

Once you've clicked on that button, you should see the third line, as shown in the following screenshot:



Figure 2.6 – Changes committed remotely

At this point, changes have been made to the remote repository, but you have also been working on your changes. To simulate this, let's make a change in the local repository as well using the following commands:

```
$ echo "This is fourth line" >> file1
$ cat file1
This is first line
This is second line
This is fourth line
$ git add file1
$ git commit -m "Added fourth line"
[master e411e91] Added fourth line
 1 file changed, 1 insertion(+)
```

Now that we've committed the changes in our local repository, let's try to push the code to the remote repository using the following commands:

```
$ git push
To github.com:<your-github-username>/first-git-repo.git
 ! [rejected]          master -> master (fetch first)
error: failed to push some refs to 'github.com:<your-github-username>/first-git-repo.git'
hint: Updates were rejected because the remote contains work that you do not have locally.
This is usually caused by another repository pushing to the same ref. You may want to
first integrate the remote changes.
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Wait, what happened? Well, the remote repository rejected the changes as we tried to push changes while someone else made some commits in the remote repository, and our changes are not current. We would need to pull the changes in our local repository first to apply our changes on top of the existing ones in the remote repository. We'll look at this in the next section.

Pulling and rebasing your code

Pulling code involves downloading up-to-date code from the remote to your local repository. **Rebasing** means applying your changes on top of the latest remote commit. It is a best practice to pull and rebase your changes on top of what already exists in the remote repository.

Let's do so using the following command:

```
$ git pull --rebase
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 652 bytes | 130.00 KiB/s, done.
From github.com:<your-github-username>/first-git-repo
  d11c139..f5fb7620 master      -> origin/master
Auto-merging file1
CONFLICT (content): Merge conflict in file1
error: could not apply e411e91... Added fourth line
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply e411e91... Added fourth line
```

Now, we have another issue: we are unable to rebase our commit as we've tried to modify a file that has been modified by someone else. Git wants us to check the file and make appropriate changes so that the changes are applied correctly. This situation is known as a **merge conflict**. Git also provides us with the file that contains the conflict. Let's open the file with a text editor and make the appropriate changes.

The current file looks like this:

```
This is first line
This is second line
<<<<< HEAD
This is third line
=====
This is fourth line
>>>> e411e91 (Added fourth line)
```

The portion depicted by HEAD is the line in the remote repository and shows the recent changes made remotely. The e411e91 commit shows the changes that we made locally. Let's change the file to the following and save it:

```
This is first line
This is second line
This is third line
This is fourth line
```

Now, let's add the file to the staging area and continue the rebase using the following commands:

```
$ git add file1
$ git rebase --continue
[detached HEAD 17a0242] Added fourth line
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/master.
```

Now that we've rebased the changes, let's look at the status of the Git repo by running the following command:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

As we can see, we've added a single commit that we need to push to the remote repository. Let's do that now using the following command:

```
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 297 bytes | 148.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:<your-github-username>/first-git-repo.git
 f5b7620..17a0242 master -> master
```

And this time, the push worked successfully.

In most cases, you would normally need to take a copy of the main code and make changes to it to test new features. You might also want someone to review your changes before they are merged into the main code base. Git allows you to manage that by using Git branches. We'll look at Git branches in the next section.

Git branches

A **Git branch** is a copy of the code base (from where the branch is created) that you can independently modify and work on without affecting the main code base. You will want to create branches while working on new features to ensure that you are not affecting the main branch, which contains reviewed code. Most technology companies normally have several environments where you have code deployed in various stages. For example, you might have a **development environment** where you test your features, a **staging environment** where you integrate all features and test the complete application, and a **production environment** where the application that your end users access resides. So, there would be a possibility that you might have additional environment-specific branches where code deployed on those branches reside. In the following sections of this chapter, we will talk about GitOps, which works on this fundamental principle. For now, let's look at how we can create and manage Git branches.

Creating and managing Git branches

To create a Git branch, you must be on the branch from where you want to branch your code. As in our example repo, we were working on the master branch. Let's stay there and create a feature branch out of that.

To create the branch, run the following command:

```
$ git branch feature/feature1
```

As we can see, the feature branch has been created. To check what branch we are on now, we can use the `git branch` command. Let's do that now:

```
$ git branch
```

And as we see by the * sign over the `master` branch, we are still on the `master` branch. The good thing is that it also shows the `feature/feature1` branch in the list. Let's switch to the feature branch now by using the following command:

```
$ git checkout feature/feature1
Switched to branch 'feature/feature1'
```

Now, we are on the `feature/feature1` branch. Let's make some changes to the `feature/feature1` branch and commit it to the local repo:

```
$ echo "This is feature 1" >> file1
$ git add file1
```

```
$ git commit -m "Feature 1"
[feature/feature1 3fa47e8] Feature 1
 1 file changed, 1 insertion(+)
```

As we can see, the code is now committed to the `feature/feature1` branch. To check the version history, let's run the following command:

```
$ git log
commit 3fa47e8595328eca0bc7d2ae45b3de8d9fd7487c (HEAD -> feature/feature1)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Fri Apr 21 11:13:20 2023 +0530
    Feature 1
commit 17a02424d2b2f945b479ab8ba028f3b535f03575 (origin/master, master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 15:35:56 2023 +0530
    Added fourth line
```

As we can see, the `Feature 1` commit is shown in the Git logs. Now, let's switch to the `master` branch and run the same command again:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$ git log
commit 17a02424d2b2f945b479ab8ba028f3b535f03575 (HEAD -> master, origin/master)
Author: Gaurav Agarwal <gaurav.agarwal@example.com>
Date:   Wed Apr 19 15:35:56 2023 +0530
    Added fourth line
commit f5b7620e522c31821a8659b8857e6fe04c2f2355
Author: Gaurav Agarwal <><your-github-username>@gmail.com>
Date:   Wed Apr 19 15:29:18 2023 +0530
    My third commit
```

As we can see here, the `Feature 1` commit changes are absent. This shows that both branches are now isolated (and have now diverged). Now, the changes are locally present and are not in the remote repository yet. To push the changes to the remote repository, we will switch to the `feature/feature1` branch again. Let's do that with the following command:

```
$ git checkout feature/feature1
Switched to branch 'feature/feature1'
```

Now that we've switched to the feature branch, let's push the branch to the remote repository using the following command:

```
$ git push -u origin feature/feature1
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 286.00 KiB/s, done.
```

```
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature/feature1' on GitHub by visiting:
remote:     https://github.com/<your-github-username>/first-git-repo/pull/new/feature/
feature1
remote:
To github.com:<your-github-username>/first-git-repo.git
 * [new branch]      feature/feature1 -> feature/feature1
Branch 'feature/feature1' set up to track remote branch 'feature/feature1' from 'origin'.
```

With that, we've successfully pushed the new branch to the remote repository. Assuming the feature is ready, we want the changes to go into the master branch. For that, we would have to raise a pull request. We'll look at pull requests in the next section.

Working with pull requests

A **pull request** is a request for merging a **source branch** to a **base branch**. The base branch is the branch where the reviewed code resides. In this case, our base branch is `master`. Pull requests are generally useful for developers to get their code peer reviewed before they merge it with the *fair* version of the code. The reviewer generally checks the quality of the code, whether best practices are being followed, and whether coding standards are appropriate. If the reviewer is unhappy, they might want to flag certain sections of the changes and request modifications. There are normally multiple cycles of reviews, changes, and re-reviews. Once the reviewer is happy with the changes, they can approve the pull request, and the requester can merge the code. Let's take a look at this process:

1. Let's try to raise a pull request for merging our code from the `feature/feature1` branch to the `master` branch. To do so, go to your GitHub repo, select **Pull requests**, and click on the **New pull request** button, as shown in the following screenshot:

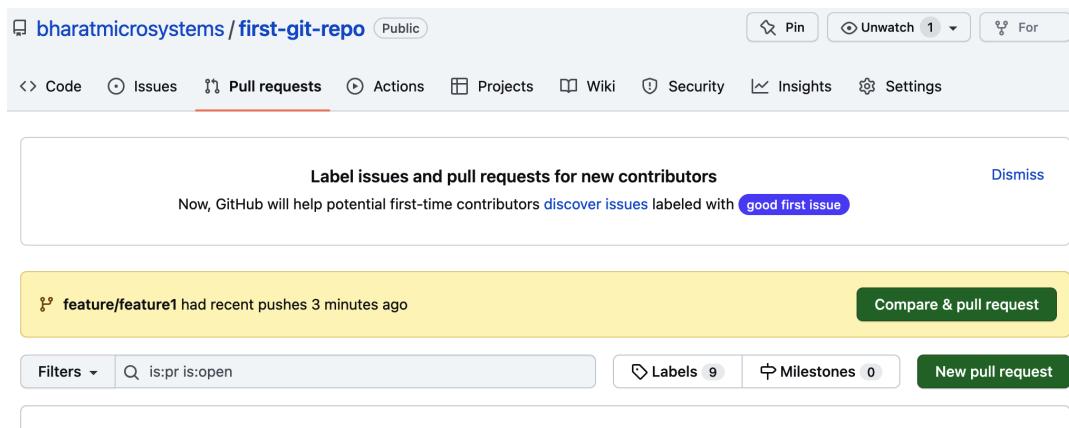


Figure 2.7 – New pull request

2. Keep **base** set to **master** and, in the **compare** dropdown, select **feature/feature1**:

The screenshot shows a GitHub repository page for 'bharatmicrosystems / first-git-repo'. A comparison is being made between the 'master' branch (selected in the dropdown) and the 'feature/feature1' branch. The comparison results show that the 'feature/feature1' branch has one commit, one file changed, and one contributor. The commit details show a file named 'file1' with the following diff:

```
diff --git a/file1 b/file1
@@ -2,3 +2,4 @@
 2 This is second line
 3 This is third line
 4 This is fourth line
+5 + This is feature 1
```

Figure 2.8 – Comparing changes

3. As you can see, it shows you all the changes we've made on the **feature/feature1** branch. Click on the **Create pull request** button to create the pull request. On the next page, stick to the defaults and click on the **Create pull request** button:

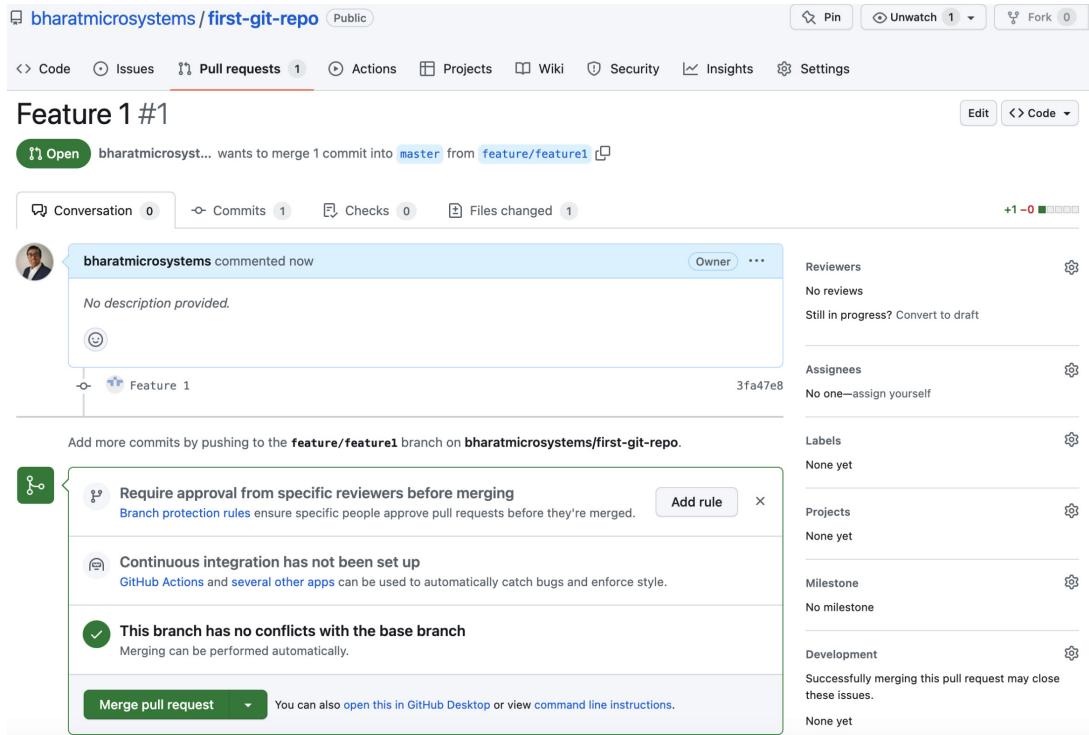


Figure 2.9 – Pull request created

4. As you can see, the pull request was created successfully. Here, you can assign a reviewer and get the code reviewed. Once the reviewer approves the code, you can merge the code to the master branch. For now, let's click on the **Merge pull request** button, followed by the **Confirm merge** button, which should merge the pull request.
5. Now, check if the master branch shows the changes within GitHub. If it does, switch to the master branch and pull the changes into your local repository. You should see the changes in your local repository as well.

I leave this to you as an exercise.

This was a crash course on Git to help you get started. Now, let's move on and understand GitOps, which uses Git as a single source of truth to spin up virtually anything within your application and infrastructure.

What is GitOps?

GitOps is a method that involves implementing DevOps so that Git forms the single source of truth. Instead of maintaining a long list of scripts and tooling to support this, GitOps focuses on writing declarative code for everything, including the infrastructure, configuration, and application code. This means you can spin anything out of thin air by simply using the Git repository. The idea is that you declare what you need in your Git repository, and there is tooling behind the scenes that ensures the desired state is always maintained in the running application and infrastructure surrounding it. The code to spin up the tooling also resides in Git, and you don't have anything outside of Git. This means everything, including the tooling, is automated in this process.

While GitOps also enables DevOps within the organization, it primarily focuses on using Git to manage infrastructure provisioning and application software deployments. DevOps is a broad term that contains a set of principles, processes, and tools to enable developers and operations teams to work seamlessly and shorten the development life cycle, with an end goal to deliver better software more quickly using a CI/CD cycle. While GitOps relies heavily on Git and its features and always looks to Git for versioning, finding configuration drift, and only applying deltas, DevOps is, as such, agnostic of any tool and focuses more on the concepts and processes. Therefore, you can implement DevOps without using Git, but you cannot implement GitOps without Git. Put simply, GitOps implements DevOps, but the reverse may not always be true.

Why GitOps?

GitOps provides us with the following benefits:

- **It deploys better software more quickly:** GitOps offers simplicity in delivering software. You don't have to worry about what tool you need for the deployment type. Instead, you can commit your changes in Git, and the behind-the-scenes tooling automatically deploys it.
- **It provides faster recovery from errors:** If you happen to make an error in deployment (for example, a wrong commit), you can easily roll it back using `git revert` and restore your environment. The idea is that you don't need to learn anything else apart from Git to do a rollout or a rollback.
- **It offers better credential management:** With GitOps, you don't need to store your credentials in different places for your deployments to work. You simply need to provide the tooling access to your Git repository and the binary repository, and GitOps will take care of the rest. You can keep your environment completely secure by restricting your developers' access to it and providing them access to Git instead.
- **Deployments are self-documenting:** Because everything is kept within Git, which records all commits, the deployments are automatically self-documenting. You can know exactly who deployed what at what time by simply looking at the commit history.

- **It promotes shared ownership and knowledge:** As Git forms the single source of truth for all code and configurations within the organization, teams have a single place to understand how things are implemented without ambiguity and dependency on other team members. This helps promote the shared ownership of the code and knowledge within the team.

Now that we know about the benefits of GitOps, let's look at its key principles.

The principles of GitOps

GitOps has the following key principles:

- **It describes the entire system declaratively:** Having declarative code forms the first principle of GitOps. This means that instead of providing instructions on how to build your infrastructure, applying the relevant configuration, and deploying your application, you declare the end state of what you need. This means that your Git repository always maintains a single source of truth. As declarative changes are idempotent, you don't need to worry about the state of your system as this will eventually become consistent with the code in Git.
- **It versions desired system state using Git:** As Git forms an excellent version control system, you don't need to worry too much about how to roll out and roll back your deployments. A simple Git commit means a new deployment, and a Git revert means a rollback. This means you do not need to worry about anything apart from ensuring that the Git repository reflects what you need.
- **It uses tooling to automatically apply approved changes:** As you've stored everything within Git, you can then use tooling that looks for changes within the repository and automatically applies them to your environment. You can also have several branches that apply changes to different environments, along with a pull request-based approval and gating process so that only approved changes end up in your environment.
- **It uses self-healing agents to alert and correct any divergence:** We have the tooling to automatically apply any changes in Git to the environment. However, we also require self-healing agents to alert us of any divergence from the repository. For example, suppose someone deletes a container manually from the environment but doesn't remove it from the Git repository. In that scenario, the agent should alert the team and recreate the container to correct the divergence. This means there is no way to bypass GitOps, and Git remains the single source of truth.

Implementing and living by these principles is simple with modern DevOps tools and techniques, and we will look at practically implementing them later in *Chapters 11* and *12*. In this chapter, however, we'll examine their design principles using a branching strategy and GitOps workflow.

Branching strategies and the GitOps workflow

GitOps requires at least two kinds of Git repositories to function: the **application repository**, which is from where your builds are triggered, and the **environment repository**, which contains all of the infrastructure and **configuration as code (CaC)**. All deployments are driven from the environment repository, and the changes to the code repository drive the deployments. GitOps follows two primary kinds of deployment models: the **push model** and the **pull model**. Let's discuss each of them.

The push model

The push model pushes any changes that occur within your Git repository to the environment. The following diagram explains this process in detail:

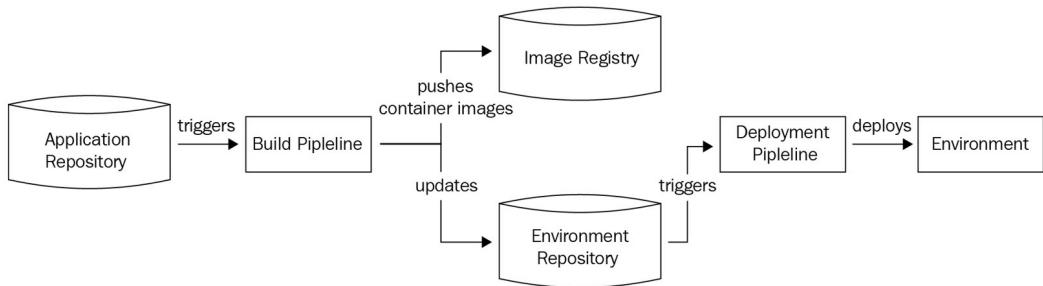


Figure 2.10 – The push model

The push model is inherently unaware of the existing configuration and reacts only to changes made to your Git repositories. Therefore, you will need to set up some form of monitoring to understand whether there are any deviations. Additionally, the push model needs to store all environment credentials within the tools. This is because it interacts with the environment and has to manage the deployments. Typically, we use **Jenkins**, **CircleCI**, or **Travis CI** to implement the push model. While the push model is not recommended, it becomes inevitable in cloud provisioning with **Terraform**, or config management with **Ansible**, as they are both push-based models. Now, let's take a closer look at the pull model.

The pull model

The pull model is an *agent-based deployment model* (also known as an *operator-based deployment model*). An *agent* (or *operator*) within your environment monitors the Git repository for changes and applies them as and when needed. The operator constantly compares the existing configuration with the configuration in the environment repository and applies changes if required. The following diagram shows this process in detail:

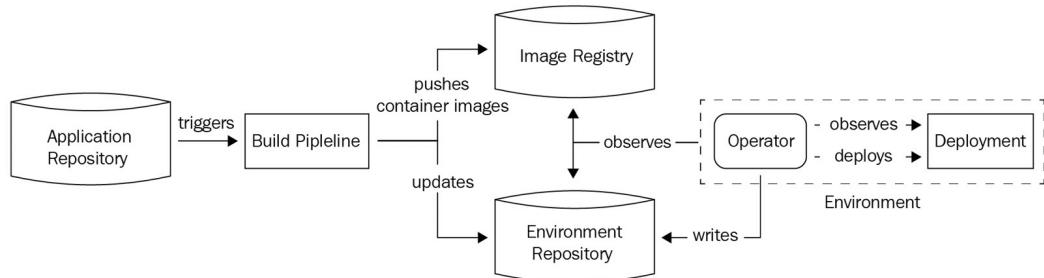


Figure 2.11 – The pull model

The advantage of the pull model is that it monitors and reacts to environment changes alongside repository changes. This ensures that any changes that do not match the Git repository are reverted from the environment. It also alerts the operations team about anything it could not fix using mail notifications, ticketing tools, or Slack notifications. Because the operator lives within the same environment where the code is deployed, we do not need to store credentials within the tools. Instead, they live securely within the environment. You can also live without storing any credentials at all with tools such as Kubernetes, where you can employ **role-based access control (RBAC)** and service accounts for the operator managing the environment.

Tip

When choosing a GitOps model, the best practice is to check whether you can implement a pull-based model instead of a push-based model. Implement a push-based model only if a pull-based model is not possible. It is also a good idea to implement polling in the push-based model by scheduling something, such as a cron job, that will run the push periodically to ensure there is no configuration drift.

We cannot solely live with one model or the other, so most organizations employ a **hybrid model** to run GitOps. This hybrid model combines push and pull models and focuses on using the pull model. It uses the push model when it cannot use the pull model. Now, let's understand how to structure our Git repository so that it can implement GitOps.

Structuring the Git repository

To implement GitOps, we require at least two repositories: the **application repository and the environment repository**. This does not mean that you cannot combine the two, but for the sake of simplicity, let's take a look at each of them separately.

The application repository

The application repository stores the application code. It is a repository in which your developers can actively develop the product that you run for your business. Typically, your builds result from

this application code, and they end up as containers (if we use a container-based approach). Your application repository may or may not have environment-specific branches. Most organizations keep the application repository independent of the environment and focus on building semantic code versions using a branching strategy. Now, there are multiple branching strategies available to manage your code, such as **Gitflow**, **GitHub flow**, and any other branching strategy that suits your needs.

Gitflow is one of the most popular branching strategies that organizations use. That said, it is also one of the most complicated ones as it requires several kinds of branches (for instance, master, hotfixes, release branches, develop, and feature branches) and has a rigid structure. The structure of Gitflow is shown in the following diagram:

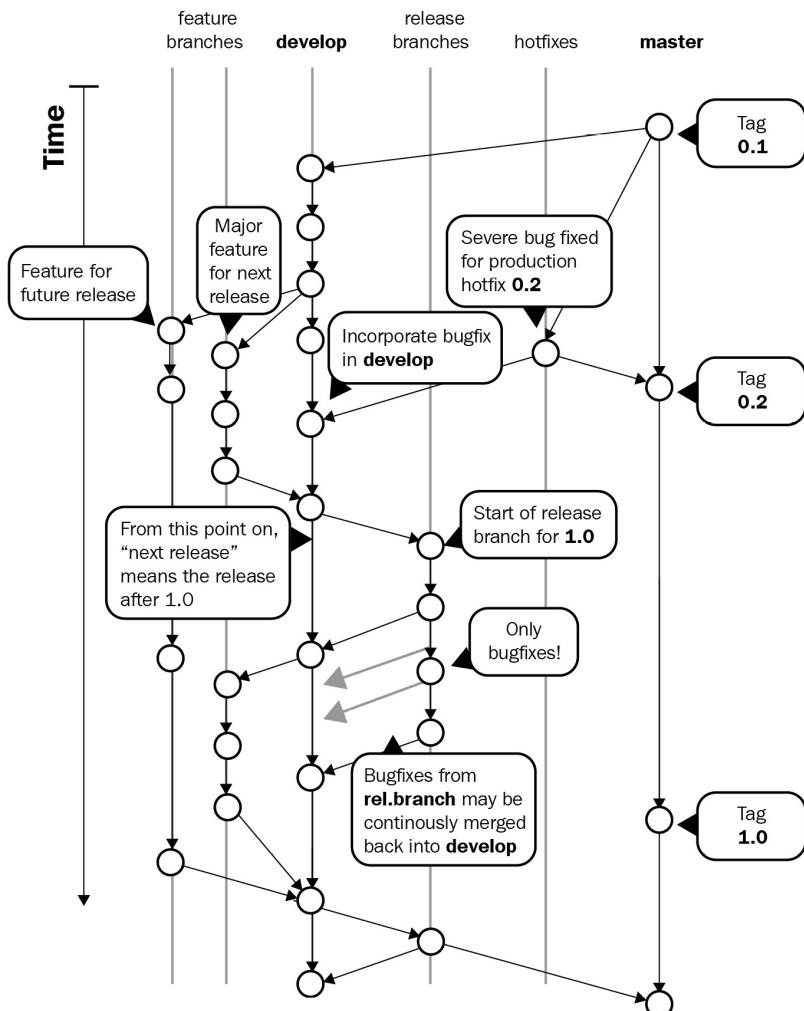


Figure 2.12 – Gitflow structure

A simplified way of doing things is using GitHub flow. It employs fewer branches and is easier to maintain. Typically, it contains a single master branch and many feature branches that eventually merge with the master branch. The master branch always has software that is ready to be deployed to the environments. You tag and version the code in the master branch, pick and deploy it, test it, and then promote it to higher environments. The following diagram shows GitHub flow in detail:

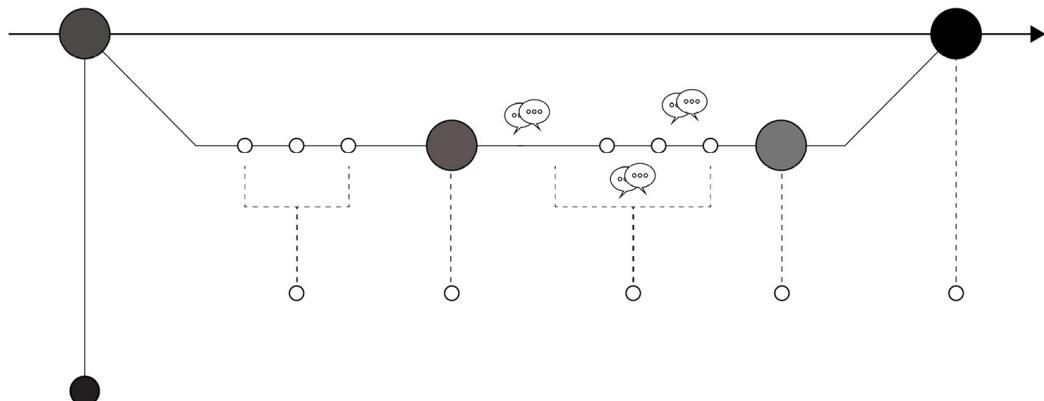


Figure 2.13 – GitHub flow

Note that you are free to create your branching strategy according to your needs and what works for you.

Tip

Choose Gitflow if you have a large team, a vast monolithic repository, and multiple releases running in parallel. Choose GitHub flow if you work for a fast-paced organization that releases updates several times a week and doesn't use the concept of parallel releases. GitHub flow also typically works for microservices where changes are minor and quick.

Typically, application repositories do not have to worry too much about environments; they can focus more on creating deployable software versions.

The environment repository

The environment repository stores the environment-specific configurations needed to run the application code. Therefore, they will typically have **Infrastructure as Code (IaC)** in the form of Terraform scripts, CaC in the form of Ansible playbooks, or Kubernetes manifests that typically help deploy the code we've built from the application repository.

The environment repository should follow an environment-specific branching strategy where a branch represents a particular environment. You can have pull request-based **gating** for these kinds of scenarios. Typically, you build your **development environments** from a development branch and then raise a

pull request to merge the changes to a staging branch. From the staging branch to production, your code progresses with environments. If you have 10 environments, you might end up with 10 different branches in the environment repository. The following diagram showcases the branching strategy you might want to follow for your environment repository:

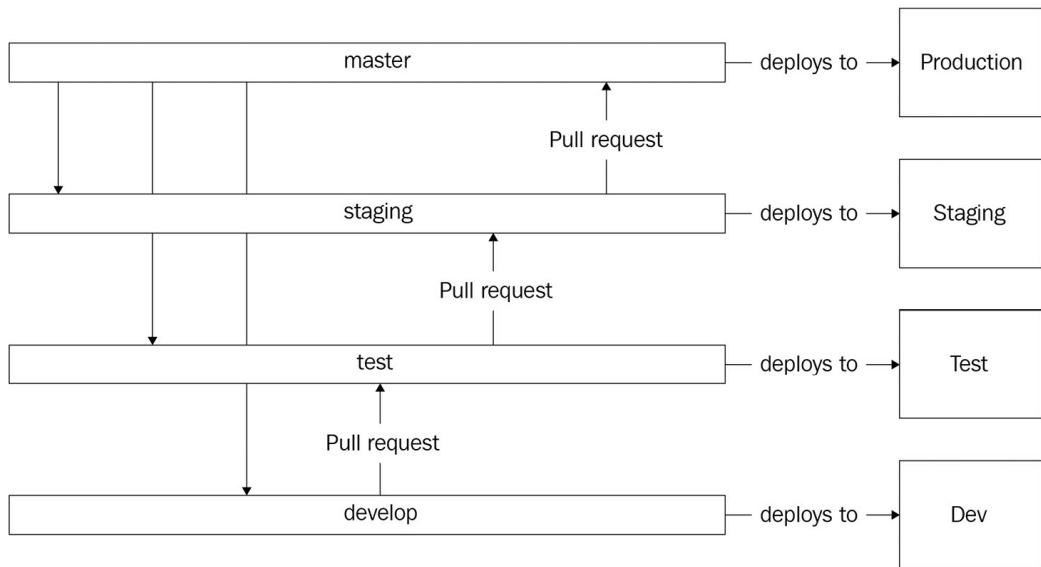


Figure 2.14 – The environment repository

The environment repository aims to act as the single source of truth for your environments. The configuration you add to the repository is applied directly to your environments.

Tip

While you can combine the environment and application repository into one, the best practice is to keep them separate. GitOps offers a clear separation between the CI and CD processes using the application and environment repositories, respectively.

Now that we've covered Git and GitOps in detail, let's look at why Git and GitOps are related but different concepts.

Git versus GitOps

The following table summarizes the differences between Git and GitOps:

	Git	GitOps
Definition	Git is a distributed version control system that tracks changes to source code and other files. It allows multiple developers to collaborate and manage code revisions efficiently.	GitOps is a set of practices and principles that leverage Git as the single source of truth for managing and automating the deployment and operation of infrastructure and applications.
Focus	Primarily focused on version control and collaboration for source code.	Focused on automating and managing the deployment and operation of infrastructure and applications through Git-based DevOps workflows.
Usage	Widely used for version control and collaboration in software development projects. Developers use Git to track changes, manage branches, and merge code.	Used for declaratively defining and managing infrastructure and application configurations. Git repositories serve as a central hub for defining desired states and driving automation.
Core Components	Repositories, branches, commits, and pull requests.	Git repositories, declarative configuration files (such as YAML), Kubernetes manifests, CI/CD pipelines, and deployment tools such as Argo CD or Flux.
Workflow	Developers clone, modify, commit, and push changes to a remote repository. They collaborate through pull requests and branch merges.	Infrastructure and application configurations are stored in Git repositories. Changes to these configurations trigger automated processes, such as CI/CD pipelines or reconciliation loops, to apply those changes to the target environment.
Benefits	Enables efficient version control, collaboration, and code management for software development teams.	Promotes infrastructure and application as code, versioning of configurations, and declarative management. It simplifies infrastructure deployment, provides consistency, and enables automated workflows.
Focus Area	Source code management.	Infrastructure and application deployment and management.
Examples	GitHub, Bitbucket, GitLab.	Argo CD, Flux, Jenkins X, Weave Flux.

Remember that while Git is a version control system, GitOps extends this concept by utilizing Git as a central source of truth for infrastructure and application configurations, allowing for automated deployment and management of DevOps workflows.

Summary

This chapter covered Git, GitOps, why we need it, its principles, and various GitOps deployments. We also looked at different kinds of repositories that we can create to implement GitOps, along with the branching strategy choices for each of them.

You should now be able to do the following:

- Understand what source code management is and how it is necessary for many activities with modern DevOps
- Create a Git repository and play around with the `clone`, `add`, `commit`, `push`, `pull`, `branch`, and `checkout` commands
- Understand what GitOps is and how it fits the modern DevOps context
- Understand why we need GitOps and how it achieves modern DevOps
- Understand the salient principles of GitOps
- Understand how to use an effective branching strategy to implement GitOps based on the org structure and product type

In the next chapter, we will develop a core understanding of containers and look at Docker.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following is true about Git? (Choose three)
 - A. It is a distributed SCM platform
 - B. It is a centralized SCM platform
 - C. It allows multiple developers to collaborate
 - D. It has commits and branches
2. In Git terms, what does Git checkout mean?
 - A. Sync code from remote to local
 - B. Switch from one branch to another
 - C. Review and approve a pull request

3. In GitOps, what forms a single source of truth
 - A. The Git repository
 - B. The configuration stored in a datastore
 - C. The secret management system
 - D. The artifact repository
4. Which of the following options are deployment models for GitOps? (Choose two)
 - A. The push model
 - B. The pull model
 - C. The staggering model
5. Should you use Gitflow for your environment repository?
6. For monolithic applications with multiple parallel developments in numerous releases, what is the most suitable Git branching strategy?
 - A. Gitflow
 - B. GitHub flow
 - C. Hybrid GitHub flow
7. Which is the recommended deployment model for GitOps?
 - A. The push model
 - B. The pull model
 - C. The staggering model

Answers

1. A,C,D
2. B
3. A
4. A,B
5. No
6. A
7. B

3

Containerization with Docker

In the previous chapter, we talked about source code management with Git, where we took a crash course on Git and then discussed GitOps and how it shapes modern DevOps practices.

In this chapter, we'll get hands-on and explore **Docker** – the de facto container runtime. By the end of this chapter, you should be able to install and configure Docker, run your first container, and then monitor it. This chapter will also form the basis for the following chapters, as we will use the same setup for the demos later.

In this chapter, we're going to cover the following main topics:

- Installing tools
- Installing Docker
- Introducing Docker storage drivers and volumes
- Running your first container
- Docker logging and logging drivers
- Docker monitoring with Prometheus
- Declarative container management with Docker Compose

Technical requirements

For this chapter, you will need a Linux machine running Ubuntu 18.04 Bionic LTS or later with sudo access. We will be using Ubuntu 22.04 Jammy Jellyfish for the entirety of this book, but feel free to use any OS of your choice. I will post links to alternative installation instructions.

You will also need to clone the following GitHub repository for some of the exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

We discussed Git extensively in the previous chapter; therefore, you can easily clone the repository using that knowledge. Now, let's move on to installing Docker on your machine.

Installing Docker

We will be installing Docker in an Ubuntu system. For other OSs, please refer to <https://docs.docker.com/engine/install/>.

To install Docker, we need to install supporting tools to allow the `apt` package manager to download Docker through HTTPS. Let's do so using the following commands:

```
$ sudo apt-get update  
$ sudo apt-get install -y ca-certificates curl gnupg
```

Download the Docker gpg key and add it to the apt package manager:

```
$ sudo install -m 0755 -d /etc/apt/keyrings  
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \  
sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg  
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Then, you need to add the Docker repository to your `apt` configuration so that you can download packages from there:

```
$ echo \  
"deb [arch="$(dpkg --print-architecture)" \  
signed-by=/etc/apt/keyrings/docker.gpg] \  
https://download.docker.com/linux/ubuntu \  
$(. /etc/os-release && echo "$VERSION_CODENAME")" \  
stable" | sudo tee /etc/apt/sources.list.d/docker.list \  
> /dev/null
```

Finally, install the Docker engine by using the following commands:

```
$ sudo apt-get update  
$ sudo apt-get -y install docker-ce docker-ce-cli \  
containerd.io docker-buildx-plugin docker-compose-plugin
```

To verify whether Docker has been installed successfully, run the following:

```
$ sudo docker --version
```

You should expect a similar output to the following:

```
Docker version 24.0.2, build cb74dfc
```

The next thing you should do is allow regular users to use Docker. You want your users to act as something other than root for building and running containers. To do that, run the following command:

```
$ sudo usermod -a -G docker <username>
```

To apply the changes to your profile, log out from your virtual machine and log back in.

Now that Docker has been fully set up on your machine, let's run a `hello-world` container to see this for ourselves:

```
$ docker run hello-world
```

You should see the following output:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
719385e32844: Pull complete
Digest: sha256:fc6cf906cbfa013e80938cdf0bb199fdbdb86d6e3e013783e5a766f50f5dbce0
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
```

You will also receive the following message, which tells you what happened behind the scenes to print the `Hello from Docker!` message on your screen:

```
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the hello-world image from Docker Hub. (amd64).
 3. The Docker daemon created a new container from that image that runs the executable
    that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it to your
    Terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

All this helpful information is self-explanatory. To explain Docker Hub a bit, it is a public Docker container registry that hosts many Docker images for people like you and me to consume.

As Docker works on a layered architecture, most Docker images are derived from one or more base images hosted on Docker Hub. So, please create a Docker Hub account for yourself to host your containers and share them with the rest of the world.

Most organizations might want to keep their images private, so you have the option of creating private repositories within Docker Hub. You can also host your own internal Docker registry using a SaaS service such as **Google Container Registry (GCR)**, or installing an artifact repository such as **Sonatype Nexus** or **JFrog Artifactory**. Whatever your choice of tool, the mechanism and how it works always remain the same.

Introducing Docker storage drivers and volumes

Docker containers are ephemeral workloads. Whatever data you store on your container filesystem gets wiped out once the container is gone. The data lives on a disk during the container's life cycle but does not persist beyond it. Pragmatically speaking, most applications in the real world are stateful. They need to store data beyond the container life cycle and want it to persist.

So, how do we go along with that? Docker provides several ways you can store data. By default, all data is stored on the writable container layer, which is ephemeral. The writable container layer interacts with the host filesystem via a storage driver. Because of the abstraction, writing files to the container layer is slower than writing directly to the host filesystem.

To solve that problem and also provide persistent storage, Docker provides volumes, bind mounts, and `tmpfs`. With them, you can interact directly with the host filesystem (and memory in the case of `tmpfs`) and save a ton of **I/O operations per second (IOPS)**, improving performance. While this section focuses on storage drivers that cater to the container filesystem, it is worth discussing multiple data storage options within Docker to provide a background.

Docker data storage options

Every option has a use case and trade-off. Let's look at each option and where you should use which.

Volumes

Docker volumes store the data directly in the host's filesystem. They do not use the storage driver layer in between, so writing to volumes is faster. They are the best way to persist data. Docker stores volumes in `/var/lib/docker/volumes` and assumes that no one apart from the Docker daemon can modify the data on them.

As a result, volumes provide the following features:

- Provide some isolation with the host filesystems. If you don't want other processes to interact with the data, then a volume should be your choice.
- You can share a volume with multiple containers.
- Volumes can either be named or anonymous. Docker stores anonymous volumes in a directory with a unique random name.
- Volumes enable you to store data remotely or in a cloud provider using volume drivers. This helps a lot if multiple containers share the same volume to provide a multi-instance active-active configuration.
- The data in the volume persists even when the containers are deleted.

Now, let's look at another storage option – bind mounts.

Bind mounts

Bind mounts are very similar to volumes but with a significant difference: they allow you to mount an existing host directory as a filesystem on the container. This lets you share important files with the Docker container, such as `/etc/resolv.conf`.

Bind mounts also allow multiple processes to modify data along with Docker. So, if you are sharing your container data with another application that is not running in Docker, bind mounts are the way to go.

tmpfs mounts

`tmpfs` mounts store data in memory. They do not store any data on disk – neither the container nor the host filesystem. You can use them to store sensitive information and the non-persistent state during the lifetime of your container.

Mounting volumes

If you mount a host directory that already contains files to an empty volume of the container, the container can see the files stored in the host. This is an excellent way to pre-populate files for your container(s) to use. However, if the directory does not exist in the host filesystem, Docker will create the directory automatically. If the volume is non-empty and the host filesystem already contains files, Docker will obscure the mount. This means that while you won't see the original files while the Docker volume is mounted to it, the files are not deleted, and you can recover them by unmounting the Docker volume.

We'll look at Docker storage drivers in the next section.

Docker storage drivers

There are numerous storage driver types. Some of the most popular ones are as follows:

- **overlay2**: This is a production-ready driver and is the preferred storage choice for Docker. It works in most environments.
- **devicemapper**: This was the preferred driver for devices running RHEL and CentOS 7 and below that did not support `overlay2`. You can use this driver if you have write-intensive activities in your containers.
- **btrfs** and **zfs**: These drivers are write-intensive and provide many features, such as allowing snapshots, and can only be used if you are using `btrfs` or `zfs` filesystems within your host.
- **vfs**: This storage driver should be used only if no copy-on-write filesystem is available. It is extremely slow, and you should refrain from using it in production.

Let's concentrate on two of the most popular ones – `overlay2` and `devicemapper`.

overlay2

`overlay2` is the default and recommended storage driver in most operating systems except RHEL 7 and CentOS 7 and older. They use file-based storage and perform best when subject to more reads than writes.

devicemapper

`devicemapper` is block-based storage and performs the best when subject to more writes than reads. Though it is compatible and the default with CentOS 7, RHEL 7, and below, as they don't support `overlay2`, it is currently not recommended in the newer versions of these operating systems that do support `overlay2`.

Tip

Use `overlay2` where possible, but if you have a specific use case for not using it (such as too many write-intensive containers), `devicemapper` is a better choice.

Configuring a storage driver

For this discussion, we will configure `overlay2` as the storage driver. Although it is configured by default, and you can skip the steps if you are following this book, it is worth a read in case you want to change it to something else.

First, let's list the existing storage driver:

```
$ docker info | grep 'Storage Driver'  
Storage Driver: overlay2
```

We can see that the existing storage driver is already `overlay2`. Let's learn how to change it to `devicemapper` if we had to.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ sudo vim /etc/docker/daemon.json
```

Add the `storage-driver` entry to the `daemon.json` configuration file:

```
{  
  "storage-driver": "devicemapper"  
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep 'Storage Driver'  
Storage Driver: devicemapper  
WARNING: The devicemapper storage-driver is deprecated, and will be removed in a future  
release.  
Refer to the documentation for more information: https://docs.docker.com/go/  
storage-driver/  
WARNING: devicemapper: usage of loopback devices is strongly discouraged for production  
use.  
Use `--storage-opt dm.thinpooldev` to specify a custom block storage device.
```

Here, we can see the `devicemapper` storage driver. We can also see several warnings with it that say that the `devicemapper` storage driver is deprecated and will be removed in a future version.

Therefore, we should stick with the defaults unless we have a particular requirement.

So, let's roll back our changes and set the storage driver to `overlay2` again:

```
$ sudo vim /etc/docker/daemon.json
```

Modify the `storage-driver` entry in the `daemon.json` configuration file to `overlay2`:

```
{  
  "storage-driver": "overlay2"  
}
```

Then, restart the Docker service and check its status:

```
$ sudo systemctl restart docker  
$ sudo systemctl status docker
```

If you rerun `docker info`, you will see the storage driver as `overlay2`, and all the warnings will disappear:

```
$ docker info | grep 'Storage Driver'  
Storage Driver: overlay2
```

Tip

Changing the storage driver will wipe out existing containers from the disk, so exercise caution when you do so and take appropriate downtimes if you're doing this in production. You will also need to pull images again since local images will fail to exist.

Now that we have installed Docker on our machine and configured the right storage driver, it's time to run our first container.

Running your first container

You can create Docker containers out of Docker container images. While we will discuss container images and their architecture in the following chapters, an excellent way to visualize them is as a copy of all files, application libraries, and dependencies comprising your application environment, similar to a virtual machine image.

To run a Docker container, we can use the `docker run` command, which has the following structure:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Let's look at the `docker run` command and its variations using working examples.

In its simplest form, you can use `docker run` by simply typing the following:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:e7c70bb24b462baa86c102610182e3efcb12a04854e8c582
838d92970a09f323
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
...
```

As you may recall, we used this command when we installed Docker. Here, I have purposefully omitted tag, options, command, and arguments. We will cover it with multiple examples to show its actual use cases.

As we didn't supply tag, Docker automatically assumed the tag as latest, so if you look at the command output, you will see that Docker is pulling the `hello-world:latest` image from Docker Hub.

Now, let's look at an example with a specific version tag.

Running containers from versioned images

We can run `nginx:1.18.0` using the following command:

```
$ docker run nginx:1.18.0
Unable to find image 'nginx:1.18.0' locally
1.18.0: Pulling from library/nginx
852e50cd189d: Pull complete
48b8657f2521: Pull complete
b4fd4d57f1a55: Pull complete
d8fbe49a7d55: Pull complete
04e4a40fabc9: Pull complete
Digest: sha256:2104430ec73de095df553d0c7c2593813e01716a48d66f
85a3dc439e050919b3
Status: Downloaded newer image for nginx:1.18.0
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
```

```
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-
listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-
envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for
start up
```

Note that the prompt will be stuck after this. There is a reason for this: `nginx` is a long-running process, also known as a daemon. Since NGINX is a web server that needs to listen to HTTP requests continuously, it should never stop. In the case of the `hello-world` application, its only job was to print the message and exit. NGINX has a different purpose altogether.

Now, no one would keep a Bash session open for a web server to run, so there has to be some way to run it in the background. You can run containers in the detached mode for that. We'll have a look at this in the next section.

Running Docker containers in the background

To run a Docker container in the background as a daemon, you can use `docker run` in detached mode using the `-d` flag:

```
$ docker run -d nginx:1.18.0
beb5dfd529c9f001539c555a18e7b76ad5d73b95dc48e8a35aec7471ea938fc
```

As you can see, it just prints a random ID and provides control back to the shell.

Troubleshooting containers

To see what's going on within the container, you can use the `docker logs` command. But before using that, we need to know the container's ID or name to see the container's logs.

To get a list of containers running within the host, run the following command

```
$ docker ps
CONTAINER ID   IMAGE      COMMAND      CREATED      STATUS      PORTS      NAMES
beb5dfd529c9   nginx:1.18.0 "/docker-      2 minutes ago  Up 2 minutes  80/tcp    fervent_
                           entrypoint..."                                shockley
```

The preceding command lists the NGINX container that we just started. Unless you specify a particular name for your container, Docker allocates a random name to it. In this case, it has called it `fervent_shockley`. It also assigns every container a unique container ID, such as `beb5dfd529c9`.

You can use the container ID or the container name to interact with the container to list the logs. Let's use the container ID this time:

```
$ docker logs beb5dfd529c9
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
...
/docker-entrypoint.sh: Configuration complete; ready for start up
```

As you can see, it prints a similar log output as it did when we ran it in the foreground.

Practically speaking, you will use `docker logs` 90% of the time unless you need to debug something with BusyBox. BusyBox is a lightweight shell container that can help you troubleshoot and debug issues with your container – mostly network issues.

Let's make BusyBox echo Hello World! for us:

```
$ docker run busybox echo 'Hello World!'
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
325d69979d33: Pull complete
Digest: sha256:560af6915bfcc8d7630e50e212e08242d37b63bd5c1ccf9bd4acccf116e262d5b
Status: Downloaded newer image for busybox:latest
Hello World!
```

As we can see, Docker pulls the latest busybox image from Docker Hub and runs the `echo 'Hello World'` command.

You can also use BusyBox in interactive mode by using the `-it` flag, which will help you run a series of commands on the BusyBox shell. It is also a good idea to add an `--rm` flag to it to tell Docker to clean up the containers once we have exited from the shell, something like this:

```
$ docker run -it --rm busybox /bin/sh
/ # echo 'Hello world!'
Hello world!
/ # wget http://example.com
Connecting to example.com (93.184.216.34:80)
saving to 'index.html'
index.html          100% |*****| 1256  0:00:00 ETA
'*' | 1256  0:00:00 ETA
'index.html' saved
/ # exit
```

Upon listing all the containers, we do not see the busybox container in there:

```
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND      CREATED     STATUS      PORTS      NAMES
beb5dfd529c9   nginx:     "/docker-      17 minutes  Up 17      80/tcp    fervent_
                  1.18.0    entrypoint..."  ago        minutes      shockley
```

There are various other flags that you can use with your containers, each serving a specific purpose. Let's look at a few common ones.

Putting it all together

The best setting for a highly available NGINX container should be something like the following:

```
$ docker run -d --name nginx --restart unless-stopped \
-p 80:80 --memory 1000M --memory-reservation 250M nginx:1.18.0
```

Let's take a look at this in more detail:

- `-d`: Run as a daemon in detached mode.
- `--name nginx`: Give the name `nginx`.
- `--restart unless-stopped`: Always automatically restart on failures unless explicitly stopped manually, and also start automatically on Docker daemon startup. Other options include `no`, `on_failure`, and `always`.
- `-p 80:80`: Forward traffic from host port 80 to container port 80. This allows you to expose your container to your host network.
- `--memory 1000M`: Limit the container memory consumption to 1000M. If the memory exceeds this limit, the container stops and acts according to the `--restart` flag.
- `--memory-reservation 250M`: Allocate a soft limit of 250M memory to the container if the server runs out of memory.

We will look into other flags in the subsequent sections as we get more hands-on.

Tip

Consider using `unless-stopped` instead of `always` as it allows you to stop the container manually if you want to do some maintenance.

Now, let's list the containers and see what we get:

```
$ docker ps -a
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
06fc749371b7 nginx "/docker- 17 seconds Up 16 0.0.0.0:   nginx
                  entrypoint..." ago    seconds 80->80/tcp
beb5dfd529c9 nginx:  "/docker- 22 minutes Up 22 80/tcp      fervent_shockley
                  1.18.0 entrypoint..." ago    minutes
```

If you look carefully, you'll see a container called `nginx` and a port forward from `0.0.0.0:80` → `80`.

Now, let's curl on `localhost:80` on the host to see what we get:

```
$ curl localhost:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

We get the NGINX welcome message. This means NGINX is running successfully, and we can access it from the machine. If you have exposed your machine's port 80 to the external world, you can also access this using your browser as follows:



Figure 3.1 – The NGINX welcome page

You also might want to restart or remove your container occasionally. We'll look at ways to do that in the next section.

Restarting and removing containers

To restart your containers, you must stop them first and then start them.

To stop your container, run the following:

```
$ docker stop nginx
```

To start your container, run the following:

```
$ docker start nginx
```

If you want to get rid of your container completely, you need to stop your container first and then remove it, using the following command:

```
$ docker stop nginx && docker rm nginx
```

Alternatively, you can use the following command to do it in one go:

```
$ docker rm -f nginx
```

Now, let's look at how we can monitor our containers with tools such as `journald` and Splunk.

Docker logging and logging drivers

Docker not only changed how applications are deployed but also the workflow for log management. Instead of writing logs to files, containers write logs to the console (`stdout/stderr`). Docker then uses a logging driver to export container logs to the specified destinations.

Container log management

Log management is an essential function within Docker, as with any application. However, due to the transient nature of Docker workloads, it becomes more critical as we lose the filesystem and potentially logs when the container is deleted or faces any issue. So, we should use log drivers to export the logs into a particular place and store and persist it. If you have a log analytics solution, the best place for your logs to be is within it. Docker supports multiple log targets via logging drivers. Let's have a look.

Logging drivers

At the time of writing, the following logging drivers are available:

- `none`: No logs are available for the container, and therefore they are not stored anywhere.
- `local`: Logs are stored locally in a custom format, which minimizes overhead.
- `json-file`: The log files are stored in JSON format. This is the default Docker logging driver.
- `syslog`: This driver uses `syslog` for storing the Docker logs as well. This option makes sense when you use `syslog` as your default logging mechanism.
- `journald`: Uses `journald` to store Docker logs. You can use the `journald` command line to browse the container and the Docker daemon logs.
- `gelf`: Sends logs to a **Graylog Extended Log Format (GELF)** endpoint such as Graylog or Logstash.
- `fluentd`: Sends logs to Fluentd.
- `awslogs`: Sends logs to AWS CloudWatch.
- `splunk`: Sends logs to Splunk using the HTTP Event Collector.
- `etwlogs`: Sends logs to **Event Tracing for Windows (ETW)** events. You can only use it on Windows platforms.
- `gcplogs`: Sends logs to Google Cloud Logging.
- `logentries`: Sends logs to Rapid7 Logentries.

While all these are viable options, we will look at `journald` and Splunk. While `journald` is a native operating system service monitoring option, Splunk is one of the most famous log analytics and monitoring tools. Now, let's understand how to configure a logging driver.

Configuring logging drivers

Let's start by finding the current logging driver:

```
$ docker info | grep "Logging Driver"  
Logging Driver: json-file
```

Currently, the default logging driver is set to `json-file`. If we want to use `journald` or Splunk as the default logging driver, we must configure the default logging driver in the `daemon.json` file.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ sudo vim /etc/docker/daemon.json
```

Add the `log-driver` entry to the `daemon.json` configuration file:

```
{  
  "log-driver": "journald"  
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep "Logging Driver"  
Logging Driver: journald
```

Now that `journald` is the default logging driver, let's launch a new NGINX container and visualize the logs:

```
$ docker run --name nginx-journald -d nginx  
66d50cc11178b0dcdb66b114ccf4aa2186b510eb1fdb1e19d563566d2e96140c
```

Now, let's look at the `journald` logs to see what we get:

```
$ sudo journalctl CONTAINER_NAME=nginx-journald  
...
```

```
Jun 01 06:11:13 99374c32101c fb8294aece02[10826]: 10-listen-on-ipv6-by-default.sh: info:  
Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf  
...  
Jun 01 06:11:13 99374c32101c fb8294aece02[10826]: 2023/06/01 06:11:13 [notice] 1#1: start  
worker process 30  
...
```

We can see the logs in the journal.

Similarly, we can configure the Splunk logging driver to send data to Splunk for analytics and visualization. Let's have a look.

Edit the `/etc/docker/daemon.json` file using an editor of your choice. If you're using `vim`, run the following command:

```
$ vim /etc/docker/daemon.json
```

Add the log-driver entry to the `daemon.json` configuration file:

```
{  
  "log-driver": "splunk",  
  "log-opt": {  
    "splunk-token": "<Splunk HTTP Event Collector token>",  
    "splunk-url": "<Splunk HTTP(S) url>"  
  }  
}
```

Then, restart the Docker service:

```
$ sudo systemctl restart docker
```

Check the status of the Docker service:

```
$ sudo systemctl status docker
```

Now, rerun `docker info` to see what we get:

```
$ docker info | grep "Logging Driver"  
Logging Driver: splunk
```

Since Splunk is now the default logging driver, let's launch a new NGINX container and visualize the logs:

```
$ docker run --name nginx-splunk -d nginx  
dedde062feba33f64efd89ef9102c7c93afa854473cda3033745d35d9065c9e5
```

Log in to your Splunk instance; you will see the Docker logs streaming. You can then analyze the logs and create visualizations out of them.

You can also have different logging drivers for different containers, and you can do so by overriding the defaults by passing the `log-driver` and `log-opt s` flags from the command line. As our current

configuration is Splunk, and we want to export data to a JSON file, we can specify `log-driver` as `json-file` while running the container. Let's have a look:

```
$ docker run --name nginx-json-file --log-driver json-file -d nginx  
379eb8d0162d98614d53aelc81ealad154745f9edbd2f64cffc2279772198bb2
```

To visualize JSON logs, we need to look into the JSON log directory – that is, `/var/lib/docker/containers/<container_id>/<container_id>-json.log`.

For the `nginx-json-file` container, we can do the following:

```
$ cat /var/lib/docker/containers\  
/379eb8d0162d98614d53aelc81ealad154745f9edbd2f64cffc2279772198bb2\  
/379eb8d0162d98614d53aelc81ealad154745f9edbd2f64cffc2279772198bb2-json.log  
{"log":"/docker-entrypoint.sh: /docker-entrypoint.d/ is not  
empty, will attempt to perform configuration\n","stream":  
"stdout","time":"2022-06-01T06:27:05.922950436Z"}  
...  
{"log":"/docker-entrypoint.sh: Configuration complete; ready  
for start up\n","stream": "stdout", "time": "2023-06-01T06:27:  
05.937629749Z"}
```

We can see that the logs are now streaming to the JSON file instead of Splunk. That is how we override the default log driver.

Tip

In most cases, it is best to stick with one default logging driver so that you have one place to analyze and visualize your logs.

Now, let's understand some of the challenges and best practices associated with Docker logging.

Typical challenges and best practices to address these challenges with Docker logging

Docker allows you to run multiple applications in a single machine or a cluster of machines. Most organizations run a mix of virtual machines and containers, and they have their logging and monitoring stack configured to support virtual machines.

Most teams struggle to make Docker logging behave the way virtual machine logging works. So, most teams will send logs to the host filesystem, and the log analytics solution then consumes the data from there. This is not ideal, and you should avoid making this mistake. It might work if your container is static, but it becomes an issue if you have a cluster of servers, each running Docker, and you can schedule your container in any virtual machine you like.

So, treating a container as an application running on a virtual machine is a mistake from a logging point of view. Instead, you should visualize the container as an entity – just like a virtual machine. It would be best if you never associated containers with a virtual machine.

One solution is to use the logging driver to forward the logs to a log analytics solution directly. But then, the logging becomes heavily dependent on the availability of the log analytics solution. So, it might not be the best thing to do. People faced issues when their services running on Docker went down because the log analytics solution was unavailable or there were network issues.

Well, the best way to approach this problem is to use JSON files to store the logs temporarily in your virtual machine and use another container to push the logs to your chosen log analytics solution the old-fashioned way. That way, you decouple from the dependency on an external service to run your application.

You can use the logging driver to export logs directly to your log analytics solution within the log forwarder container. There are many logging drivers available that support many log targets. Always mark the logs in such a way that the containers appear as their own entities. This will disassociate containers from virtual machines, and you can then make the best use of a distributed container-based architecture.

So far, we've looked at the logging aspects of containers, but one of the essential elements of a DevOps engineer's role is monitoring. We'll have a look at this in the next section.

Docker monitoring with Prometheus

Monitoring Docker nodes and containers is an essential part of managing Docker. There are various tools available for monitoring Docker. While you can use traditional tools such as Nagios, Prometheus is gaining ground in cloud-native monitoring because of its simplicity and pluggable architecture.

Prometheus is a free, open source monitoring tool that provides a dimensional data model, efficient and straightforward querying using the **Prometheus query language (PromQL)**, efficient time series databases, and modern alerting capabilities.

It has several exporters available for exporting data from various sources and supports both virtual machines and containers. Before we delve into the details, let's look at some of the challenges with container monitoring.

Challenges with container monitoring

From a conceptual point of view, there is no difference between container monitoring and the traditional method. You still need metrics, logs, health checks, and service discovery. These aren't things that are unknown or haven't been explored before. The problem with containers is the abstraction that they bring with them; let's look at some of the issues:

- Containers behave like mini virtual machines; however, in reality, they are processes running on a server. However, they still have everything to monitor that we would in a virtual machine.

A container process will have many metrics, very similar to virtual machines, to be treated as separate entities altogether. When dealing with containers, most people make this mistake when they map containers to a particular virtual machine.

- Containers are temporary, and most people don't realize that. When you have a container and it is recreated, it has a new IP. This can confuse traditional monitoring systems.
- Containers running on clusters can move from one node (server) to another. This adds another layer of complexity as your monitoring tool needs to know where your containers are to scrape metrics from them. This should not matter with the more modern, container-optimized tools.

Prometheus helps us address these challenges as it is built from a distributed application's point of view. To understand this, we'll look at a hands-on example. However, before that, let's install Prometheus on a separate Ubuntu 22.04 Linux machine.

Installing Prometheus

Installing Prometheus consists of several steps, and for simplicity, I've created a Bash script for installing and setting up Prometheus on an Ubuntu machine.

Use the following commands on a separate machine where you want to set up Prometheus:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch3/prometheus/
$ sudo bash prometheus_setup.sh
```

To check whether Prometheus is installed and running, check the status of the Prometheus service using the following command:

```
$ sudo systemctl status prometheus
prometheus.service - Prometheus
   Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset:
enabled)
     Active: active (running) since Tue 2023-06-01 09:26:57 UTC; 1min 22s ago
```

As the service is `Active`, we can conclude that Prometheus has been installed and is running successfully. The next step is configuring the Docker server to enable Prometheus to collect logs from it.

Configuring cAdvisor and the node exporter to expose metrics

Now, we'll launch a cAdvisor container on the machine running Docker to expose the metrics of the Docker containers. cAdvisor is a metrics collector that scrapes metrics from containers. To launch the container, use the following command:

```
$ docker run -d --restart always --name cadvisor -p 8080:8080 \
-v "/:/rootfs:ro" -v "/var/run:/var/run:rw" -v "/sys:/sys:ro" \
-v "/var/lib/docker:/var/lib/docker:ro" google/cadvisor:latest
```

Now that cAdvisor is running, we need to configure the node exporter to export node metrics on the Docker machine. To do so, run the following commands:

```
$ cd ~/modern-devops/ch3/prometheus/  
$ sudo bash node_exporter_setup.sh
```

Now that the node exporter is running, let's configure Prometheus to connect to cAdvisor and the node exporter and scrape metrics from there.

Configuring Prometheus to scrape metrics

We will now configure Prometheus on the Prometheus machine so that it can scrape the metrics from cAdvisor. To do so, modify the `/etc/prometheus/prometheus.yml` file so that it includes the following within the server running Prometheus:

```
$ sudo vim /etc/prometheus/prometheus.yml  
...  
- job_name: 'node_exporter'  
  scrape_interval: 5s  
  static_configs:  
    - targets: ['localhost:9100', '<Docker_IP>:9100']  
- job_name: 'Docker Containers'  
  static_configs:  
    - targets: ['<Docker_IP>:8080']
```

After changing this configuration, we need to restart the Prometheus service. Use the following command to do so:

```
$ sudo systemctl restart prometheus
```

Now, let's launch a sample web application that we will monitor using Prometheus.

Launching a sample container application

Now, let's run an NGINX container called `web` that runs on port 8081 on the Docker machine. To do so, use the following command:

```
$ docker run -d --name web -p 8081:80 nginx  
f9b613d6bdf3d6aee0cb3a08cb55c99a7c4821341b058d8757579b52cabbb0f5
```

Now that we've set up the Docker container, let's go ahead and open the Prometheus UI by visiting `https://<PROMETHEUS_SERVER_EXTERNAL_IP>:9090` and then running the following query by typing it in the textbox:

```
container_memory_usage_bytes{name=~"web"}
```

It should show something like the following:

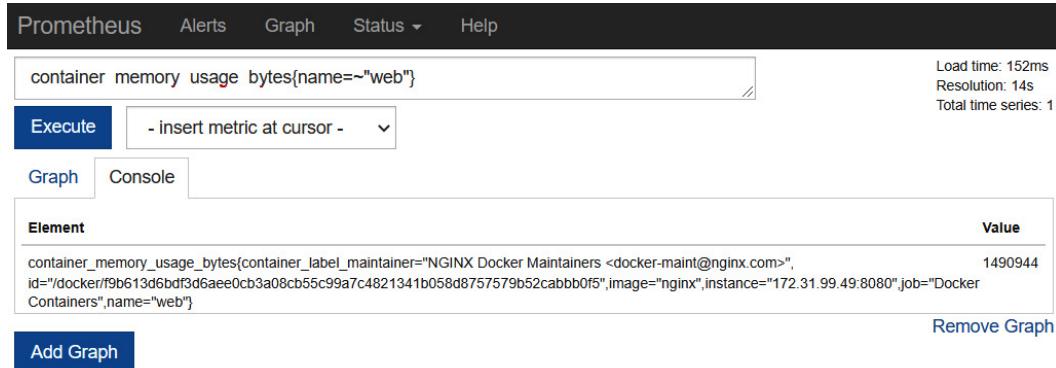


Figure 3.2 – Prometheus – container_memory_usage_bytes

We can also view the time series of this metric by clicking on the **Graph** tab. However, before doing so, let's load our NGINX service using the Apache Bench tool. Apache Bench is a load-testing tool that helps us fire HTTP requests to the NGINX endpoint using the command line.

On your Docker server, run the following command to start a load test:

```
$ ab -n 100000 http://localhost:8081
```

It will hit the endpoint with 100,000 requests, which means it provides a fair amount of load to do a memory spike. Now, if you open the **Graph** tab, you should see something like the following:

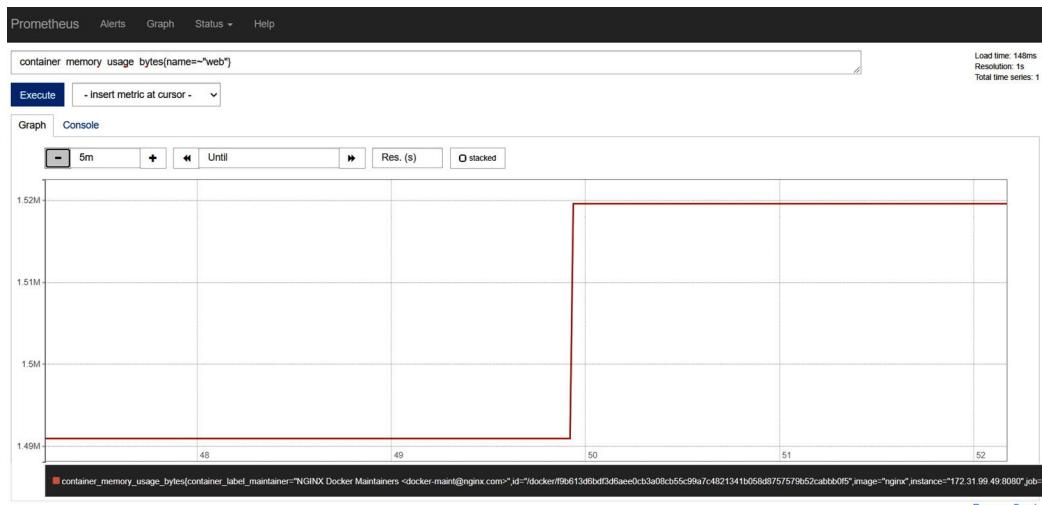


Figure 3.3 – Prometheus – container_memory_usage_bytes – Graph

To visualize node metrics, we can use the following PromQL statement to get the `node_cpu` value of the Docker host:

```
node_cpu{instance=<Docker_IP>:9100,job="node_exporter"}
```

As shown in the following screenshot, it will provide us with the `node_cpu` metrics for multiple modes:

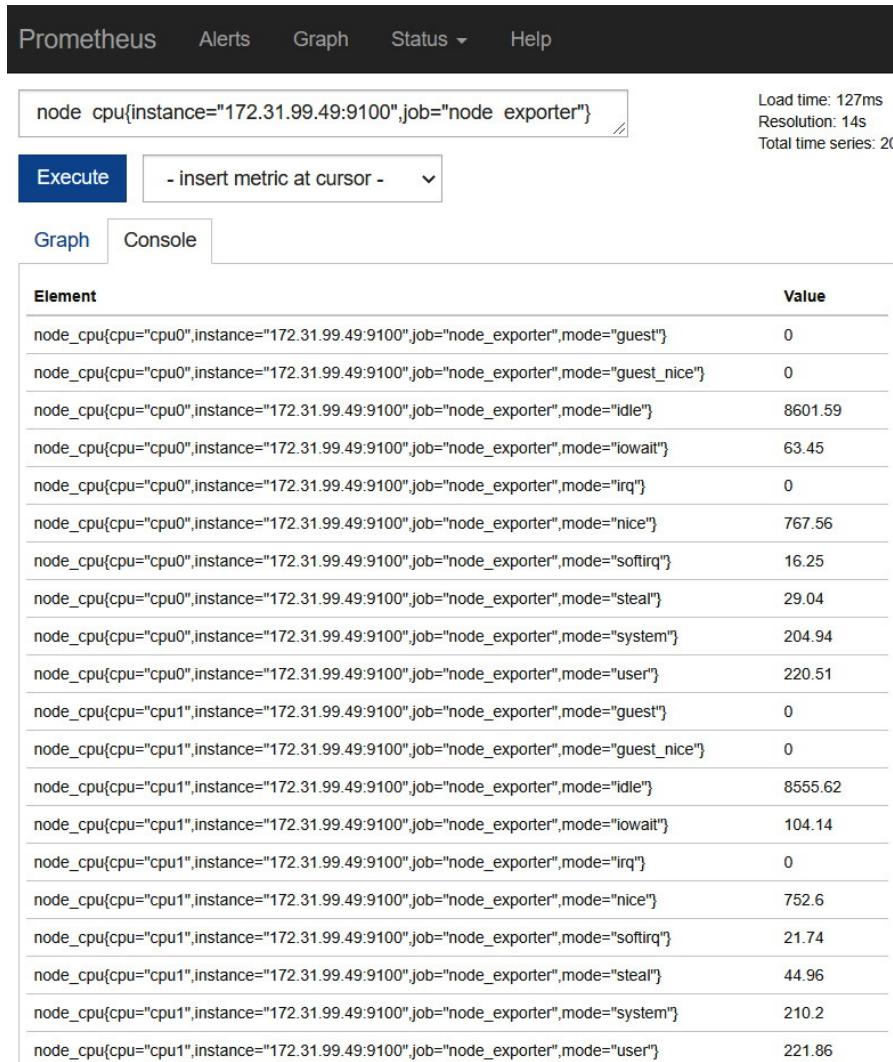


Figure 3.4 – Prometheus – node_cpu

There are a variety of other metrics that Prometheus gives you to visualize. Let's understand some of the metrics you can monitor.

Metrics to monitor

Monitoring metrics is a complex subject, and it would depend mostly on your use case. However, the following are some guidelines on what metrics you want to monitor.

Host metrics

You need to monitor your host metrics as your containers run on them. Some of the metrics that you can watch are as follows:

- **Host CPU:** It's good to know whether your host has sufficient CPU to run your containers. If not, it might terminate some of your containers to account for that. So, to ensure reliability, you need to keep this in check.
- **Host memory:** Like the host CPU, you need to watch the host memory to detect issues such as memory leaks and runaway memory.
- **Host disk space:** As Docker containers use the host filesystem to store transient and persistent files, you need to monitor it.

Docker container metrics

Docker container metrics are the next thing to consider:

- **Container CPU:** This metric will provide the amount of CPU used by the Docker container. You should monitor it to understand the usability pattern and decide where to place your container effectively.
- **Throttled CPU time:** This metric allows us to understand the total time when the CPU was throttled for a container. This lets us know whether a particular container needs more CPU time than others, and you can adjust the CPU share constraint accordingly.
- **Container memory fail counters:** This metric provides the number of times the container requested more than the allocated memory. It will help you understand what containers require more than the allocated memory, and you can plan to run those containers accordingly.
- **Container memory usage:** This metric will provide the amount of memory used by the Docker container. You can set memory limits according to the usage.
- **Container swap:** This metric will tell you what containers were using swap instead of RAM. It helps us identify memory-hungry containers.
- **Container disk I/O:** This is an important metric and will help us understand containers' disk profiles. Spikes can indicate a disk bottleneck or suggest that you might want to revisit your storage driver configuration.

- **Container network metrics:** This metric will tell us how much network bandwidth the containers use and help us understand traffic patterns. You can use these to detect an unexpected network spike or a denial-of-service attack.

Important tip

Profiling your application during the performance testing phase in the non-production environment will give you a rough idea of how the system will behave in production. The actual fine-tuning of your application begins when you deploy them to production. Therefore, monitoring is critical, and fine-tuning is a continuous process.

So far, we have been running commands to do most of our work. That is the imperative way of doing this. But what if I told you that instead of typing commands, you could declare what you want, and something could run all the required commands on your behalf? That is known as the declarative method of managing an application. Docker Compose is one of the popular tools to achieve this. We'll have a look at this in the next section.

Declarative container management with Docker Compose

Docker Compose helps you manage multiple containers in a declarative way. You can create a YAML file and specify what you want to build, what containers you want to run, and how the containers interact with each other. You can define mounts, networks, port mapping, and many different configurations in the YAML file.

After that, you can simply run `docker compose` up to run your entire containerized application.

Declarative management is quickly gaining ground because of its power and simplicity. Now, sysadmins don't need to remember what commands they had run or write lengthy scripts or playbooks to manage containers. Instead, they can simply declare what they want in a YAML file, and `docker compose` or other tools can help them achieve that state. We installed Docker Compose when we installed Docker, so let's see it in action with a sample application.

Deploying a sample application with Docker Compose

We have a Python Flask application that listens on port 5000, which we will eventually map to host port 80. The application will connect to the Redis database as a backend service on its default port, 6379, and fetch the page's last visit time. We will not expose that port to the host system. This means the database is entirely out of bounds for any external party with access to the application.

The following diagram depicts the application architecture:

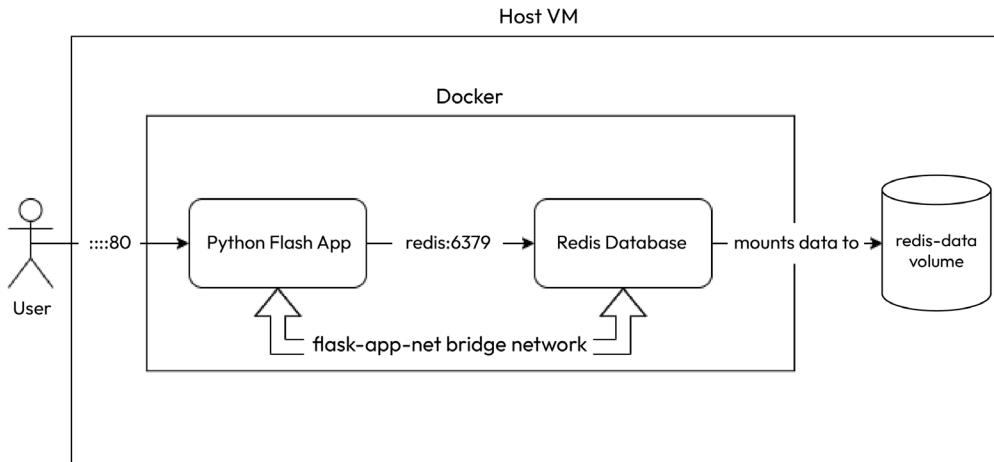


Figure 3.5 – Sample application

The necessary files are available in this book's GitHub repository. Run the following command to locate the files:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch3/docker-compose
$ ls -l
total 16
-rw-r--r-- 1 root root 681 Nov 25 06:11 app.py
-rw-r--r-- 1 root root 389 Nov 25 06:45 docker-compose.yaml
-rw-r--r-- 1 root root 238 Nov 25 05:27 Dockerfile
-rw-r--r-- 1 root root 12 Nov 25 05:26 requirements.txt
```

The `app.py` file looks as follows:

```
import time
import redis
from flask import Flask
from datetime import datetime
app = Flask(__name__)
cache = redis.Redis(host='redis', port=6379)
def get_last_visited():
    try:
        last_visited = cache.getset('last_visited', str(datetime.now().strftime("%Y-%m-%d,
%H:%M:%S")))
        if last_visited is None:
            return cache.getset('last_visited', str(datetime.now().strftime("%Y-%m-%d,
%H:%M:%S")))
        return last_visited
    except redis.exceptions.ConnectionError as e:
```

```
        raise e
@app.route('/')
def index():
    last_visited = str(get_last_visited().decode('utf-8'))
    return 'Hi there! This page was last visited on {}.\n'.format(last_visited)
```

The requirements.txt file looks as follows:

```
flask
redis
```

I've already built the application for you, and the image is available on Docker Hub. The next chapter will cover how to build a Docker image in detail. For now, let's have a look at the docker-compose file.

Creating the docker-compose file

The next step in the process is to create a docker-compose file. A docker-compose file is a YAML file that contains a list of services, networks, volumes, and other associated configurations. Let's look at the following example docker-compose.yaml file to understand it better:

```
version: "2.4"
services:
  flask:
    image: "bharanmicrosystems/python-flask-redis:latest"
    ports:
      - "80:5000"
    networks:
      - flask-app-net
  redis:
    image: "redis:alpine"
    networks:
      - flask-app-net
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis-data:/data
  networks:
    flask-app-net:
      driver: bridge
volumes:
  redis-data:
```

The YAML file describes two services – flask and redis.

The flask service uses the python-flask-redis:latest image – the image we built with the preceding code. It also maps host port 80 to container port 5000, exposing this application to your host machine on port 80, and you can access it via <http://localhost>.

The redis service uses the official redis:alpine image and does not expose any port, as we don't want this service outside the container network's confines. However, it declares a persistent volume,

redis-data, that comprises the /data directory. We can mount this volume on the host filesystem for persistence beyond the container life cycle.

There is also a flask-app-net network that uses the bridge driver, and both services share the same network. This means the services can call each other by using their service names. If you look at the app.py code, you will see that we established a Redis service connection using the redis hostname.

To apply the configuration, simply run docker-compose up -d:

```
$ docker compose up -d
[+] Running 17/17
  flask 9 layers []      0B/0B      Pulled 10.3s
  redis 6 layers []      0B/0B      Pulled 9.1s
  [+] Building 0.0s (0/0)
  [+] Running 4/4
Network docker-compose_flask-app-net  Created 0.1s
Volume "docker-compose_redis-data"    Created 0.0s
Container docker-compose-flask-1      Started 3.8s
Container docker-compose-redis-1      Stated
```

Now, let's list the Docker containers to see how we fare:

```
$ docker ps
CONTAINER ID IMAGE          COMMAND       CREATED      STATUS      PORTS          NAMES
9151e72f5d66  redis:        "docker-      3 minutes   Up 3       6379/tcp      docker-compose
               alphone        entrypoint.s..."   ago         minutes
9332c2aaf2c4  bharamicrosystems "flask run"  3 minutes   Up 3       0.0.0.0:80->  docker-compose
               /python-flask-
               redis:latest           ago         minutes   5000/tcp,
                                         ::80->5000/tcp      -flask-1
```

We can see that two containers are running for both services. We can also see host port 80 forwarding connections to container port 5000 on the flask service.

The redis service is internal and therefore, there is no port mapping.

Let's run curl localhost and see what we get:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:27.
```

Here, we get the last visited page from the Redis cache according to the sample Flask application code.

Let's run this a few times and see whether the time changes:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:28.
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:51.
```

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:54:52.
```

We can see that the last visited time changes every time we `curl`. Since the volume is persistent, we should get similar last visited times even after a container restarts.

First, let's `curl` and get the last visited time and also the current date:

```
$ curl localhost && date
Hi there! This page was last visited on 2023-06-01, 06:54:53.
Thu Jun  1 06:55:50 UTC 2023
```

Now, the next time we `curl`, we should get a date-time similar to `2023-06-01, 06:55:50`. But before that, let's restart the container and see whether the data persists:

```
$ docker compose restart redis
[+] Restarting 1/1
Container docker-compose-redis-1 Started
```

Now that Redis has been restarted, let's run `curl` again:

```
$ curl localhost
Hi there! This page was last visited on 2023-06-01, 06:55:50.
```

As we can see, we get the correct last visited time, even after restarting the `redis` service. This means that data persistence works correctly, and the volume is adequately mounted.

You can do many other configurations on `docker compose` that you can readily get from the official documentation. However, you should now have a general idea about using `docker compose` and its benefits. Now, let's look at some of the best practices associated with Docker Compose.

Docker Compose best practices

Docker Compose provides a declarative way of managing Docker container configuration. This enables GitOps for your Docker workloads. While Docker Compose is primarily used in development environments, you can use it in production very effectively, especially when Docker runs in production and does not use another container orchestrator such as Kubernetes.

Always use `docker-compose.yml` files alongside code

The YAML file defines how to run your containers. So, it becomes a valuable tool for declaratively building and deploying your containers from a single space. You can add all dependencies to your application and run related applications in a single network.

Separate multiple environment YAMLs using overrides

Docker Compose YAML files allow us to both build and deploy Docker images. Docker has enabled the *build once, run anywhere* concept. This means we build once in the development environment and then use the created image in subsequent environments. So, the question arises of how we can achieve that. Docker Compose allows us to apply multiple YAML files in a sequence where the next configuration overrides the last. That way, we can have separate override files for various environments and manage multiple environments using a set of these files.

For example, say we have the following base docker-compose.yaml file:

```
version: "2.4"
services:
  flask:
    image: "bharamicrosystems/python-flask-redis:latest"
    ports:
      - "80:5000"
    networks:
      - flask-app-net
  redis:
    image: "redis:alpine"
    networks:
      - flask-app-net
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - redis-data:/data
  networks:
    flask-app-net:
      driver: bridge
  volumes:
    redis-data:
```

We only have to build the Flask application container image in the development environment so that we can create an override file for the development environment – that is, docker-compose.override.yaml:

```
web:
  build: .
  environment:
    DEBUG: 'true'
  redis:
    ports:
      - 6379:6379
```

Here, we added a build parameter within the web service. This means the Python Flask application will be rebuilt and then deployed. We also set the DEBUG environment variable within the web service and exposed the redis port to the host filesystem. This makes sense in the development environment, as we might want to debug Redis from the development machine directly. Still, we would not want something of that sort in the production environment. Therefore, the default docker-compose.yaml file will work in the production environment, as we saw in the previous section.

Use an .env file to store sensitive variables

You might not want to store sensitive content such as passwords and secrets in version control. Instead, you can use an `.env` file that contains a list of variable names and values and keep it in a secret management system such as HashiCorp Vault.

Be mindful of dependencies in production

When you change a particular container and want to redeploy it, `docker-compose` also redeploys any dependencies. Now, this might not be something that you wish to do, so you can override this behavior by using the following command:

```
$ docker-compose up --no-deps -d <container_service_name>
```

Treat docker-compose files as code

Always version control your `docker-compose` files and keep them alongside the code. This will allow you to track their versions and use gating and Git features such as pull requests.

Summary

This chapter was designed to cater to both beginners and experienced individuals. We started by covering the foundational concepts of Docker and gradually delved into more advanced topics and real-world use cases. This chapter began with installing Docker, running our first Docker container, understanding various modes of running a container, and understanding Docker volumes and storage drivers. We also learned how to select the right storage driver, volume options, and some best practices. All these skills will help you easily set up a production-ready Docker server. We also discussed the logging agent and how to quickly ship Docker logs to multiple destinations, such as **journald**, **Splunk**, and JSON files, to help you monitor your containers. We looked at managing Docker containers declaratively using **Docker Compose** and deployed a complete composite container application. Please try out all the commands mentioned in this chapter for a more hands-on experience – practice is vital to achieving something worthwhile and learning something new.

As a next step, in the following chapter, we will look at Docker images, creating and managing them, and some best practices.

Questions

Answer the following questions to test your knowledge of this chapter:

1. You should use `overlay2` for CentOS and RHEL 7 and below. (True/False)
2. Which of the following statements is true? (Choose four)
 - A. Volumes increase IOPS.
 - B. Volumes decrease IOPS.

- C. `tmpfs` mounts use system memory.
 - D. You can use bind mounts to mount host files to containers.
 - E. You can use volume mounts for a multi-instance active-active configuration.
3. Changing the storage driver removes existing containers from the host. (True/False)
 4. `devicemapper` is a better option than `overlay2` for write-intensive containers. (True/False)
 5. Which of the following logging drivers are supported by Docker? (Choose four)
 - A. `journald`
 - B. Splunk
 - C. JSON files
 - D. Syslog
 - E. Logstash
6. Docker Compose is an imperative approach to managing containers. (True/False)
 7. Which of the following `docker run` configurations are correct? (Choose three)
 - A. `docker run nginx`
 - B. `docker run --name nginx nginx:1.17.3`
 - C. `docker run -d --name nginx nginx`
 - D. `docker run -d --name nginx nginx --restart never`

Answers

The following are the answers to this chapter's questions:

1. False – You should use `devicemapper` for CentOS and RHEL 7 and below as they do not support `overlay2`.
2. B, C, D, E.
3. True.
4. True.
5. A, B, C, D.
6. False – Docker Compose is a declarative approach to container management.
7. A, B, C.

4

Creating and Managing Container Images

In the previous chapter, we covered containerization with Docker, where we installed Docker and ran our first container. We covered some core fundamentals, including Docker volumes, mounts, storage drivers, and logging drivers. We also covered Docker Compose as a declarative method of managing containers.

Now, we will discuss the core building blocks of containers: container images. Container images also fulfill a core principle of modern DevOps practices: config as code. Therefore, understanding container images, how they work, and how to build an image effectively is very important for a modern DevOps engineer.

In this chapter, we're going to cover the following main topics:

- Docker architecture
- Understanding Docker images
- Understanding Dockerfiles, components, and directives
- Building and managing Docker images
- Flattening Docker images
- Optimizing containers with distroless images
- Understanding Docker registries

Technical requirements

For this chapter, we assume that you have Docker installed on a Linux machine running Ubuntu 18.04 Bionic LTS or later with sudo access. You can read *Chapter 3, Containerization with Docker*, for more details on how to do that.

You will also need to clone a GitHub repository for some of the exercises in this chapter, which you can find at <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>. Also, you need a Docker Hub account for most of the activities. To create one, go to <https://hub.docker.com/>.

Docker architecture

Imagine you're a passionate chef dedicated to creating mouthwatering dishes that satisfy hungry customers. In your kitchen, which is a magical place called Docker, you have special powers to plan, make, and showcase your culinary creations. Let's break down the key parts:

Ingredients (Application Code and Dependencies): Imagine your kitchen has shelves filled with ingredients such as flour, eggs, and spices. These ingredients come together in a specific way to make a dish. Similarly, your application code and dependencies work together to build your application.

Recipe (Image): Each recipe is like a plan for a particular dish. Imagine having a recipe for chocolate cake or pasta carbonara. These recipes are like the building blocks for your creations. In the same way, a Docker image is a plan for making your Docker container.

Recipe Cards (Dockerfile): Your cooking journey involves using special recipe cards called Dockerfiles. These cards show you the important steps and ingredients (commands) to follow. For example, a Dockerfile for a chocolate cake might have steps such as “Mix the flour and sugar” or “Add eggs and cocoa powder.” These Dockerfiles guide your helpers (Docker) in making the dish (container).

Cooked Dish (Container): When someone wants a dish, you use the recipe (image) to make it. Then, you have a fresh, hot dish ready to serve. These dishes are separate, but they can be made again and again (thanks to the recipe), just like a container.

Kitchen Staff (Docker Engine): In your bustling kitchen, your helpers (Docker Engine) play a big role. They do the hard work, from getting ingredients to following the recipe and serving the dish. You give them instructions (Docker commands), and they make it happen. They even clean up after making each dish.

Special Set Menu (Docker Compose): Sometimes, you want to serve a special meal with multiple dishes that go well together. Think of a meal with an appetizer, a main course, and a dessert. Using Docker Compose is like creating a special menu for that occasion. It lists recipes (images) for each part of the meal and how they should be served. You can even customize it to create a whole meal experience with just one command.

Storage Area (Volumes): In your kitchen, you need a place to keep ingredients and dishes. Think of Docker volumes as special storage areas where you can keep important things, such as data and files, that multiple dishes (containers) can use.

Communication Channels (Networks): Your kitchen is a busy place with lots of talking and interacting. In Docker, networks are like special communication paths that help different parts of your kitchen (containers) talk to each other.

So, Docker is like your magical kitchen where you make dishes (containers) using plans (Dockerfiles) and ingredients (images) with the assistance of your kitchen helpers (Docker Engine). You can even serve entire meals (Docker Compose) and use special storage areas (volumes) and communication paths (networks) to make your dishes even more amazing. Just like a chef gets better with practice, exploring Docker will help you become a master of DevOps in no time! Now, let's dive deeper into Docker architecture to understand its nuances!

As we already know, Docker uses the *build once, run anywhere* concept. Docker packages applications into images. Docker images form the blueprint of containers, so a container is an instance of an image.

A container image packages applications and their dependencies, so they are a single immutable unit you can run on any machine that runs Docker. You can also visualize them as a snapshot of the container.

We can build and store Docker images in a Docker registry, such as **Docker Hub**, and then download and use those images in the system where we want to deploy them. Images comprise several layers, which helps break images into multiple parts. The layers tend to be reusable stages that other images can build upon. This also means we don't have to transmit the entire image over a network when changing images. We only transmit the delta, which saves a lot of network I/O. We will talk about the layered filesystem in detail later in this chapter.

The following diagram shows the components Docker uses to orchestrate the following activities:

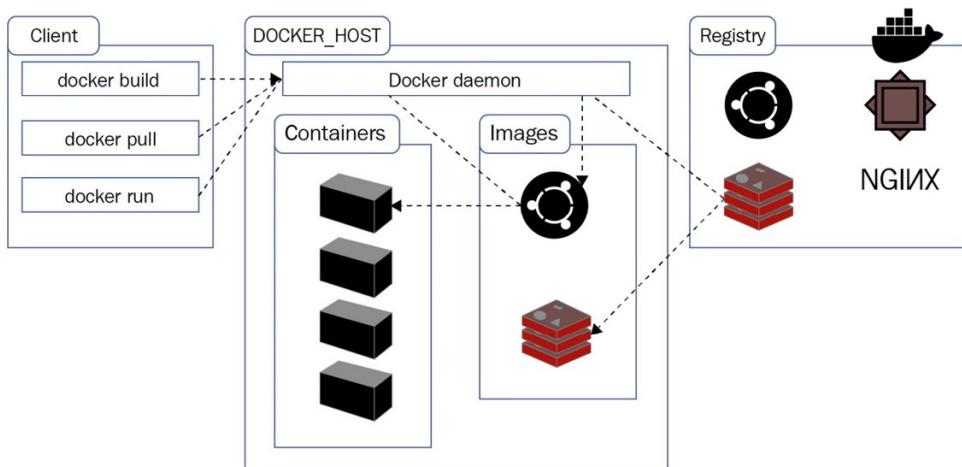


Figure 4.1 – Docker architecture

The components are:

- **Docker daemon:** This process runs on the servers that we want to run our containers on. They deploy and run containers on the Docker server.
- **Docker registries:** These store and distribute Docker images.
- **Docker client:** This is the command-line utility that we've been using to issue `docker` commands to the Docker daemon.

Now that we understand Docker architecture's key components and how Docker images play an essential role, let's understand Docker images and their components, directives, and registries in detail.

Understanding Docker images

Docker images form the blueprint of Docker containers. Just like you need a blueprint for a shipping container to determine its size and what goods it will contain, a Docker image specifies what packages, source code, dependencies, and libraries it needs to use. It also determines what it needs to do for the source code to run effectively.

Technically, it consists of a series of steps you would perform on a base OS image to get your application up and running. This may include installing packages and dependencies, copying the source code to the correct folder, building your code to generate a binary, and so on.

You can store Docker images in a container registry, a centralized location from where your Docker machines can pull images to create containers.

Docker images use a layered filesystem. Instead of a huge monolithic block on the filesystem that comprises the template to run containers, we have many layers, one on top of the other. But what does this mean? What problem does this solve? Let's have a look in the next section.

The layered filesystem

Layers in Docker are intermediate Docker images. The idea is that every Dockerfile statement we execute on top of a layer changes something within the layer and builds a new one. The subsequent statement modifies the current one to generate the next one. The final layer executes the Docker CMD or ENTRYPOINT command, and the resulting image comprises several layers arranged one on top of the other. Let's understand this by looking at a simple example.

If we pull the *Flask application* we built in the previous chapter, we will see the following:

```
$ docker pull bharamicrosystems/python-flask-redis
Using default tag: latest
latest: Pulling from bharamicrosystems/python-flask-redis
188c0c94c7c5: Pull complete
a2f4f20ac898: Pull complete
f8a5b284ee96: Pull complete
```

```
28e9c106bfa8: Pull complete
8fe1e74827bf: Pull complete
95618753462e: Pull complete
03392fcaa2ba: Pull complete
4de3b61e85ea: Pull complete
266ad40b3bdb: Pull complete
Digest: sha256:bb40a44422b8a7fea483a775fe985d4e05f7e5c59b0806a2
4f6cca50edad824
Status: Downloaded newer image for bharamicrosystems/python-flask-redis:latest
docker.io/bharamicrosystems/python-flask-redis:latest
```

As you can see, many `Pull complete` statements are beside random IDs. These are called **layers**. The current layer contains just the differences between the previous and current filesystem. A container image comprises several layers.

Containers contain an additional writable filesystem on top of the image layers. This is the layer where your containers modify the filesystem to provide the expected functionality.

There are several advantages of using layers instead of merely copying the entire filesystem of the container. Since image layers are read-only, multiple containers created from an image share the same layered filesystem, decreasing the overall disk and network footprint. Layers also allow you to share filesystems between images. For example, if two images come from a single base image, both images share the same base layer.

The following diagram shows a Python application that runs on an Ubuntu OS. At a high level, you will see a base layer (Ubuntu OS) and Python installed on top of it. On top of Python, we've installed the Python app. All these components form the image. When we create a container out of the image and run it, we get the writable filesystem on top as the final layer:

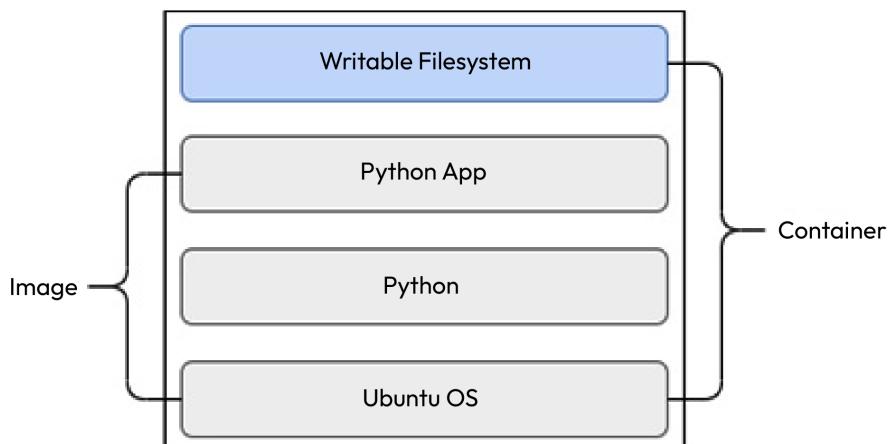


Figure 4.2 – Container layers

So, you can create multiple Python app images from the same base image and customize them according to your needs.

The writable filesystem is unique for every container you spin from container images, even if you create containers from the same image.

Image history

To understand images and their layers, you can always inspect the image history.

Let's inspect the history of the last Docker image by running the following command:

```
$ docker history bharamicrosystems/python-flask-redis
IMAGE      CREATED     CREATED BY
6d33489ce4d9  2 years ago  /bin/sh -c #(nop)  CMD ["flask" "run"]          0B
<missing>   2 years ago  /bin/sh -c #(nop) COPY dir:61bb30c35fb351598...  1.2kB
<missing>   2 years ago  /bin/sh -c #(nop)  EXPOSE 5000                0B
<missing>   2 years ago  /bin/sh -c pip install -r requirements.txt    11.2MB
<missing>   2 years ago  /bin/sh -c #(nop) COPY file:4346cf08412270cb...  12B
<missing>   2 years ago  /bin/sh -c apk add --no-cache gcc musl-dev l...  143MB
<missing>   2 years ago  /bin/sh -c #(nop) ENV FLASK_RUN_HOST=0.0.0.0  0B
<missing>   2 years ago  /bin/sh -c #(nop) ENV FLASK_APP=app.py       0B
<missing>   2 years ago  /bin/sh -c #(nop) CMD ["python3"]            0B
<missing>   2 years ago  /bin/sh -c set -ex;  wget -O get-pip.py "$P...  7.24MB
<missing>   2 years ago  /bin/sh -c #(nop) ENV PYTHON_GET_PIP_SHA256...  0B
<missing>   2 years ago  /bin/sh -c #(nop) ENV PYTHON_GET_PIP_URL=ht...  0B
<missing>   2 years ago  /bin/sh -c #(nop) ENV PYTHON_PIP_VERSION=20...  0B
<missing>   2 years ago  /bin/sh -c cd /usr/local/bin && ln -s idle3...  32B
<missing>   2 years ago  /bin/sh -c set -ex && apk add --no-cache ---  28.3MB
<missing>   2 years ago  /bin/sh -c #(nop) ENV PYTHON_VERSION=3.7.9  0B
<missing>   2 years ago  /bin/sh -c #(nop) ENV GPG_KEY=0D96DF4D4110E...  0B
<missing>   2 years ago  /bin/sh -c set -eux;  apk add --no-cache c...  512kB
<missing>   2 years ago  /bin/sh -c #(nop) ENV LANG=C.UTF-8        0B
<missing>   2 years ago  /bin/sh -c #(nop) ENV PATH=/usr/local/bin:/...  0B
<missing>   2 years ago  /bin/sh -c #(nop) CMD ["/bin/sh"]           0B
<missing>   2 years ago  /bin/sh -c #(nop) ADD file:f17f65714f703db90...  5.57MB
```

As you can see, there are several layers, and every layer has associated commands. You can also see when the layers were created and the size of the disk space occupied by each. Some layers do not occupy any disk space, as they haven't added anything new to the filesystem, such as `CMD` and `EXPOSE` directives. These perform some functions, but they do not write anything to the filesystem. While commands such as `apk add` write to the filesystem, you can see them taking up disk space.

Every layer modifies the old layer in some way, so every layer is just a delta of the filesystem configuration.

In the next section, we will deep dive into Dockerfiles and find out how we can build Docker images and see what the layered architecture looks like.

Understanding Dockerfiles, components, and directives

A Dockerfile is a simple file that constitutes a series of steps to build a Docker image. Each step is known as a **directive**. There are different kinds of directives. Let's look at a simple example to understand how this works.

We will create a simple NGINX container by building the image from scratch rather than using the one available on Docker Hub. NGINX is very popular web server software that you can use for a variety of applications; for example, it can serve as a load balancer or a reverse proxy.

Start by creating a Dockerfile:

```
$ vim Dockerfile
FROM ubuntu:bionic
RUN apt update && apt install -y curl
RUN apt update && apt install -y nginx
CMD ["nginx", "-g", "daemon off;"]
```

Let's look at each line and directive one by one to understand how this Dockerfile works:

- The `FROM` directive specifies what the base image for this container should be. This means we are using another image as the base and will be building layers on top of it. We use the `ubuntu:bionic` package as the base image for this build since we want to run NGINX on Ubuntu.
- The `RUN` directives specify the commands we need to run on a particular layer. You can run more than one command by separating them with `&&`. We want to run multiple commands in a single line if we're going to club dependent commands in a single layer. Every layer should meet a particular objective. In the preceding example, the first `RUN` directive is used to install `curl`, while the next `RUN` directive is used to install `nginx`.
- You might be wondering why we have `apt update` before every installation. This is required, as Docker builds images using layers. So, one layer should not have implicit dependencies on the previous one. In this example, if we omit `apt update` while installing `nginx`, and if we want to update the `nginx` version without changing anything in the directive containing `apt update` (that is, the line that installs `curl`), when we run the build, `apt update` will not run again, so your `nginx` installation might fail.
- The `CMD` directive specifies a list of commands that we need to run when the built image runs as a container. This is the default command that will be executed, and its output will end up in the container logs. Your container can contain one or more `CMD` directives. For a long-running process such as NGINX, the last `CMD` should contain something that will not pass control back to the shell and continue to run for the container's lifetime. In this case, we run `nginx -g daemon off;`, which is a standard way of running NGINX in the foreground.

Some directives can easily be confused with each other, such as `ENTRYPOINT` and `CMD` or `CMD` and `RUN`. These also test how solid your Docker fundamentals are, so let's look at both.

Can we use ENTRYPPOINT instead of CMD?

Instead of CMD, you can use ENTRYPPOINT. While they serve a similar purpose, they are two very different directives. Every Docker container has a default ENTRYPPOINT - /bin/sh -c. Anything you add to CMD is appended post-ENTRYPPOINT and executed; for example, CMD ["nginx", "-g", "daemon off;"] will be generated as /bin/sh -c nginx -g daemon off;. If you use a custom ENTRYPPOINT instead, the commands you use while launching the container will be appended after it. So, if you define ENTRYPPOINT ["nginx", "-g"] and use docker run nginx daemon off;, you will get a similar result.

To get a similar result without adding any CMD arguments while launching the container, you can also use ENTRYPPOINT ["nginx", "-g", "daemon off;"].

Tip

Use ENTRYPPOINT unless there is a need for a specific CMD requirement. Using ENTRYPPOINT ensures that users cannot change the default behavior of your container, so it's a more secure alternative.

Now, let's look at RUN versus CMD.

Are RUN and CMD the same?

No, RUN and CMD are different and serve different purposes. While RUN is used to build the container and only modifies the filesystem while building it, CMD commands are only executed on the writable container layer after the container is running.

While there can be several RUN statements in a Dockerfile, each modifying the existing layer and generating the next, if a Dockerfile contains more than one CMD command, all but the last one are ignored.

The RUN directives are used to execute statements within the container filesystem to build and customize the container image, thus modifying the image layers. The idea of using a CMD command is to provide the default command(s) with the container image that will be executed at runtime. This only changes the writeable container filesystem. You can also override the commands by passing a custom command in the docker run statement.

Now, let's go ahead and build our first container image.

Building our first container

Building a container image is very simple. It is actually a one-line command: `docker build -t <image-name>:<version> <build_context>`. While we will discuss building container images in detail in the *Building and managing container images* section, let's first build the Dockerfile:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 50.0s (7/7) FINISHED
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 171B 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:bionic 2.4s
=> [1/3] FROM docker.io/library/ubuntu:bionic@sha256:152dc042... 2.8s
=> => resolve docker.io/library/ubuntu:bionic@sha256:152dc04... 0.0s
=> => sha256:152dc042... 1.33kB / 1.33kB 0.0s
=> => sha256:dca176c9... 424B / 424B 0.0s
=> => sha256:f9a80a55... 2.30kB / 2.30kB 0.0s
=> => sha256:7c457f21... 25.69MB / 25.69MB 1.0s
=> => extracting sha256:7c457f21... 1.6s
=> [2/3] RUN apt update && apt install -y curl 22.4s
=> [3/3] RUN apt update && apt install -y nginx 21.6s
=> exporting to image 0.6s
=> => exporting layers 0.6s
=> => writing image sha256:9d34cdda... 0.0s
=> => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

You might have noticed that the name of the container had a prefix in front of it. That is your Docker Hub account name. The name of the image has a structure of `<registry-url>/<account-name>/<container-image-name>:<version>`.

Here, we have the following:

- **registry-url**: The URL to the Docker registry – defaults to `docker.io`
- **account-name**: The user or account that owns the image
- **container-image-name**: The container image's name
- **version**: The image version

Now, let's create a container out of the image using the following command:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
092374c4501560e96a13444ce47cb978b961cf8701af311884bfe...
$ docker ps
CONTAINER ID   IMAGE          COMMAND       CREATED      STATUS      PORTS          NAMES
092374c45015   <your_dockerhub_ "nginx -g      28 seconds  Up 27    0.0.0.0:80->80/   loving_
                 _user>/nginx-     'daemon of..."  ago           seconds  tcp,    :::80->80/tcp  noether
                               hello-world
```

Here, we can see that the container is up and running.

If we run `curl localhost`, we get the default nginx html response:

```
$ curl localhost
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</body>
</html>
```

That's great! We have built our first image using a Dockerfile.

What if we wanted to customize the image according to our requirements? Practically speaking, no one would want an NGINX container just responding with the default `Welcome to nginx!` message, so let's create an index page and use that instead:

```
$ vim index.html
Hello World! This is my first docker image!
```

This one outputs a custom message instead of the default NGINX HTML page.

We all know that the default NGINX directory containing the `index.html` file is `/var/www/html`. If we can copy the `index.html` file into this directory, it should sort out our problem.

So, modify the Dockerfile so that it includes the following:

```
$ vim Dockerfile
FROM ubuntu:bionic
RUN apt update && apt install -y curl
RUN apt update && apt install -y nginx
WORKDIR /var/www/html/
ADD index.html .
CMD ["nginx", "-g", "daemon off;"]
```

Here, we've added two directives to the file: `WORKDIR` and `ADD`. Let's understand what each one does:

- `WORKDIR`: This defines the current working directory, which is `/var/www/html` in this case. The last `WORKDIR` in the Dockerfile also specifies the working directory when the container is executed. So, if you `exec` into a running container, you will land in the last defined `WORKDIR`. `WORKDIR` can be absolute as well as relative to the current working directory.
- `ADD`: This adds a local file to the container filesystem – the working directory, in this case. You can also use a `COPY` directive here instead of `ADD`, though `ADD` offers some more features, such as downloading files from a URL and using an archive such as a TAR or ZIP package.

When we build this file, we expect the `index.html` file to be copied to the `/var/www/html` directory within the container filesystem. Let's have a look:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 1.6s (10/10) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 211B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:bionic 1.4s
=> [1/5] FROM docker.io/library/ubuntu:bionic@sha256:152dc042... 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 81B 0.0s
=> CACHED [2/5] RUN apt update && apt install -y curl 0.0s
=> CACHED [3/5] RUN apt update && apt install -y nginx 0s
=> [4/5] WORKDIR /var/www/html/ 0.0s
=> [5/5] ADD index.html ./ 0.0s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:cb2e67bd... 0.0s
=> => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

This time, the build was much faster! When we executed the Docker build, it used a lot of layers from the cache. That is one of the advantages of layered architecture; you only build the changing part and use the existing one the way it is.

Tip

Always add source code after installing packages and dependencies. The source code changes frequently and the packages more or less remain the same. This will result in faster builds and save a lot of CI/CD time.

Let's rerun the container and see what we get. Note that you need to remove the old container before doing so:

```
$ docker ps
CONTAINER ID  IMAGE          COMMAND       CREATED      STATUS      PORTS          NAMES
092374c45015  <your_dockerhub> "nginx -g      28 seconds  Up 27    0.0.0.0:80->80/    loving_
                  _user>/nginx-
                  'daemon of..."   ago           seconds   tcp,    :::80->80/tcp  noether
$ docker rm 092374c45015 -f
092374c45015
```

At this point, we can't see the container anymore. Now, let's rerun the container using the following command:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
cc4fe116a433c505ead816fd64350cb5b25c5f3155bf5eda8cede5a4...
```

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
cc4fe116a433 <your_dockerhub> "nginx -g
 _user>/nginx- 'daemon off;" 52 seconds ago
seconds/tcp, ::80->80/tcp gates
hello-world
```

Here, we can see that the container is up and running. Let's use `curl localhost` to see what we get:

```
$ curl localhost
Hello World! This is my first docker image!
```

Here, we get a custom message instead of the default NGINX HTML response!

This looks good enough for now, but I will discuss a few more directives to make this image more reliable. First, we haven't explicitly documented what port this container should expose. This works perfectly fine, as we know that NGINX runs on port 80, but what if someone wants to use your image and doesn't know the port? In that scenario, it is best practice to define the port explicitly. We will use the `EXPOSE` directive for that.

Tip

Always use the `EXPOSE` directive to give more clarity and meaning to your image.

We also need to define the action to the container process if someone sends a `docker stop` command. While most processes take the hint and kill the process, it makes sense to explicitly specify what `STOPSIG` the container should send on a `docker stop` command. We will use the `STOPSIG` directive for that.

Now, while Docker monitors the container process and keeps it running unless it receives a `SIGTERM` or a `stop`, what would happen if your container process hangs for some reason? While your application is in a hung state, Docker still thinks it is running as your process is still running. Therefore, monitoring the application through an explicit health check would make sense. We will use the `HEALTHCHECK` directive for this.

Let's combine all these aspects and see what we get in the Dockerfile:

```
$ vim Dockerfile
FROM ubuntu:bionic
RUN apt update && apt install -y curl
RUN apt update && apt install -y nginx
WORKDIR /var/www/html/
ADD index.html .
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
STOPSIG TERM
HEALTHCHECK --interval=60s --timeout=10s --start-period=20s --retries=3 CMD curl -f
localhost
```

While `EXPOSE` and `STOPSIGALN` are self-explanatory, let's look at the `HEALTHCHECK` directive. The `HEALTHCHECK` directive runs a command (hence `CMD`) called `curl -f localhost`. So, this container will report itself as healthy until the result of the `curl` command is a success.

The `HEALTHCHECK` directive also contains the following optional fields:

- `--interval (default: 30s)`: The interval between two subsequent health checks.
- `--timeout (default: 30s)`: The health check probe timeout. If the health check times out, it implies a health check failure.
- `--start-period (default: 0s)`: The time lag between starting the container and the first health check. This allows you to ensure your container is up before a health check.
- `--retries (default: 3)`: The number of times the probe will retry before declaring an unhealthy status.

Now, let's build this container:

```
$ docker build -t <your_dockerhub_user>/nginx-hello-world .
[+] Building 1.3s (10/10) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 334B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/ubuntu:bionic 1.2s
=> [1/5] FROM docker.io/library/ubuntu@sha256:152dc0... 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 31B 0.0s
=> CACHED [2/5] RUN apt update && apt install -y curl 0.0s
=> CACHED [3/5] RUN apt update && apt install -y nginx 0s
=> CACHED [4/5] WORKDIR /var/www/html/ 0.0s
=> CACHED [5/5] ADD index.html ./ 0.0s
=> exporting to image 0.0s
=> => exporting layers 0.0s
=> => writing image sha256:bba3123d... 0.0s
=> => naming to docker.io/<your_dockerhub_user>/nginx-hello-world
```

It's time to run it and see for ourselves:

```
$ docker run -d -p 80:80 <your_dockerhub_user>/nginx-hello-world
94cbf3fdd7ff1765c92c81a4d540df3b4dbe1bd9748c91e2ddf565d8...
```

Now that we have successfully launched the container, let's try `ps` and see what we get:

\$ docker ps						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
94cbf3fdd7ff	<your_dockerhub_user>/nginx-hello-world	"nginx -g _daemon off;"	5 seconds ago	Up 4 (health: starting)	0.0.0.0:80->80/tcp	wonderful_hodgkin

As we can see, the container shows `health: starting`, which means the health check hasn't been started yet, and we are waiting for the start time to expire.

Let's wait a while and then try `docker ps` again:

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
94cbf3fdd7ff <your_dockerhub> "nginx -g      2 minutes Up 2  0.0.0.0:80->80/   wonderful_
               _user>/nginx-  'daemon of..." ago          (healthy)  tcp,  :::80->80/tcp  hodgkin
                           hello-world
```

This time, it reports the container as healthy. So, our container is now more reliable, as anyone monitoring it will know what part of the application is healthy and what part is not.

This health check only reports on the container's health status. It takes no action beyond that. You are responsible for periodically monitoring the containers and writing a script to action unhealthy containers.

One way to manage this would be to create a script that checks for unhealthy containers and restarts them. You can schedule such a script in your crontab. You can also create a long-running `systemd` script that continuously polls the container processes and checks for the health status.

Tip

While using `HEALTHCHECK` is a great option, you should avoid using it to run your containers on Kubernetes or a similar container orchestrator. You should make use of liveness and readiness probes instead. Similarly, you can define health checks on Docker Compose if you are using it, so use that instead of baking the health check into the container image.

Now, let's go ahead and learn how to build and manage Docker images.

Building and managing Docker images

We built some Docker images in the previous section, so by now, you should know how to write Dockerfiles and create Docker images from them. We've also covered a few best practices regarding it, which, in summary, are as follows:

- Always add the layers that do not change frequently first, followed by the layers that may change often. For example, install your packages and dependencies first and copy the source code later. Docker builds the Dockerfile from the part you change until the end, so if you change a line that comes later, Docker takes all the existing layers from the cache. Adding more frequently changing parts later in the build helps reduce the build time and will result in a faster CI/CD experience.
- Combine multiple commands to create as few layers as possible. Avoid multiple consecutive `RUN` directives. Instead, combine them into a single `RUN` directive using the `&&` clauses. This will help reduce the overall container footprint.

- Only add the required files within your container. Your container does not need the heavyweight package managers and the Go toolkit while running your containers if you have already compiled the code into a binary. We will discuss how to do this in detail in the following sections.

Docker images are traditionally built using a sequence of steps specified in the Dockerfile. But as we already know, Docker is DevOps-compliant and uses config management practices from the beginning. Most people build their code within the Dockerfile. Therefore, we will also need the programming language library in the build context. With a simple sequential Dockerfile, these programming language tools and libraries end up within the container image. These are known as single-stage builds, which we will cover next.

Single-stage builds

Let's containerize a simple Go application that prints `Hello, World!` on the screen. While I am using **Golang** in this application, this concept is applicable universally, irrespective of the programming language.

The respective files for this example are present in the `ch4/go-hello-world/single-stage` directory within this book's GitHub repository.

Let's look at the Go application file, `app.go`, first:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
```

The Dockerfile appears as follows:

```
FROM golang:1.20.5
WORKDIR /tmp
COPY app.go .
RUN GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app
CMD ["./app"]
```

This is standard stuff. We take the `golang:1.20.5` base image, declare a `WORKDIR /tmp`, copy `app.go` from the host filesystem to the container, and build the Go application to generate a binary. Finally, we use the `CMD` directive with the generated binary to be executed when we run the container.

Let's build the Dockerfile:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:single_stage .
[+] Building 10.3s (9/9) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 189B 0.0s
=> [internal] load .dockerignore 0.0s
```

```
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/golang:1.20.5 0.6s
=> [1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d... 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 27B 0.0s
=> [2/4] WORKDIR /tmp 0.0s
=> [3/4] COPY app.go . 0.0s
=> [4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app 9.3s
=> exporting to image 0.3s
=> => exporting layers 0.3s
=> => writing image sha256:3fd3d261... 0.0s
=> => naming to docker.io/<your_dockerhub_user>/go-hello-world:single_stage
```

Now, let's run the Docker image and see what we get:

```
$ docker run <your_dockerhub_user>/go-hello-world:single_stage
Hello, World!
```

We get the expected response back. Now, let's run the following command to list the image:

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
<your_dockerhub_user>				
/go-hello-world	single_stage	3fd3d26111a1	3 minutes ago	803MB

This image is huge! It takes 803 MB to print Hello, World! on the screen. This is not the most efficient way of building Docker images.

Before we look at the solution, let's understand why the image is so bloated in the first place. We use the Golang base image, which contains the entire Go toolkit and generates a simple binary. We do not need the complete Go toolkit for this application to run; it can efficiently run in an Alpine Linux image.

Docker solves this problem by providing multi-stage builds. You can split your build into stages where you can build your code in one stage and then, in the second stage, export the built code to another context that begins with a different base image that is much lighter and only contains those files and components that we need to run the code. We'll have a look at this in the next section.

Multi-stage builds

Let's modify the Dockerfile according to the multi-stage build process and see what we get.

The respective files for this example are present in the ch4/go-hello-world/multi-stage directory within this book's GitHub repository.

The following is the Dockerfile:

```
FROM golang:1.20.5 AS build
WORKDIR /tmp
```

```
COPY app.go .
RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app

FROM alpine:3.18.0
WORKDIR /tmp
COPY --from=build /tmp/app .
CMD ["./app"]
```

The Dockerfile contains two FROM directives: FROM golang:1.20.5 AS build and FROM alpine:3.18.0. The first FROM directive also includes an AS directive that declares the stage and names it build. Anything we do after this FROM directive can be accessed using the build term until we encounter another FROM directive, which would form the second stage. Since the second stage is the one we want to run our image from, we are not using an AS directive.

In the first stage, we build our Golang code to generate the binary using the golang base image.

In the second stage, we use the Alpine base image and copy the /tmp/app file from the build stage into our current stage. This is the only file we need to run in the container. The rest were only required to build and bloat our container during runtime.

Let's build the image and see what we get:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:multi_stage
[+] Building 12.9s (13/13) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 259B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/alpine:3.18.0 2.0s
=> [internal] load metadata for docker.io/library/golang:1.20.5 1.3s
=> [build 1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d... 0.0s
=> [stage-1 1/3] FROM docker.io/library/alpine:3.18.0@sha256:02bb6f42... 0.1s
=> => resolve docker.io/library/alpine:3.18.0@sha256:02bb6f42... 0.0s
=> => sha256:c0669ef3... 528B / 528B 0.0s
=> => sha256:5e2b554c... 1.47kB / 1.47kB 0.0s
=> => sha256:02bb6f42... 1.64kB / 1.64kB 0.0s
=> CACHED [build 2/4] WORKDIR /tmp 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 108B 0.0s
=> [build 3/4] COPY app.go . 0.0s
=> [build 4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . &&
chmod +x ./app 10.3s
=> [stage-1 2/3] WORKDIR /tmp 0.1s
=> [stage-1 3/3] COPY --from=build /tmp/app . 0.3s
=> exporting to image 0.1s
=> => exporting layers 0.1s
```

```
=> => writing image sha256:e4b793b3... 0.0s
=> => naming to docker.io/<your_dockerhub_user>/go-hello-world:multi_stage
```

Now, let's run the container:

```
$ docker run <your_dockerhub_user>/go-hello-world:multi_stage .
Hello, World!
```

We get the same output, but this time with a minimal footprint. Let's look at the image to confirm this:

\$ docker images					
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE	
<your_dockerhub_user>					
/go-hello-world	multi_stage	e4b793b39a8e	5 minutes ago	9.17MB	

This one occupies just 9.17 MB instead of the huge 803 MB. This is a massive improvement! We have reduced the image size by almost 100 times.

That is how we increase efficiency within our container image. Building efficient images is the key to running production-ready containers, and most professional images you find on Docker Hub use multi-stage builds to create efficient images.

Tip

Use multi-stage builds where possible to include minimal content within your image. Consider using an Alpine base image if possible.

In the next section, we will look at managing images within Docker, some best practices, and some of the most frequently used commands.

Managing Docker images

In modern DevOps practices, Docker images are primarily built either on a developer machine or a CI/CD pipeline. The images are stored in a container registry and then deployed to multiple staging environments and production machines. They might run Docker or a container orchestrator, such as Kubernetes, on top of them.

To efficiently use images, we must understand how to tag them.

Primarily, Docker pulls the image once when you do a Docker run. This means that once an image with a particular version is on the machine, Docker will not attempt to pull it on every run unless you explicitly pull it.

To pull the image explicitly, you can use the `docker pull` command:

```
$ docker pull nginx
Using default tag: latest
```

```
latest: Pulling from library/nginx
f03b40093957: Pull complete
eed12bbd6494: Pull complete
fa7eb8c8eee8: Pull complete
7ff3b2b12318: Pull complete
0f67c7de5f2c: Pull complete
831f51541d38: Pull complete
Digest: sha256:af296b18...
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Now, if we attempt to launch a container using this image, it will instantly launch the container without pulling the image:

```
$ docker run nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
...
2023/06/10 08:09:07 [notice] 1#1: start worker processes
2023/06/10 08:09:07 [notice] 1#1: start worker process 29
2023/06/10 08:09:07 [notice] 1#1: start worker process 30
```

So, using the latest tag on an image is a bad idea, and the best practice is to use semantic versions as your tag. There are two primary reasons for this:

- If you build the latest image every time, orchestrators such as Docker Compose and Kubernetes will assume the image is already on your machine and will not pull your image by default. Using an image pull policy such as Always on Kubernetes or a script to pull the image is a waste of network bandwidth. It is also important to note that Docker Hub limits the number of pulls you can make on open source images, so you must limit your pulls to only when necessary.
- Docker tags allow you to roll out or roll back your container deployment quickly. If you always use the latest tag, the new build overrides the old one, so there is no way you can roll back a faulty container to the last known good version. Using versioned images in production is also a good idea to ensure your container's stability. If, for some reason, you lose the local image and decide to rerun your container, you may not get the same version of the software you were already running, as the latest tag changes frequently. So, it's best to use a particular container version in production for stability.

Images comprise multiple layers, and most of the time, there is a relationship between various versions of containers that run on your server. With time, new versions of images roll out in your production environment, so removing the old images by doing some housekeeping is best. This will reclaim some valuable space the container images occupy, resulting in a cleaner filesystem.

To remove a particular image, you can use the `docker rmi` command:

```
$ docker rmi nginx
Error response from daemon: conflict: unable to remove repository reference "nginx" (must
force) - container d5c84356116f is using its referenced image f9c14fe76d50
```

Oh! We get an error, but why? It's because we have a container running and using this image.

Tip

You cannot remove images currently used by a running container.

First, you will have to stop and remove the container. Then, you can remove the image using the preceding command. If you want to do everything at once, you can force removal by using the `-f` flag, which will stop the container, remove it, and then remove the image. So, unless you know what you are doing, do not use the `-f` flag:

```
$ docker rmi -f nginx
Untagged: nginx:latest
Untagged: nginx@sha256:af296b18...
Deleted: sha256:f9c14fe7...
```

We built our container many times, but what should we do if we need to push it to Docker Hub or other registries? But before we do that, we will have to authenticate it with Docker Hub using the following command:

```
$ docker login
```

Now, you can push the image to Docker Hub using the following command:

```
$ docker push <your_dockerhub_user>/nginx-hello-world:latest
The push refers to repository [docker.io/<your_dockerhub_user>/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Mounted from library/ubuntu
latest: digest: sha256:11ec56f0... size: 1366
```

This has pushed four layers and mounted the rest from Ubuntu. We used Ubuntu as the base image, which is already available on Docker Hub.

If you have multiple tags for the image and you want to push all of them, then you can use the `-a` or `--all-tags` option in the `push` command. This will push all the tags for that particular image:

```
$ docker push -a <your_dockerhub_user>/go-hello-world
The push refers to repository [docker.io/<your_dockerhub_user>/go-hello-world]
9d61dbd763ce: Pushed
```

```
5f70bf18a086: Mounted from <your_dockerhub_user>/nginx-hello-world
bb01bd7e32b5: Mounted from library/alpine
multi_stage: digest: sha256:9e1067ca... size: 945
445ef31efc24: Pushed
d810ccdfdc04: Pushed
5f70bf18a086: Layer already exists
70ef08c04fa6: Mounted from library/golang
41cf9ea1d6fd: Mounted from library/golang
d4ebbc3dd11f: Mounted from library/golang
b4b4f5c5ff9f: Mounted from library/golang
b0df24a95c80: Mounted from library/golang
974e52a24adf: Mounted from library/golang
single_stage: digest: sha256:08b5e52b... size: 2209
```

When your build fails for some reason and you make changes to your Dockerfile, it's possible that the old images' layers will remain dangling. Therefore, it is best practice to prune the dangling images at regular intervals. You can use `docker images prune` for this:

```
$ docker images prune
REPOSITORY TAG IMAGE ID CREATED SIZE
```

In the next section, we'll look at another way to improve Docker image efficiency: flattening Docker images.

Flattening Docker images

Docker inherently uses a layered filesystem, and we have already discussed why it is necessary and how it is beneficial in depth. However, in some particular use cases, Docker practitioners have observed that a Docker image with fewer layers performs better. You can reduce layers in an image by flattening it. However, it is still not a best practice, and you need to do this only if you see a performance improvement, as this would result in a filesystem overhead.

To flatten a Docker image, follow these steps:

1. Run a Docker container with the usual image.
2. Do a `docker export` of the running container to a `.tar` file.
3. Do a `docker import` of the `.tar` file into another image.

Let's use the `nginx-hello-world` image to flatten it and export it to another image; that is, `<your_dockerhub_user>/nginx-hello-world:flat`.

Before we move on, let's get the history of the latest image:

```
$ docker history <your_dockerhub_user>/nginx-hello-world:latest
IMAGE      CREATED     CREATED BY          SIZE      COMMENT
bba3123dde01  2 hours ago  HEALTHCHECK &
                  {"CMD-SHELL"
                               "curl -f localhost...    0B      buildkit.dockerfile.v0
<missing>   2 hours ago  STOPSIGALN          0B      buildkit.dockerfile.v0
<missing>   2 hours ago  SIGTERM             0B      buildkit.dockerfile.v0
<missing>   2 hours ago  CMD ["nginx"
                               "-g" "daemon off;"]    0B      buildkit.dockerfile.v0
<missing>   2 hours ago  EXPOSE map[80/
                               tcp:{}]              0B      buildkit.dockerfile.v0
<missing>   2 hours ago  ADD index.html ./ #
                               buildkit                44B     buildkit.dockerfile.v0
<missing>   2 hours ago  WORKDIR /var/www/
                               html/                 0B      buildkit.dockerfile.v0
<missing>   2 hours ago  RUN /bin/sh -c apt
                               update && apt           57.2MB   buildkit.dockerfile.v0
                               install -y...
<missing>   2 hours ago  RUN /bin/sh -c apt
                               update && apt           59.8MB   buildkit.dockerfile.v0
                               install -y...
<missing>  10 days ago  /bin/sh -c #(nop)    0B
                               CMD ["/bin/bash"]
<missing>  10 days ago  /bin/sh -c #(nop) ADD
                               file:3c74e7e08cbf9a876...
<missing>  10 days ago  /bin/sh -c #(nop)  LABEL  0B
                               org.opencontainers...
<missing>  10 days ago  /bin/sh -c #(nop)  LABEL  0B
                               org.opencontainers...
<missing>  10 days ago  /bin/sh -c #(nop)  ARG    0B
                               LAUNCHPAD_BUILD_ARCH
<missing>  10 days ago  /bin/sh -c #(nop)  ARG RELEASE
                               ARG RELEASE
```

Now, let's run a Docker image with the latest image:

```
$ docker run -d --name nginx <your_dockerhub_user>/nginx-hello-world:latest
e2d0c4b884556a353817aada13f0c91ecfeb01f5940e91746f168b...
```

Next, let's take an export out of the running container:

```
$ docker export nginx > nginx-hello-world-flat.tar
```

Import `nginx-hello-world-flat.tar` to a new image; that is, `<your_dockerhub_user>/nginx-hello-world:flat`:

```
$ cat nginx-hello-world-flat.tar | \
docker import - <your_dockerhub_user>/nginx-hello-world:flat
sha256:57bf5a9ada46191aelaa16bcf837a4a80e8a19d0bcb9fc...
```

Now, let's list the images and see what we get:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
<your_dockerhub_user>/ flat    57bf5a9ada46  34 seconds  177MB
nginx-hello-world
                    ago
<your_dockerhub_user>/ latest   bba3123dde01  2 hours
nginx-hello-world
                    ago           180MB
```

Here, we can see that the flat image is present and that it occupies less space than the latest image. If we view its history, we should see just a single layer:

```
$ docker history <your_dockerhub_user>/nginx-hello-world:flat
IMAGE      CREATED      CREATED BY      SIZE      COMMENT
57bf5a9ada46  About a minute ago            177MB  Imported from -
```

It has flattened the image. But is it a best practice to flatten Docker images? Well, it depends. Let's understand when and how to flatten Docker images and what you should consider:

- Are several applications using a similar base image? If that is the case, then flattening images will only increase the disk footprint, as you won't be able to take advantage of a layered filesystem.
- Consider alternatives to flattening images using a small base image, such as Alpine.
- Multi-stage builds are helpful for most complied languages and can reduce your image's size considerably.
- You can also slim down images by using as few layers as possible by combining multiple steps into a single RUN directive.
- Consider whether the benefits of flattening the image outweigh the disadvantages, whether you'll get considerable performance improvements, and whether performance is critical for your application needs.

These considerations will help you understand your container image footprint and help you manage container images. Remember that although reducing the image's size is ideal, flattening it should be a last resort.

So far, all the images we've used have been derived from a Linux distribution and always used a distro as their base image. You can also run a container without using a Linux distro as the base image to make it more secure. We'll have a look at how in the next section.

Optimizing containers with distroless images

Distroless containers are one of the latest trends in the container world. They are promising because they consider all the aspects of optimizing containers for the Enterprise environment. You should consider three important things while optimizing containers – performance, security, and cost.

Performance

You don't make containers out of thin air. You must download images from your container registry and then run the container out of the image. Each step uses network and disk I/O. The bigger the image, the more resources it consumes and the less performance you get from it. Therefore, a smaller Docker image naturally performs better.

Security

Security is one of the most important aspects of the current IT landscape. Companies usually focus on this aspect and invest a lot of money and time. Since containers are a relatively new technology, they are vulnerable to hacking, so appropriately securing your containers is important. Standard Linux distributions have a lot of stuff that can allow hackers to access more than they could have if you secured your container properly. Therefore, you must ensure you only have what you need within the container.

Cost

A smaller image also results in a lower cost. The lower your container footprint, the more containers you can pack within a machine, so there are fewer machines you would need to run your applications. This means you save a lot of money that would accumulate over time.

As a modern DevOps engineer, you must ensure your images are optimized for all these aspects. Distroless images help take care of all of them. Therefore, let's understand what distroless images are and how to use them.

Distroless images are the most minimal images and only contain your application, dependencies, and the necessary files for your container process to run. Most of the time, you do not need package managers such as `apt` or a shell such as `bash`. Not having a shell has its advantages. For one, it will help you avoid any outside party gaining access to your container while it is running. Your container has a small attack surface and won't have many security vulnerabilities.

Google provides distroless images in their official GCR registry, available on their GitHub page at <https://github.com/GoogleContainerTools/distroless>. Let's get hands-on and see what we can do with them.

The required resources for this exercise are in `ch4/go-hello-world/distroless` in this book's GitHub repository.

Let's start by creating a Dockerfile:

```
FROM golang:1.20.5 AS build
WORKDIR /tmp
COPY app.go .
RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o app . && chmod +x ./app
FROM gcr.io/distroless/base
WORKDIR /tmp
```

```
COPY --from=build /tmp/app .
CMD ["./app"]
```

This Dockerfile is similar to the multi-stage build Dockerfile for the go-hello-world container, but instead of using alpine, it uses gcr.io/distroless/base as the base image. This image contains a minimalistic Linux glibc-enabled system and lacks a package manager or a shell. You can use it to run binaries compiled in a language such as Go, Rust, or D.

So, let's build this first using the following command:

```
$ docker build -t <your_dockerhub_user>/go-hello-world:distroless .
[+] Building 7.6s (14/14) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> => transferring dockerfile: 268B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for gcr.io/distroless/base:latest 3.1s
=> [internal] load metadata for docker.io/library/golang:1.20.5 1.4s
=> [auth] library/golang:pull token for registry-1.docker.io 0.0s
=> [stage-1 1/3] FROM gcr.io/distroless/base@
sha256:73deaaf6a207c1a33850257ba74e0f196bc418636cada9943a03d7abea980d6d 3.2s
=> [build 1/4] FROM docker.io/library/golang:1.20.5@sha256:4b1fc02d 0.0s
=> [internal] load build context 0.0s
=> => transferring context: 108B 0.0s
=> CACHED [build 2/4] WORKDIR /tmp 0.0s
=> CACHED [build 3/4] COPY app.go . 0.0s
=> CACHED [build 4/4] RUN GO111MODULE=off GOOS=linux go build -a -installsuffix cgo -o
app . && chmod +x ./app 0.0s
=> [stage-1 2/3] WORKDIR /tmp 0.9s
=> [stage-1 3/3] COPY --from=build /tmp/app . 0.3s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image sha256:51ced401 0.0s
=> => naming to docker.io/<your_dockerhub_user>/go-hello-world:distroless
```

Now, let's run this image and see what we get:

```
$ docker run <your_dockerhub_user>/go-hello-world:distroless
Hello, World!
```

It works! Let's look at the size of the image:

```
$ docker images
REPOSITORY                      TAG      IMAGE ID      CREATED       SIZE
<your_dockerhub_user>/go-hello-world  distroless  51ced401d7bf  6 minutes ago  22.3MB
```

It's just 22.3 MB. Yes, it's a bit more than the Alpine image, but it does not contain a shell, so it is more secure from that point of view. Also, there are distroless images available for interpreted programming languages, such as Python and Java, that you can use instead of the bloated image containing the toolkits

Docker images are stored in Docker registries, and we have all been using Docker Hub for a while. In the next section, we'll understand what they are and what our options are for storing our images.

Understanding Docker registries

A **Docker registry** is a stateless, highly scalable server-side application that stores and lets you distribute Docker images. The registry is open source under the permissive **Apache license**. It is a storage and distribution system where all your Docker servers can connect and upload and download images as and when needed. It acts as a distribution site for your images.

A Docker registry contains several Docker repositories. A Docker repository holds several versions of a specific image. For example, all the versions of the `nginx` image are stored within a single repository within Docker Hub called `nginx`.

By default, Docker interacts with its public Docker registry instance, called Docker Hub, which helps you distribute your images to the broader open source community.

Not all images can be public and open source, and many proprietary activities are ongoing. Docker allows you to use a private Docker registry for a scenario you can host within your infrastructure called **Docker Trusted Registry**. Several online options are available, including using a SaaS service, such as GCR, or creating private repositories at Docker Hub.

While the SaaS option is readily available and intuitive, let's consider hosting our private Docker registry.

Hosting your private Docker registry

Docker provides an image that you can run on any server that has Docker installed. Once the container is up and running, you can use that as the Docker registry. Let's have a look:

```
$ docker run -d -p 80:5000 --restart=always --name registry registry:2
Unable to find image 'registry:2' locally
2: Pulling from library/registry
8a49fdb3b6a5: Already exists
58116d8bf569: Pull complete
4cb4a93be51c: Pull complete
cbdeff65a266: Pull complete
6b102b34ed3d: Pull complete
Digest: sha256:20d08472...
Status: Downloaded newer image for registry:2
ae4c4ec9fc7b17733694160b5b3b053bd1a41475dc4282f3eccaa10...
```

Since we know that the registry is running on localhost and listening on port 80, let's try to push an image to this registry. First, let's tag the image to specify `localhost` as the registry. We will add a registry location at the beginning of the Docker tag so that Docker knows where to push the image. We already know that the structure of a Docker tag is `<registry_url>/<user>/<image_name>:<image_version>`. We will use the `docker tag` command to give another name to an existing image, as shown in the following command:

```
$ docker tag your_dockerhub_user>/nginx-hello-world:latest \
localhost/<your_dockerhub_user>/nginx-hello-world:latest
```

Now, we can go ahead and push the image to the local Docker registry:

```
$ docker push localhost/<your_dockerhub_user>/nginx-hello-world:latest
The push refers to repository [localhost/your_dockerhub_user/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Pushed
latest: digest: sha256:6ad07e74... size: 1366
```

And that's it! It is as simple as that!

There are other considerations as well since this is too simplistic. You will also have to mount volumes; otherwise, you will lose all the images when you restart the registry container. Also, there is no authentication in place, so anyone accessing this server can push or pull images, but we don't desire this. Also, communication is insecure, and we want to encrypt the images during transit.

First, let's create the local directories that we will mount to the containers:

```
$ sudo mkdir -p /mnt/registry/certs
$ sudo mkdir -p /mnt/registry/auth
$ sudo chmod -R 777 /mnt/registry
```

Now, let's generate an `htpasswd` file for adding authentication to the registry. For this, we will run the `htpasswd` command from within a new Docker registry container to create a file on our local directory:

```
$ docker run --entrypoint htpasswd registry:2.7.0 \
-Bn user pass > /mnt/registry/auth/htpasswd
```

The next step is to generate some self-signed certificates for enabling TLS on the repository. Add your server name or IP when asked for a **Fully Qualified Domain Name (FQDN)**. You can leave the other fields blank or add appropriate values for them:

```
$ openssl req -newkey rsa:4096 -nodes -sha256 -keyout \
/mnt/registry/certs/domain.key -x509 -days 365 -out /mnt/registry/certs/domain.crt
```

Before we proceed further, let's remove the existing registry:

```
$ docker rm -f registry
registry
```

Now, we are ready to launch our container with the required configuration:

```
$ docker run -d -p 443:443 --restart=always \
--name registry \
```

```
-v /mnt/registry/certs:/certs \
-v /mnt/registry/auth:/auth \
-v /mnt/registry/registry:/var/lib/registry \
-e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
-e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
-e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \
-e REGISTRY_AUTH=htpasswd \
-e "REGISTRY_AUTH_HTPASSWD_REALM=Registry Realm" \
-e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
registry:2
02bf92c9c4a6d1d9c9f4b75ba80e82834621b1570f5f7c4a74b215960
```

The container is now up and running. Let's use `https` this time, but before that, let's `docker login` to the registry. Add the username and password you set while creating the `htpasswd` file (in this case, `user` and `pass`):

```
$ docker login https://localhost
Username: user
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/
#credentials-store
Login Succeeded
```

Since the login succeeded, we can go ahead and push our image to the registry:

```
$ docker push localhost/<your_dockerhub_user>/nginx-hello-world
The push refers to repository [localhost/<your_dockerhub_user>/nginx-hello-world]
2b7de406bdcd: Pushed
5f70bf18a086: Pushed
845348333310: Pushed
96a9e6a097c6: Pushed
548a79621a42: Pushed
latest: digest: sha256:6ad07e7425331456a3b8ea118bce36c82af242ec14072d483b5dcaa3bd607e65
size: 1366
```

This time, it works the way we want it to.

Other public registries

Apart from running your registry in a dedicated Docker server, other cloud and on-premises options exist.

Most public cloud providers offer paid online registries and container-hosting solutions that you can easily use while running in the cloud. Some of them are as follows:

- **Amazon Elastic Container Registry (ECR):** This is a popular AWS offering you can use if your infrastructure runs on AWS. It is a highly available, highly performant, fully managed solution.

It can host public and private registries, and you only pay for the storage you consume and the amount of data transferred to the internet. The best part is that it integrates with AWS IAM.

- **Google Container Registry (GCR):** Backed by **Google Cloud Storage (GCS)**, GCR is one of the best choices if you run your infrastructure on GCP. It hosts both public and private repositories, and you only pay for the storage on GCS.
- **Azure Container Registry (ACR):** This fully managed, geo-replicated container registry only supports a private registry. It is a good option if you are running your infrastructure on Azure. Besides storing container images, it also stores Helm charts and other artifacts that help you manage your containers.
- **Oracle Cloud Infrastructure Registry:** Oracle Cloud Infrastructure Registry is a highly available Oracle-managed container registry. It can host both public and private repositories.
- **CoreOS Quay:** This supports OAuth and LDAP authentication. It offers both (paid) private and (free) public repositories, automatic security scanning, and automated image builds via integration with GitLab, GitHub, and Bitbucket.

If you don't want to go with managed options in the cloud or run on-premises, you can also use distribution management software such as *Sonatype Nexus* or *JFrog Artifactory*. Both tools support Docker registries out of the box. You can create a Docker registry there using fancy UIs, and then use `docker login` to connect to the registry.

Summary

In this chapter, we have covered a lot of ground. At this point, you should understand Docker from a hands-on perspective. We started with Docker images, how to use a Dockerfile to build Docker images, the components and directives of the Dockerfile, and how to create efficient images by following some best practices. We also discussed flattening Docker images and improving container security using distroless images. Finally, we discussed Docker registries, how to run a private Docker registry on a Docker server, and how to use other turnkey solutions, such as Sonatype Nexus and JFrog Artifactory.

Here is a quick summary of some best practices for managing Docker containers effectively and efficiently:

- **Use Official Images:** Whenever possible, start with official Docker images from reputable sources such as Docker Hub. These images are well-maintained, regularly updated, and often come with better security practices.
- **Minimize Containers:** Follow the “one service per container” principle. Each container should have a single responsibility, which helps with maintainability and scaling.
- **Optimize Container Sizes:** Keep containers as lightweight as possible. Use Alpine Linux or other minimal base images and remove unnecessary files and dependencies.
- **Use Environment Variables:** Store configuration and sensitive data in environment variables rather than hardcoding it into the container. This enhances portability and security.

- **Persistent Data:** Store application data outside containers using Docker volumes or bind mounts. This ensures that data persists even if containers are replaced or stopped.
- **Container Naming:** Give containers meaningful and unique names. This helps with easy identification and troubleshooting.
- **Resource Limits:** Set resource limits (CPU and memory) for containers to prevent one misbehaving container from affecting others on the same host.
- **Container Restart Policies:** Define restart policies to determine how containers should behave when they exit or crash. Choose the appropriate policy based on your application's requirements.
- **Docker Compose:** Use Docker Compose to define and manage multi-container applications. It simplifies the deployment and orchestration of complex setups.
- **Network Isolation:** Use Docker networks to isolate containers and control communication between them. This enhances security and manageability.
- **Health Checks:** Implement health checks in your containers to ensure they run as expected. This helps with automated monitoring and recovery.
- **Container Logs:** Redirect container logs to standard output (`stdout`) and standard error (`stderr`) streams. This makes it easier to collect and analyze logs using Docker's logging mechanisms.
- **Security Best Practices:** Keep containers up to date with security patches, avoid running containers as the root, and follow security best practices to avoid vulnerabilities.
- **Version Control Dockerfiles:** Store Dockerfiles in version control systems (e.g., Git) and regularly review and update them.
- **Container Cleanup:** Regularly remove unused containers, images, and volumes to free up disk space. Consider using tools such as Docker's built-in prune commands.
- **Orchestration Tools:** Explore container orchestration tools such as Kubernetes or Docker Swarm for managing larger and more complex container deployments.
- **Documentation:** Maintain clear and up-to-date documentation for your containers and images, including how to run them, their required environment variables, and any other configuration details.
- **Backup and Restore:** Establish backup and restore processes for container data and configuration to recover them quickly in case of failures.
- **Monitoring and Scaling:** Implement monitoring and alerting for your containers to ensure they run smoothly. Use scaling mechanisms to handle the increased load.

By following these best practices, you can ensure that your Docker container environment is well-organized, secure, maintainable, and scalable.

In the next chapter, we will delve into container orchestration using Kubernetes.

Questions

1. Docker images use a layered model. (True/False)
2. You can delete an image from a server if a container using that image is already running. (True/False)
3. How do you remove a running container from a server? (Choose two)
 - A. `docker rm <container_id>`
 - B. `docker rm -f <container_id>`
 - C. `docker stop <container_id> && docker rm <container_id>`
 - D. `docker stop -f <container_id>`
4. Which of the following options are container build best practices? (Choose four)
 - A. Always add layers that don't frequently change at the beginning of the Dockerfile.
 - B. Combine multiple steps into a single directive to reduce layers.
 - C. Only use the required files in the container to keep it lightweight and reduce the attack surface.
 - D. Use semantic versioning in your Docker tags and avoid the latest version.
 - E. Include package managers and a shell within the container, as this helps with troubleshooting a running container.
 - F. Only use an `apt update` at the start of your Dockerfile.
5. You should always flatten Docker images to a single layer. (True/False)
6. A distroless container contains a shell. (True/False)
7. What are some of the ways to improve container efficiency? (Choose four)
 - A. Try to use a smaller base image if possible, such as Alpine.
 - B. Only use multi-stage builds to add the required libraries and dependencies to the container and omit heavyweight toolkits that are not necessary.
 - C. Use distroless base images where possible.
 - D. Flatten Docker images.
 - E. Use single-stage builds to include package managers and a shell, as this will help in troubleshooting in production.
8. It is a best practice to prune Docker images from time to time. (True/False)
9. Health checks should always be baked into your Docker image. (True/False)

Answers

1. True
2. False – you cannot delete an image that is being used by a running container.
3. B, C
4. A, B, C, D
5. False – only flatten Docker images if you'll benefit from better performance.
6. False – distroless containers do not contain a shell.
7. A, B, C, D
8. True
9. False – if you're using Kubernetes or Docker Compose, use the liveness probes or define health checks with a YAML file instead.

Part 2:

Container Orchestration and Serverless

This part will build upon *Part 1* and introduce you to managing containers with container orchestration and serverless technologies. In this part, you will learn how to manage containers both on-premises and in the cloud using cutting-edge tools and technologies.

This part has the following chapters:

- *Chapter 5, Container Orchestration with Kubernetes*
- *Chapter 6, Managing Advanced Kubernetes Resources*
- *Chapter 7, Containers as a Service (CaaS) and Serverless*



5

Container Orchestration with Kubernetes

In the previous chapter, we covered creating and managing container images, where we discussed container images, Dockerfiles, and their directives and components. We also looked at the best practices for writing a Dockerfile and building and managing efficient images. We then looked at flattening Docker images and investigated in detail distroless images to improve container security. Finally, we created a private Docker registry.

Now, we will deep dive into container orchestration. We will learn how to schedule and run containers using the most popular container orchestrator – Kubernetes.

In this chapter, we're going to cover the following main topics:

- What is Kubernetes, and why do I need it?
- Kubernetes architecture
- Installing Kubernetes (Minikube and KinD)
- Understanding Kubernetes pods

Technical requirements

For this chapter, we assume you have Docker installed on a Linux machine running **Ubuntu 18.04 Bionic LTS** or later, with `sudo` access. You can follow *Chapter 3, Containerization with Docker*, for more details on how to do that.

You will also need to clone the following GitHub repository for some exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

Run the following command to clone the repository into your home directory, and `cd` into the `ch5` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
  modern-devops
$ cd modern-devops/ch5
```

As the repository contains files with placeholders, you must replace the `<your_dockerhub_user>` string with your actual Docker Hub user. Use the following commands to substitute the placeholders:

```
$ grep -rl '<your_dockerhub_user>' . | xargs sed -i -e \
's/<your_dockerhub_user>/<your actual docker hub user>/g'
```

What is Kubernetes, and why do I need it?

By now, you should understand what containers are and how to build and run containers using Docker. However, how we ran containers using Docker was not optimal from a production standpoint. Let me give you a few considerations to think about:

- As portable containers can run on any Docker machine just fine, multiple containers also share server resources to optimize resource consumption. Now, think of a microservices application that comprises hundreds of containers. How will you choose what machine to run the containers on? What if you want to dynamically schedule the containers to another machine based on resource consumption?
- Containers provide horizontal scalability as you can create a copy of the container and use a **load balancer** in front of a pool of containers. One way of doing this is to decide upfront and deploy the desired number of containers, but that isn't optimal resource utilization. What if I tell you that you need to horizontally scale your containers dynamically with traffic – in other words, by creating additional container instances to handle the extra load when there is more traffic and reducing them when there is less?
- There are container health check reports on the containers' health. What if the container is unhealthy, and you want to auto-heal it? What would happen if an entire server goes down and you want to schedule all containers running on that server to another?
- As containers mostly run within a server and can see each other, how would I ensure that only the required containers can interact with the other, something we usually do with VMs? We cannot compromise on security.
- Modern cloud platforms allow us to run autoscaling VMs. How can we utilize that from the perspective of containers? For example, if I need just one VM for my containers during the night and five during the day, how can I ensure that the machines are dynamically allocated when we need them?

- How do you manage the networking between multiple containers if they are part of a more comprehensive service mesh?

The answer to all these questions is a container orchestrator, and the most popular and *de facto* standard for that is Kubernetes.

Kubernetes is an open source container orchestrator. A bunch of Google engineers first developed it and then open sourced it to the **Cloud Native Computing Foundation (CNCF)**. Since then, the buzz around Kubernetes has not subsided, and for an excellent reason – Kubernetes with containers has changed the technology mindset and how we look at infrastructure entirely. Instead of treating servers as dedicated machines to an application or as part of an application, Kubernetes has allowed visualizing servers as an entity with a container runtime installed. When we treat servers as a standard setup, we can run virtually anything in a cluster of servers. So, you don't have to plan for **high availability (HA)**, **disaster recovery (DR)**, and other operational aspects for every application on your tech stack. Instead, you can cluster all your servers into a single unit – a Kubernetes cluster – and containerize all your applications. You can then offload all container management functions to Kubernetes. You can run Kubernetes on bare-metal servers, VMs, and as a managed service in the cloud through multiple Kubernetes-as-a-Service offerings.

Kubernetes solves these problems by providing HA, scalability, and zero downtime out of the box. It essentially performs the following functions to provide them:

- **Provides a centralized control plane for interacting with it:** The API server exposes a list of useful APIs that you can interact with to invoke many Kubernetes functions. It also provides a Kubernetes command line called **kubectl** to interact with the API using simple commands. Having a centralized control plane ensures that you can interact with Kubernetes seamlessly.
- **Interacts with the container runtime to schedule containers:** When we send the request to schedule a container to **kube-apiserver**, Kubernetes decides what server to schedule the container based on various factors and then interacts with the server's container runtime through the **kubelet** component.
- **Stores the expected configuration in a key-value data store:** Kubernetes applies the cluster's anticipated configuration and stores that in a key-value data store – **etcd**. That way, Kubernetes continuously ensures that the containers within the cluster remain in the desired state. If there is any deviation from the expected state, Kubernetes will take every action to bring it back to the desired configuration. That way, Kubernetes ensures that your containers are up and running and healthy.
- **Provides a network abstraction layer and service discovery:** Kubernetes uses a network abstraction layer to allow communication between your containers. Therefore, every container is allocated a virtual IP, and Kubernetes ensures a container is reachable from another container running on a different server. It provides the necessary networking by using an **overlay network** between the servers. From the container's perspective, all containers in the cluster behave as if they are running on the same server. Kubernetes also uses a **DNS** to allow communication

between containers through a domain name. That way, containers can interact with each other by using a domain name instead of an IP address to ensure that you don't need to change the configuration if a container is recreated and the IP address changes.

- **Interacts with the cloud provider:** Kubernetes interacts with the cloud provider to commission objects such as **load balancers** and **persistent disks**. So, if you tell Kubernetes that your application needs to persist data and define a **volume**, Kubernetes will automatically request a disk from your cloud provider and mount it to your container wherever it runs. You can also expose your application on an external load balancer by requesting Kubernetes. Kubernetes will interact with your cloud provider to spin up a load balancer and point it to your containers. That way, you can do everything related to containers by merely interacting with your Kubernetes API server.

Kubernetes comprises multiple moving parts that take over each function we've discussed. Now, let's look at the Kubernetes architecture to understand each of them.

Kubernetes architecture

Kubernetes is made of a cluster of nodes. There are two possible roles for nodes in Kubernetes – **control plane** nodes and **worker** nodes. The control plane nodes control the Kubernetes cluster, scheduling the workloads, listening to requests, and other aspects that help run your workloads and make the cluster function. They typically form the brain of the cluster.

On the other hand, the worker nodes are the powerhouses of the Kubernetes cluster and provide raw computing for running your container workloads.

Kubernetes architecture follows the client-server model via an API server. Any interaction, including internal interactions between components, happens via the Kubernetes API server. Therefore, the Kubernetes API server is known as the brain of the Kubernetes control plane.

There are other components of Kubernetes as well, but before we delve into the details, let's look at the following diagram to understand the high-level Kubernetes architecture:

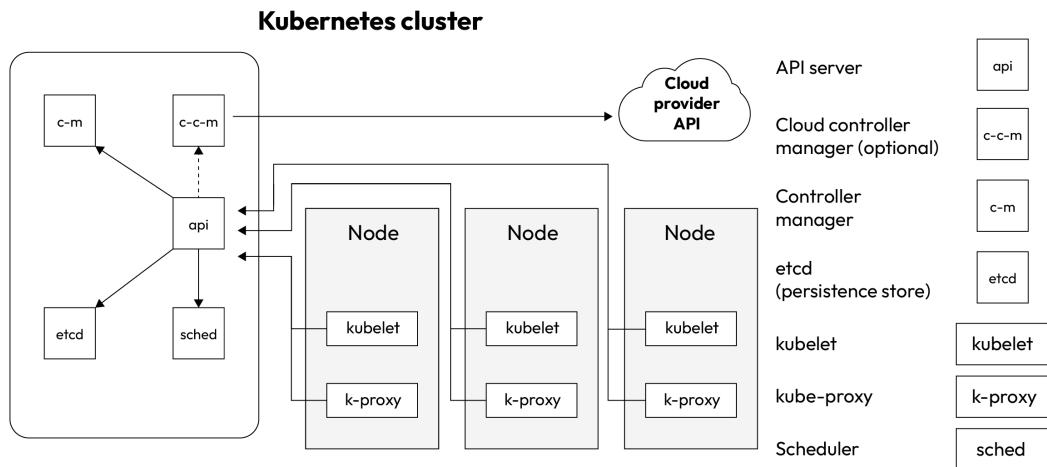


Figure 5.1 – Kubernetes cluster architecture

The control plane comprises the following components:

- **API server:** As discussed previously, the API server exposes a set of APIs for external and internal actors to interact with Kubernetes. All interactions with Kubernetes happen via the API server, as evident from the preceding diagram. If you visualize the Kubernetes cluster as a ship, the API server is the ship's captain.
- **Controller manager:** The controller manager is the ship's executive officer and is tasked with ensuring that the captain's orders are followed in the cluster. From a technical perspective, the controller manager reads the current and desired states and takes all actions necessary to move the current state to the desired state. It contains a set of controllers that interact with the Kubernetes components via the API server as and when needed. Some of these are as follows:
 - **Node controller:** This watches for when the node goes down and responds by interacting with the **Kube scheduler** via the **Kube API server** to schedule the pods to a healthy node.
 - **Replication controller:** This ensures that the correct amount of container replicas defined by replication controller objects in the cluster exist.
 - **Endpoints controller:** These assist in providing endpoints to your containers via services.
 - **Service account and token controllers:** These create default **accounts** and **tokens** for new **namespaces**.
- **Cloud controller manager:** This is an optional controller manager that you would run if you run Kubernetes in a public cloud, such as **AWS**, **Azure**, or **GCP**. The cloud controller manager interacts with the cloud provider APIs to provision resources such as **persistent disks** and **load balancers** that you declare in your Kubernetes configuration.

- **etcd:** **etcd** is the log book of the ship. That is where all the details about the expected configuration exist. From a technical perspective, this is a key-value store where all the desired Kubernetes configuration is stored. The controller manager refers to the information in this database to action changes in the cluster.
- **Scheduler:** The schedulers are the boatswain of the ship. They are tasked with supervising the process of loading and unloading containers on the ship. A Kubernetes scheduler schedules containers in a worker node it finds fit after considering the availability of resources to run it, the HA of your application, and other aspects.
- **kubelet:** kubelets are the seamen of the ship. They carry out the actual loading and unloading of containers from a ship. From a technical perspective, the kubelet interacts with the underlying container runtime to run containers on the scheduler's instruction. While most Kubernetes components can run as a container, the kubelet is the only component that runs as a **systemd** service. They usually run on worker nodes, but if you plan to run the control plane components as containers instead, the kubelet will also run on the control plane nodes.
- **kube-proxy:** **kube-proxy** runs on each worker node and provides the components for your containers to interact with the network components inside and outside your cluster. They are vital components that facilitate Kubernetes networking.

Well, that's a lot of moving parts, but the good news is that tools are available to set that up for you, and provisioning a Kubernetes cluster is very simple. If you are running on a public cloud, it is only a few clicks away, and you can use your cloud's web UI or CLI to provision it very quickly. You can use **kubeadm** for the setup if you have an on-premises installation. The steps are well documented and understood and won't be too much of a hassle.

For development and your CI/CD environments, you can use **Minikube** or **Kubernetes in Docker (KinD)**. While Minikube can run a single-node Kubernetes cluster on your development machine directly by using your machine as the node, it can also run a multi-node cluster by running Kubernetes nodes as containers. KinD, on the other hand, exclusively runs your nodes as containers on both single-node and multi-node configurations. You need a VM with the requisite resources in both cases, and you'll be good to go.

In the next section, we'll boot a single-node Kubernetes cluster with Minikube.

Installing Kubernetes (Minikube and KinD)

Now, let's move on and install Kubernetes for your development environment. We will begin with Minikube to get you started quickly and then look into KinD. We will then use KinD for the rest of this chapter.

Installing Minikube

We will install Minikube in the same Linux machine we used to install Docker in *Chapter 3, Containerization with Docker*. So, if you haven't done that, please go to *Chapter 3, Containerization with Docker*, and follow the instructions provided to set up Docker on your machine.

First, we will install **kubectl**. As described previously, kubectl is the command-line utility that interacts with the Kubernetes API server. We will use kubectl multiple times in this book.

To download the latest release of kubectl, run the following command:

```
$ curl -LO "https://storage.googleapis.com/kubernetes-release/release\\n/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)\\n/bin/linux/amd64/kubectl"
```

You can also download a specific version of kubectl. To do so, use the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release\\n/v<kubectl_version>/bin/linux/amd64/kubectl
```

We will stick with the latest release for this chapter. Now, let's go ahead and make the binary executable and then move it to any directory in your system PATH:

```
$ chmod +x ./kubectl\n$ sudo mv kubectl /usr/local/bin/
```

Now, let's check whether kubectl has been successfully installed by running the following command:

```
$ kubectl version --client\nClient Version: version.Info{Major:"1", Minor:"27", GitVersion:"v1.27.3"}
```

Since kubectl was installed successfully, you must download the minikube binary and then move it to your system path using the following commands:

```
$ curl -Lo minikube \\nhttps://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64\n$ chmod +x minikube\n$ sudo mv minikube /usr/local/bin/
```

Now, let's install the packages required by Minikube to function correctly by running the following command:

```
$ sudo apt-get install -y conntrack
```

Finally, we can bootstrap a Minikube cluster using the following command:

```
$ minikube start --driver=docker\nDone! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

As Minikube is now up and running, we will use the `kubectl` command-line utility to interact with the Kube API server to manage Kubernetes resources. The `kubectl` commands have a standard structure and are self-explanatory in most cases. They are structured as follows:

```
kubectl <verb> <resource type> <resource name> [--flags]
```

Here, we have the following:

- `verb`: The action to perform – for example, `get`, `apply`, `delete`, `list`, `patch`, `run`, and so on
- `resource type`: The Kubernetes resource to manage, such as `node`, `pod`, `deployment`, `service`, and so on
- `resource name`: The name of the resource to manage

Now, let's use `kubectl` to get nodes and check whether our cluster is ready to run our containers:

```
$ kubectl get nodes
NAME      STATUS    ROLES     AGE      VERSION
minikube  Ready     control-plane   2m25s   v1.26.3
```

Here, we can see that it is a single-node Kubernetes cluster running version **v1.26.3**. Kubernetes is now up and running!

This setup is excellent for development machines where developers want to deploy and test a single component they are working on.

To stop the Minikube cluster and delete it from the machine, you can use the following command:

```
$ minikube stop
```

Now that we have removed Minikube, let's look at another exciting tool for creating a multi-node Kubernetes cluster.

Installing KinD

KinD allows you to run a multi-node Kubernetes cluster on a single server that runs Docker. We understand that a multi-node Kubernetes cluster requires multiple machines, but how can we run a multi-node Kubernetes cluster on a single server? The answer is simple: KinD uses a Docker container as a Kubernetes node. So, if we need a four-node Kubernetes cluster, KinD will spin up four containers that behave like four Kubernetes nodes. It is as simple as that.

While you need Docker to run KinD, KinD internally uses **containerd** as a container runtime instead of Docker. Containerd implements the container runtime interface; therefore, Kubernetes does not require any specialized components, such as **dockershim**, to interact with it. This means that KinD still works with Kubernetes since Docker isn't supported anymore as a Kubernetes container runtime.

As KinD supports a multi-node Kubernetes cluster, you can use it for your development activities and also in your CI/CD pipelines. In fact, KinD redefines CI/CD pipelines as you don't require a static Kubernetes environment to test your build. KinD is swift to boot up, which means you can integrate the bootstrapping of the KinD cluster, run and test your container builds within the cluster, and then destroy it all within your CI/CD pipeline. This gives development teams immense power and speed.

Important

Never use KinD in production. Docker in Docker implementations are not very secure; therefore, KinD clusters should not exist beyond your dev environments and CI/CD pipelines.

Bootstrapping KinD is just a few commands away. First, we need to download KinD, make it executable, and then move it to the default PATH directory using the following commands:

```
$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64  
$ chmod +x kind  
$ sudo mv kind /usr/local/bin/
```

To check whether KinD is installed, we can run the following command:

```
$ kind version  
kind v0.20.0 go1.20.4 linux/amd64
```

Now, let's bootstrap a multi-node KinD cluster. First, we need to create a KinD config file. The KinD config file is a simple YAML file where you can declare what configuration you want for each node. If we need to bootstrap a single control plane and three worker node clusters, we can add the following configuration:

```
$ vim kind-config.yaml  
kind: Cluster  
apiVersion: kind.x-k8s.io/v1alpha4  
nodes:  
- role: control-plane  
- role: worker  
- role: worker  
- role: worker
```

You can also have an HA configuration with multiple control planes using multiple node items with the control plane role. For now, let's stick with a single control plane, three-worker node configuration.

To bootstrap your KinD cluster with the preceding configuration, run the following command:

```
$ kind create cluster --config kind-config.yaml
```

With that, our KinD cluster is up and running. Now, let's list the nodes to see for certain by using the following command:

```
$ kubectl get nodes
NAME           STATUS   ROLES      AGE    VERSION
kind-control-plane   Ready    control-plane   72s    v1.27.3
kind-worker     Ready    <none>     47s    v1.27.3
kind-worker2    Ready    <none>     47s    v1.27.3
kind-worker3    Ready    <none>     47s    v1.27.3
```

Here, we can see four nodes in the cluster – one control plane and three workers. Now that the cluster is ready, we'll dive deep into Kubernetes and look at some of the most frequently used Kubernetes resources in the next section.

Understanding Kubernetes pods

Kubernetes pods are the basic building blocks of a Kubernetes application. A pod contains one or more containers, and all containers within a pod are always scheduled in the same host. Usually, there is a single container within a pod, but there are use cases where you need to schedule multiple containers in a single pod.

It takes a while to digest why Kubernetes started with the concept of pods in the first place instead of using containers, but there are reasons for that, and you will appreciate this as you gain more experience with the tool. For now, let's look at a simple example of a pod and how to schedule it in Kubernetes.

Running a pod

We will start by running an NGINX container in a pod using simple imperative commands. We will then look at how we can do this declaratively.

To access the resources for this section, cd into the following directory:

```
$ cd ~/modern-devops/ch5/pod/
```

To run a pod with a single NGINX container, execute the following command:

```
$ kubectl run nginx --image=nginx
```

To check whether the pod is running, run the following command:

```
$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
nginx    1/1     Running   0          26s
```

And that's it! As we can see, the pod is now running.

To delete the pod, you can run the following command:

```
$ kubectl delete pod nginx
```

The `kubectl run` command was the imperative way of creating pods, but there's another way of interacting with Kubernetes – by using declarative manifests. **Kubernetes manifests** are YAML or JSON files you can use to declare the desired configuration instead of telling Kubernetes everything through a command line. This method is similar to `docker compose`.

Tip

Always use the declarative method to create Kubernetes resources in staging and production environments. They allow you to store and version your Kubernetes configuration in a source code management tool such as Git and enable GitOps. You can use imperative methods during development because commands have a quicker turnaround than YAML files.

Let's look at an example pod manifest, `nginx-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx
  name: nginx
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "200m"
      requests:
        memory: "100Mi"
        cpu: "100m"
  restartPolicy: Always
```

Let's understand the file first. The file contains the following:

- `apiVersion`: This defines the resource version we are trying to define. In this case, as it is a pod and a **generally available (GA)** resource, the version we will use is `v1`.
- `kind`: This defines the kind of resource we want to create – a pod.
- `metadata`: The `metadata` section defines the name and labels surrounding this resource. It helps in uniquely identifying the resource and grouping multiple resources using labels.
- `spec`: This is the main section where we define the actual specifications for the resource.

- `spec.containers`: This section defines one or more containers that form the pod.
- `spec.containers.name`: This is the container's name, which is `nginx-container` in this case.
- `spec.containers.image`: This is the container image, which is `nginx` in this case.
- `spec.containers.imagePullPolicy`: This can be `Always`, `IfNotPresent`, or `Never`. If set to `Always`, Kubernetes always pulls the image from the registry. If set to `IfNotPresent`, Kubernetes pulls the image only if the image is not found on the node where the pod is scheduled. If set to `Never`, Kubernetes will never attempt to pull images from the registry and will rely completely on local images.
- `spec.containers.resources`: This defines the resource requests and limits.
- `spec.containers.resources.limits`: This defines the resource limits. This is the maximum amount of resources that the pod can allocate, and if the resource consumption increases beyond it, the pod is evicted.
- `spec.containers.resources.limits.memory`: This defines the memory limit.
- `spec.containers.resources.limits.cpu`: This defines the CPU limit.
- `spec.containers.resources.requests`: This defines the resource requests. This is the minimum amount of resources the pod would request during scheduling and will not be scheduled on a node that cannot allocate it.
- `spec.containers.resources.requests.memory`: This defines the amount of memory to be requested.
- `spec.containers.resources.requests.cpu`: This defines the number of CPU cores to be requested.
- `spec.restartPolicy`: This defines the restart policy of containers – `Always`, `OnFailure`, or `Never`. This is similar to the restart policy on Docker.

There are other settings on the pod manifest, but we will explore these as and when we progress.

Important tips

Set `imagePullPolicy` to `IfNotPresent` unless you have a strong reason for using `Always` or `Never`. This will ensure that your containers boot up quickly and you don't download images unnecessarily.

Always use resource requests and limits while scheduling pods. These ensure that your pod is scheduled in an appropriate node and does not exhaust any existing resources. You can also apply a default resource policy at the cluster level to ensure that your developers don't cause any harm if they miss out on this section for some reason.

Let's apply the manifest using the following command:

```
$ kubectl apply -f nginx-pod.yaml
```

The pod that we created is entirely out of bounds from the host. It runs within the container network, and by default, Kubernetes does not allow any pod to be exposed to the host network unless we explicitly want to expose it.

There are two ways to access the pod – using port forwarding with `kubectl port-forward`, or exposing the pod through a Service resource.

Using port forwarding

Before we get into the service side of things, let's consider using the `port-forward` option.

To expose the pod using port forwarding, execute the following command:

```
$ kubectl port-forward nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

The prompt is stuck here. This means it has opened a port forwarding session and is listening on port 8080. It will automatically forward the request it receives on port 8080 to NGINX port 80.

Open a duplicate Terminal session and `curl` on the preceding address to see what we get:

```
$ curl 127.0.0.1:8080
...
<title>Welcome to nginx!</title>
...
```

We can see that it is working as we get the default NGINX response.

Now, there are a few things to remember here.

When we use HTTP `port-forward`, we are forwarding requests from the client machine running `kubectl` to the pod, something similar to what's shown in the following diagram:

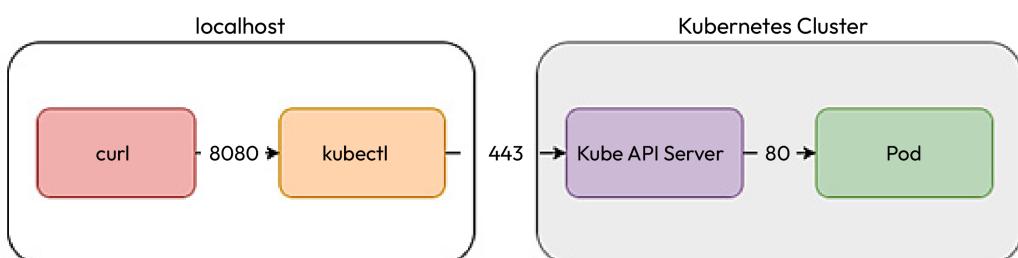


Figure 5.2 – `kubectl port-forward`

When you run `kubectl port-forward`, the `kubectl` client opens a TCP tunnel via the Kube API server, and the Kube API server then forwards the connection to the correct pod. As the connection between the `kubectl` client and the API server is encrypted, it is a very secure way of accessing your pod, but hold your horses before deciding to use `kubectl port-forward` to expose pods to the outside world.

There are particular use cases for using `kubectl port-forward`:

- For troubleshooting any misbehaving pod.
- For accessing an internal Kubernetes service, such as the Kubernetes dashboard – that is, when you don't want to expose the service to the external world but only allow Kubernetes admins and users to log into the dashboard. It is assumed that only these users will have access to the cluster via `kubectl`.

For anything else, you should use `Service` resources to expose your pod, internally or externally. While we will cover the `Service` resource in the next chapter, let's look at a few operations we can perform with a pod.

Troubleshooting pods

Similar to how we can browse logs from a container using `docker logs`, we can browse logs from a container within a Kubernetes pod using the `kubectl logs` command. If more than one container runs within the pod, we can specify the container's name using the `-c` flag.

To access the container logs, run the following command:

```
$ kubectl logs nginx -c nginx
...
127.0.0.1 - - [18/Jun/2023:14:08:01 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"
```

As the pod is running a single container, we need not specify the `-c` flag, so instead, you can use the following command:

```
$ kubectl logs nginx
```

There might be instances where you may want to get a shell to a running container and troubleshoot what's going on within that. We use `docker exec` for that in the Docker world. Similarly, we can use `kubectl exec` for that within Kubernetes.

Run the following command to open a shell session with the container:

```
$ kubectl exec -it nginx -- /bin/bash
root@nginx:/# cd /etc/nginx/ && ls
conf.d fastcgi_params mime.types modules nginx.conf scgi_params uwsgi_params
root@nginx:/etc/nginx# exit
```

You can even run specific commands without opening a shell session. For example, we can perform the preceding operation with a single line, something like the following:

```
$ kubectl exec nginx -- ls /etc/nginx  
conf.d fastcgi_params mime.types modules nginx.conf scgi_params uwsgi_params
```

`kubectl exec` is an important command that helps us troubleshoot containers.

Tip

If you modify files or download packages within the container in `exec` mode, they will persist until the current pod is alive. Once the pod is gone, you will lose all changes. Therefore, it isn't a great way of fixing issues. You should only diagnose problems using `exec`, bake the correct changes in a new image, and then redeploy it.

When we looked at distroless containers in the previous chapter, they did not allow `exec` into the container for security reasons. There are debug images available for distroless that will enable you to open a shell session for troubleshooting purposes if you wish.

Tip

By default, a container runs as the root user if you don't specify the user within the Dockerfile while building the image. You can set a `runAsUser` attribute within your pod's security context if you want to run your pod as a specific user, but this is not ideal. The best practice is to bake the user within the container image.

We've discussed troubleshooting running containers, but what if the containers fail to start for some reason?

Let's look at the following example:

```
$ kubectl run nginx-1 --image=nginx-1
```

Now, let's try to get the pod and see for ourselves:

```
$ kubectl get pod nginx-1  
NAME      READY    STATUS             RESTARTS   AGE  
nginx-1   0/1     ImagePullBackOff   0          25s
```

Oops! There is some error now, and the status is `ImagePullBackOff`. Well, it seems like there is some issue with the image. While we understand that the issue is with the image, we want to understand the real issue, so for further information on this, we can describe the pod using the following command:

```
$ kubectl describe pod nginx-1
```

Now, this gives us a wealth of information regarding the pod, and if you look at the `events` section, you will find a specific line that tells us what is wrong with the pod:

```
Warning Failed 60s (x4 over 2m43s) kubelet Failed to pull image "nginx-1": rpc error: code = Unknown desc = failed to pull and unpack image "docker.io/library/nginx-1:latest": failed to resolve reference "docker.io/library/nginx-1:latest": pull access denied, repository does not exist or may require authorization: server message: insufficient_scope: authorization failed
```

So, this one is telling us that either the repository does not exist, or the repository exists but it is private, and hence authorization failed.

Tip

You can use `kubectl describe` for most Kubernetes resources. It should be the first command you use while troubleshooting issues.

Since we know that the image does not exist, let's change the image to a valid one. We must delete the pod and recreate it with the correct image to do that.

To delete the pod, run the following command:

```
$ kubectl delete pod nginx-1
```

To recreate the pod, run the following command:

```
$ kubectl run nginx-1 --image=nginx
```

Now, let's get the pod; it should run as follows:

```
$ kubectl get pod nginx-1
NAME      READY     STATUS    RESTARTS   AGE
nginx-1   1/1      Running   0          42s
```

The pod is now running since we have fixed the image issue.

So far, we've managed to run containers using pods, but pods are very powerful resources that help you manage containers. Kubernetes pods provide probes to ensure your application's reliability. We'll have a look at this in the next section.

Ensuring pod reliability

We talked about health checks in *Chapter 4, Creating and Managing Container Images*, and I also mentioned that you should not use them on the Docker level and instead use the ones provided by your container orchestrator. Kubernetes provides three **probes** to monitor your pod's health – the **startup probe**, **liveness probe**, and **readiness probe**.

The following diagram depicts all three probes graphically:

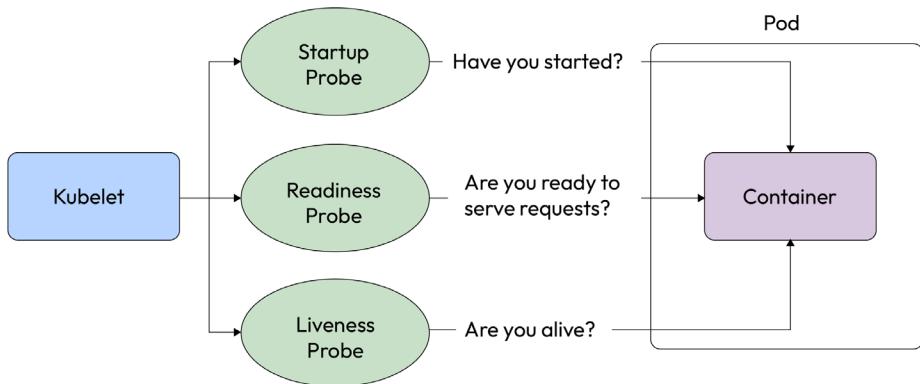


Figure 5.3 – Kubernetes probes

Let's look at each one in turn and understand how and when to use them.

Startup probe

Kubernetes uses **startup probes** to check whether the application has started. You can use startup probes on applications that start slow or those you don't know how long it might take to start. While the startup probe is active, it disables other probes so that they don't interfere with its operation. As the application has not started until the startup probe reports it, there is no point in having any other probes active.

Readiness probe

Readiness probes ascertain whether a container is ready to serve requests. They differ from startup probes because, unlike the startup probe, which only checks whether the application has started, the readiness probe ensures that the container can begin to process requests. A pod is ready when all the containers of the pod are ready. Readiness probes ensure that no traffic is sent to a pod if the pod is not ready. Therefore, it allows for a better user experience.

Liveness probe

Liveness probes are used to check whether a container is running and healthy. The probe checks the health of the containers periodically. If a container is found to be unhealthy, the liveness probe will kill the container. If you've set the `restartPolicy` field of your pod to `Always` or `OnFailure`, Kubernetes will restart the container. Therefore, it improves the service's reliability by detecting deadlocks and ensuring the containers are running instead of just reporting as running.

Now, let's look at an example to understand probes better.

Probes in action

Let's improve the last manifest and add some probes to create the following `nginx-probe.yaml` manifest file:

```
...
  startupProbe:
    exec:
      command:
        - cat
        - /usr/share/nginx/html/index.html
    failureThreshold: 30
    periodSeconds: 10
  readinessProbe:
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 5
    periodSeconds: 5
  livenessProbe:
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 5
    periodSeconds: 3
  restartPolicy: Always
```

The manifest file contains all three probes:

- The startup probe checks whether the `/usr/share/nginx/html/index.html` file exists. It will continue checking it 30 times at an interval of 10 seconds until one of them succeeds. Once it detects the file, the startup probe will stop probing further.
- The readiness probe checks whether there is a listener on port 80 and responds with HTTP `2xx – 3xx` on `path /`. It waits for 5 seconds initially and then checks the pod every 5 seconds. If it gets a `2xx – 3xx` response, it will report the container as ready and accept requests.
- The liveness probe checks whether the pod responds with HTTP `2xx – 3xx` on port 80 and `path /`. It waits for 5 seconds initially and probes the container every 3 seconds. Suppose, during a check, that it finds the pod not responding for `failureThreshold` times (this defaults to 3). In that case, it will kill the container, and the kubelet will take appropriate action based on the pod's `restartPolicy` field.
- Let's apply the YAML file and watch the pods come to life by using the following command:

```
$ kubectl delete pod nginx && kubectl apply -f nginx-probe.yaml && \
  kubectl get pod -w
NAME      READY     STATUS          RESTARTS   AGE
nginx     0/1      Running        0          4s
```

nginx	0/1	Running	0	11s
nginx	1/1	Running	0	12s

As we can see, the pod is quickly ready from the running state. It takes approximately 10 seconds for that to happen as the readiness probe kicks in 10 seconds after the pod starts. Then, the liveness probe keeps monitoring the health of the pod.

Now, let's do something that will break the liveness check. Imagine someone getting a shell to the container and deleting some important files. How do you think the liveness probe will react? Let's have a look.

Let's delete the `/usr/share/nginx/html/index.html` file from the container and then check how the container behaves using the following command:

```
$ kubectl exec -it nginx -- rm -rf /usr/share/nginx/html/index.html && \
  kubectl get pod nginx -w
  NAME      READY     STATUS    RESTARTS   AGE
  nginx     1/1      Running   0          2m5s
  nginx     0/1      Running   1 (2s ago)  2m17s
  nginx     1/1      Running   1 (8s ago)  2m22s
```

So, while we watch the pod, the initial delete is only detected after 9 seconds. That's because of the liveness probe. It tries for 9 seconds, three times `periodSeconds`, since `failureThreshold` defaults to 3, before declaring the pod as unhealthy and killing the container. No sooner does it kill the container than the kubelet restarts it as the pod's `restartPolicy` field is set to `Always`. Then, we see the startup and readiness probes kicking in, and soon, the pod gets ready. Therefore, no matter what, your pods are reliable and will work even if a part of your application is faulty.

Tip

Using readiness and liveness probes will help provide a better user experience, as no requests go to pods that are not ready to process any request. If your application does not respond appropriately, it will replace the container. If multiple pods are running to serve the request, your service is exceptionally resilient.

As we discussed previously, a pod can contain one or more containers. Let's look at some use cases where you might want multiple containers instead of one.

Pod multi-container design patterns

You can run multiple containers in pods in two ways – running a container as an init container or running a container as a helper container to the main container. We'll explore both approaches in the following subsections.

Init containers

Init containers are run before the main container is bootstrapped, so you can use them to initialize your container environment before the main container takes over. Here are some examples:

- A directory might require a particular set of ownership or permissions before you want to start your container using the non-root user
- You might want to clone a Git repository before starting the web server
- You can add a startup delay
- You can generate configuration dynamically, such as for containers that want to dynamically connect to some other pod that it is not aware of during build time but should be during runtime

Tip

Use init containers only as a last resort, as they hamper the startup time of your containers. Try to bake the configuration within your container image or customize it.

Now, let's look at an example to see init containers in action.

To access the resources for this section, cd into the following:

```
$ cd ~/modern-devops/ch5/multi-container-pod/init/
```

Let's serve the `example.com` website from our `nginx` web server. We will get the `example.com` web page and save it as `index.html` in the `nginx` default HTML directory before starting `nginx`.

Access the manifest file, `nginx-init.yaml`, which should contain the following:

```
...
spec:
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: html-volume
  initContainers:
  - name: init-nginx
    image: busybox:1.28
    command: ['sh', '-c', 'mkdir -p /usr/share/nginx/html && wget -O /usr/share/nginx/html/index.html http://example.com']
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: html-volume
  volumes:
  - name: html-volume
    emptyDir: {}
```

If we look at the `spec` section of the manifest file, we'll see the following:

- `containers`: This section defines one or more containers that form the pod.
- `containers.name`: This is the container's name, which is `nginx-container` in this case.
- `containers.image`: This is the container image, which is `nginx` in this case.
- `containers.volumeMounts`: This defines a list of volumes that should be mounted to the container. It is similar to the volumes we read about in *Chapter 4, Creating and Managing Container Images*.
- `containers.volumeMounts.mountPath`: This defines the path to mount the volume on, which is `/usr/share/nginx/html` in this case. We will share this volume with the init container so that when the init container downloads the `index.html` file from `example.com`, this directory will contain the same file.
- `containers.volumeMounts.name`: This is the name of the volume, which is `html-volume` in this case.
- `initContainers`: This section defines one or more init containers that run before the main containers.
- `initContainers.name`: This is the init container's name, which is `init-nginx` in this case.
- `initContainers.image`: This is the init container image, which is `busybox:1.28` in this case.
- `initContainers.command`: This is the command that the busybox should execute. In this case, '`mkdir -p /usr/share/nginx/html && wget -O /usr/share/nginx/html/index.html http://example.com`' will download the content of `example.com` to the `/usr/share/nginx/html` directory.
- `initContainers.volumeMounts`: We will mount the same volume we defined in `nginx-container` on this container. So, anything we save in this volume will automatically appear in `nginx-container`.
- `initContainers.volumeMounts.mountPath`: This defines the path to mount the volume on, which is `/usr/share/nginx/html` in this case.
- `initContainers.volumeMounts.name`: This is the name of the volume, which is `html-volume` in this case.
- `volumes`: This section defines one or more volumes associated with the pod's containers.
- `volumes.name`: This is the volume's name, which is `html-volume` in this case.
- `volumes.emptyDir`: This defines an `emptyDir` volume. It is similar to a `tmpfs` volume in Docker. Therefore, it is not persistent and lasts just for the container's lifetime.

So, let's go ahead and apply the manifest and watch the pod come to life using the following commands:

```
$ kubectl delete pod nginx && kubectl apply -f nginx-init.yaml && \
  kubectl get pod nginx -w
NAME      READY   STATUS        RESTARTS   AGE
nginx    0/1     Init:0/1     0          0s
nginx    0/1     PodInitializing 0          1s
nginx    1/1     Running      0          3s
```

Initially, we can see that the `nginx` pod shows a status of `Init : 0 / 1`. This means that 0 out of 1 init containers have started initializing. After some time, we can see that the pod reports its status, `PodInitializing`, which means that the init containers have started running. The pod reports a running status once the init containers have run successfully.

Now, once the pod starts to run, we can port-forward the container from port 80 to host port 8080 using the following command:

```
$ kubectl port-forward nginx 8080:80
```

Open a duplicate Terminal and try to `curl` the localhost on port 8080 by using the following command:

```
$ curl localhost:8080
<title>Example Domain</title>
```

Here, we can see the example domain response from our web server. This means that the init container worked perfectly fine.

As you may have understood by now, the life cycle of init containers ends before the primary containers start, and a pod can contain one or more main containers. So, let's look at a few design patterns we can use in the main container.

The ambassador pattern

The **ambassador pattern** derives its name from an ambassador, an envoy representing a country overseas. You can also think of an ambassador as a proxy of a particular application. Let's say, for example, that you have migrated one of your existing Python Flask applications to containers, and one of your containers needs to communicate with a Redis database. The database always existed in the local host. Therefore, the database connection details within your application contain `localhost` everywhere.

Now, there are two approaches you can take:

- You can change the application code and use config maps and secrets (more on these later) to inject the database connection details into the environment variable.

- You can keep using the existing code and use a second container as a TCP proxy to the Redis database. The TCP proxy will link with the config map and secrets and contain the Redis database's connection details.

Tip

The ambassador pattern helps developers focus on the application without worrying about the configuration details. Consider using it if you want to decouple application development from config management.

The second approach solves our problem if we wish to do a like-for-like migration. We can use config maps to define the environment-specific configuration without changing the application code. The following diagram shows this approach:

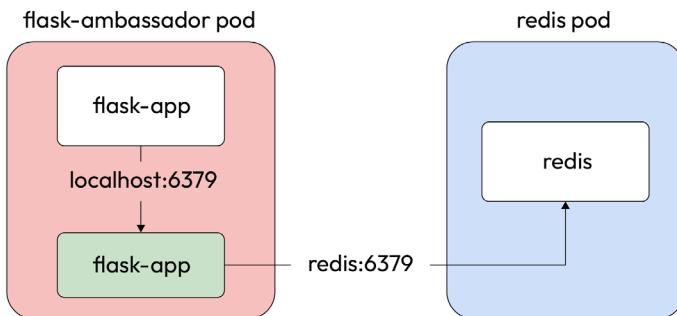


Figure 5.4 – The ambassador pattern

Before we delve into the technicalities, let's understand a config map.

Config map

A **config map** contains key-value pairs that we can use for various purposes, such as defining environment-specific properties or injecting an external variable at container startup or during runtime.

The idea of the config map is to decouple the application with configuration and to externalize configuration at a Kubernetes level. It is similar to using a properties file, for example, to define the environment-specific configuration.

The following diagram explains this beautifully:

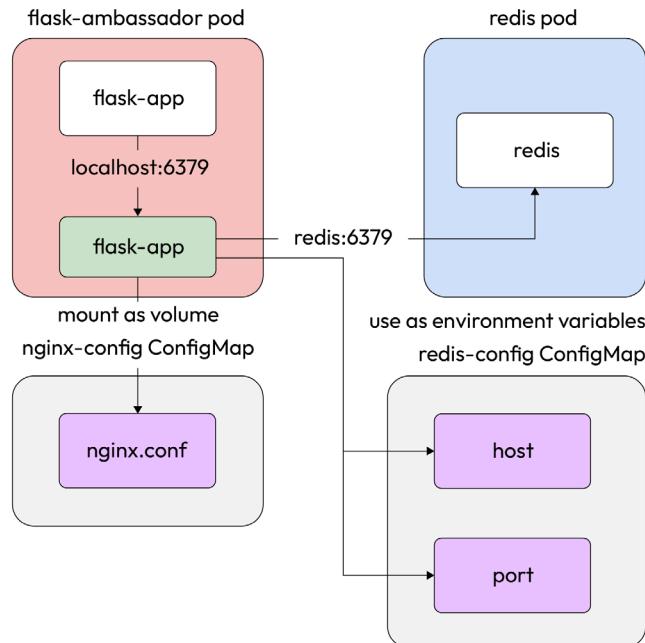


Figure 5.5 – Config maps

We will use `ConfigMap` to define the connection properties of the external Redis database within the ambassador container.

Example application

We will use the example application we used in *Chapter 3, Containerization with Docker*, in the *Deploying a sample application with Docker Compose* section. The source code has been replicated into the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/ambassador
```

You can visualize the `app.py` file of the Flask application, the `requirements.txt` file, and the Dockerfile to understand what the application does.

Now, let's build the container using the following command:

```
$ docker build -t <your_dockerhub_user>/flask-redis .
```

Let's push it to our container registry using the following command:

```
$ docker push <your_dockerhub_user>/flask-redis
```

As you may have noticed, the `app.py` code defines the cache as `localhost:6379`. We will run an ambassador container on `localhost:6379`. The proxy will tunnel the connection to the `redis` pod running elsewhere.

First, let's create the `redis` pod using the following command:

```
$ kubectl run redis --image=redis
```

Now, let's expose the `redis` pod to the cluster resources via a `Service` resource. This will allow any pod within the cluster to communicate with the `redis` pod using the `redis` hostname. We will discuss Kubernetes `Service` resources in the next chapter in detail:

```
$ kubectl expose pod redis --port 6379
```

Cool! Now that the pod and the `Service` resource are up and running, let's work on the ambassador pattern.

We need to define two config maps first. The first describes the `redis` host and port details, while the second defines the template `nginx.conf` file to work as a reverse proxy.

The `redis-config-map.yaml` file looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  host: "redis"
  port: "6379"
```

The preceding YAML file defines a config map called `redis-config` that contains host and port properties. You can have multiple config maps, one for each environment.

The `nginx-config-map.yaml` file looks as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    ...
    stream {
      server {
        listen      6379;
        proxy_pass stream_redis_backend;
      }
      upstream stream_redis_backend {
        server REDIS_HOST:REDIS_PORT;
      }
    }
```

This config map injects the `nginx.conf` template as a config map value. This template defines the configuration of our ambassador pod to listen on `localhost:6379` and tunnel the connection to `REDIS_HOST:REDIS_PORT`. As the `REDIS_HOST` and `REDIS_PORT` values are placeholders, we must fill these up with the correct values that we obtained from the `redis-config` config map. To do that, we can mount this file to a volume and then manipulate it. We can use `initContainer` to initialize the proxy with the correct configuration.

Now, let's look at the pod configuration manifest, `flask-ambassador.yaml`. There are multiple parts of this YAML file. Let's look at the `containers` section first:

```
...
spec:
  containers:
    - name: flask-app
      image: <your_dockerhub_user>/flask-redis
    - name: nginx-ambassador
      image: nginx
      volumeMounts:
        - mountPath: /etc/nginx
          name: nginx-volume
...

```

This section contains a container called `flask-app` that uses the `<your_dockerhub_user>/flask-redis` image that we built in the previous section. The second container is the `nginx-ambassador` container that will act as the proxy to `redis`. Therefore, we have mounted the `/etc/nginx` directory on a volume. This volume is also mounted on the init container to generate the required configuration before `nginx` boots up.

The following is the `initContainers` section:

```
initContainers:
  - name: init-nginx
    image: busybox:1.28
    command: ['sh', '-c', 'cp -L /config/nginx.conf /etc/nginx/nginx.conf && sed -i "s/REDIS_HOST/${REDIS_HOST}/g" /etc/nginx/nginx.conf']
    env:
      - name: REDIS_HOST
        valueFrom:
          configMapKeyRef:
            name: redis-config
            key: host
      - name: REDIS_PORT
        valueFrom:
          configMapKeyRef:
            name: redis-config
            key: port
    volumeMounts:
      - mountPath: /etc/nginx
        name: nginx-volume
      - mountPath: /config
        name: config

```

This section defines a busybox container – `init-nginx`. The container needs to generate the `nginx-ambassador` proxy configuration to communicate with Redis; therefore, two environment variables are present. Both environment variables are sourced from the `redis-config` config map. Apart from that, we have also mounted the `nginx.conf` file from the `nginx-config` config map. The command section within the init container uses the environment variables to replace placeholders within the `nginx.conf` file, after which we get a TCP proxy to the Redis backend.

The `volumes` section defines `nginx-volume` as an `emptyDir` volume, and the `config` volume is mounted from the `nginx.conf` file present in the `nginx-config` config map:

```
volumes:
- name: nginx-volume
  emptyDir: {}
- name: config
  configMap:
    name: nginx-config
    items:
      - key: "nginx.conf"
        path: "nginx.conf"
```

Now, let's start applying the YAML files in steps.

Apply both of the config maps using the following commands:

```
$ kubectl apply -f redis-config-map.yaml
$ kubectl apply -f nginx-config-map.yaml
```

Let's apply the pod configuration using the following command:

```
$ kubectl apply -f flask-ambassador.yaml
```

Get the pod to see whether the configuration is correct by using the following command:

```
$ kubectl get pod/flask-ambassador
NAME          READY   STATUS    RESTARTS   AGE
flask-ambassador   2/2     Running   0          10s
```

As the pod is running successfully now, let's port-forward 5000 to the localhost for some tests by using the following command:

```
$ kubectl port-forward flask-ambassador 5000:5000
```

Now, open a duplicate Terminal and try to `curl` on `localhost:5000` using the following command:

```
$ curl localhost:5000
Hi there! This page was last visited on 2023-06-18, 16:52:28.
$ curl localhost:5000
Hi there! This page was last visited on 2023-06-18, 16:52:28.
$ curl localhost:5000
Hi there! This page was last visited on 2023-16-28, 16:52:32.
```

As we can see, every time we `curl` the application, we get the last visited time on our screen. The ambassador pattern is working.

This was a simple example of the ambassador pattern. There are advanced configurations you can do to add fine-grained control on how your application should interact with the outside world. You can use the ambassador pattern to secure traffic that moves from your containers. It also simplifies application development for your development team as they need not worry about these nuances. In contrast, the operations team can use these containers to manage your environment in a better way without stepping on each other's toes.

Tip

As the ambassador pattern adds some overhead as you tunnel connections via a proxy, you should only use it if the management benefits outweigh the extra cost you incur because of the ambassador container.

Now, let's look at another multi-container pod pattern – sidecars.

The sidecar pattern

Sidecars derive their names from motorcycle sidecars. The sidecar does not change the bike's core functionality and can work perfectly without it. Instead, it adds an extra seat, a functionality that helps you give an additional person a ride. Similarly, sidecars in a pod are helper containers that provide functionalities unrelated to the main container's core functionality and enhance it instead. Examples include logging and monitoring containers. Keeping a separate container for logging will help decouple the logging responsibilities from your main container, which will help you monitor your application even when the main container goes down for some reason.

It also helps if there is some issue with the logging code, and instead of the entire application going down, only the logging container is impacted. You can also use sidecars to keep helper or related containers together with the main container since we know containers within the pod share the same machine.

Tip

Only use multi-container pods if two containers are functionally related and work as a unit.

You can also use sidecars to segregate your application with secrets. For example, if you are running a web application that needs access to specific passwords to operate, it would be best to mount the secrets to a sidecar and let the sidecar provide the passwords to the web application via a link. This is because if someone gains access to your application container's filesystem, they cannot get hold of your passwords as another container is responsible for sourcing it, as shown in the following diagram:

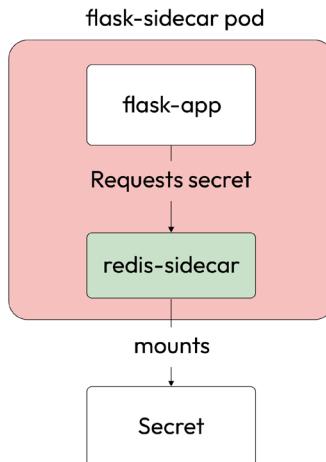


Figure 5.6 – The sidecar pattern

Let's implement the preceding pattern to understand a sidecar better. We have a Flask application that interacts with a Redis sidecar. We will pre-populate the Redis sidecar with a secret `foobar`, and we will do that by using the Kubernetes secret resource.

Secrets

Secrets are very similar to config maps, with the difference that the secret values are `base64`-encoded instead of `plaintext`. While `base64` encoding does not make any difference, and it is as bad as `plaintext` from a security standpoint, you should use secrets for sensitive information such as passwords. That is because the Kubernetes community will develop a solution to tighten the security around secrets in future releases. If you use secrets, you will directly benefit from it.

Tip

As a rule of thumb, always use secrets for confidential data, such as API keys and passwords, and config maps for non-sensitive configuration data.

To access the files for this section, go to the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar
```

Now, let's move on to the example Flask application.

Example application

The Flask application queries a Redis sidecar for the secret and sends that as a response. That is not ideal, as you won't send secrets back as a response, but for this demo, let's go ahead with that.

So, first, let's design our sidecar so that it pre-populates data within the container after it starts.

We need to create a secret named `secret` with a value of `foobar`. Now, base64-encode the Redis command to set the secret into the cache by running the following command:

```
$ echo 'SET secret foobar' | base64  
U0VUIHNlY3JldCBmb29iYXIK
```

Now that we have the base64-encoded secret, we can create a `redis-secret.yaml` manifest with the string as follows:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: redis-secret  
data:  
  redis-secret: U0VUIHNlY3JldCBmb29iYXIK
```

Then, we need to build the Redis container so that this secret is created at startup. To access the files for this section, go to the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar/redis/
```

Create an `entrypoint.sh` file, as follows:

```
redis-server --daemonize yes && sleep 5  
redis-cli < /redis-master/init.redis  
redis-cli save  
redis-cli shutdown  
redis-server
```

The shell script looks for a file, `init.redis`, within the `/redis-master` directory and runs the `redis-cli` command on it. This means the cache will be pre-populated with the values defined in our secret, provided we mount the secret as `/redis-master/init.redis`.

Then, we must create a Dockerfile that will use this `entrypoint.sh` script, as follows:

```
FROM redis  
COPY entrypoint.sh /tmp/  
CMD ["sh", "/tmp/entrypoint.sh"]
```

Now that we are ready, we can build and push the code to Docker Hub:

```
$ docker build -t <your_dockerhub_user>/redis-secret .  
$ docker push <your_dockerhub_user>/redis-secret
```

Now that we are ready with the Redis image, we must build the Flask application image. To access the files for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch5/multi-container-pod/sidecar/flask
```

Let's look at the `app.py` file first:

```
...
cache = redis.Redis(host='localhost', port=6379)
def get_secret():
    try:
        secret = cache.get('secret')
        return secret
    ...
def index():
    secret = str(get_secret()).decode('utf-8'))
    return 'Hi there! The secret is {}'.format(secret)
```

The code is simple – it gets the secret from the cache and returns that in the response.

We also created the same Dockerfile that we did in the previous section.

So, let's build and push the container image to Docker Hub:

```
$ docker build -t <your_dockerhub_user>/flask-redis-secret .
$ docker push <your_dockerhub_user>/flask-redis-secret
```

Now that our images are ready, let's look at the pod manifest, `flask-sidecar.yaml`, which is present in the `~/modern-devops/ch5/multi-container-pod/sidecar/` directory:

```
...
spec:
  containers:
    - name: flask-app
      image: <your_dockerhub_user>/flask-redis-secret
    - name: redis-sidecar
      image: <your_dockerhub_user>/redis-secret
      volumeMounts:
        - mountPath: /redis-master
          name: secret
  volumes:
    - name: secret
      secret:
        secretName: redis-secret
        items:
          - key: redis-secret
            path: init.redis
```

The pod defines two containers – `flask-app` and `redis-sidecar`. The `flask-app` container runs the Flask application that will interact with `redis-sidecar` for the secret. The `redis-sidecar` container has mounted the `secret` volume on `/redis-master`. The pod definition also contains a single volume called `secret`, and the volume points to the `redis-secret` secret and mounts that as a file, `init.redis`.

So, in the end, we have a file, `/redis-master/init.redis`, and, as we know, the `entrypoint.sh` script looks for this file and runs the `redis-cli` command to pre-populate the Redis cache with the secret data.

Let's apply the secret first using the following command:

```
$ kubectl apply -f redis-secret.yaml
```

Then, we can apply the `flask-sidecar.yaml` file using the following command:

```
$ kubectl apply -f flask-sidecar.yaml
```

Now, let's get the pods using the following command:

```
$ kubectl get pod flask-sidecar
NAME        READY   STATUS    RESTARTS   AGE
flask-sidecar  2/2     Running   0          11s
```

As the pod is running, it's time to port-forward it to the host using the following command:

```
$ kubectl port-forward flask-sidecar 5000:5000
```

Now, let's open a duplicate Terminal, run the `curl localhost:5000` command, and see what we get:

```
$ curl localhost:5000
Hi there! The secret is foobar.
```

As we can see, we get the secret, `foobar`, in the response. The sidecar is working correctly!

Now, let's look at another popular multi-container pod pattern – the adapter pattern.

The adapter pattern

As its name suggests, the **adapter pattern** helps change something to fit a standard, such as cell phones and laptop adapters, which convert our main power supply into something our devices can digest. A great example of the adapter pattern is transforming log files so that they fit an enterprise standard and feed your log analytics solution:

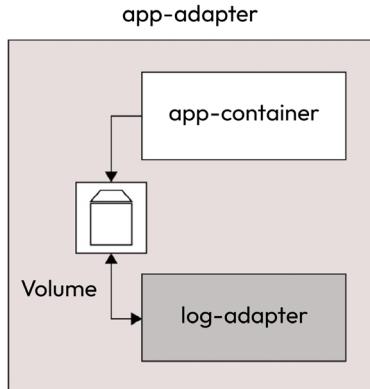


Figure 5.7 – The adapter pattern

It helps when you have a heterogeneous solution outputting log files in several formats but a single log analytics solution that only accepts messages in a particular format. There are two ways of doing this: changing the code for outputting log files in a standard format or using an adapter container to execute the transformation.

Let's look at the following scenario to understand it further.

We have an application that continuously outputs log files without a date at the beginning. Our adapter should read the stream of logs and append the timestamp as soon as a logline is generated.

For this, we will use the following pod manifest, `app-adapter.yaml`:

```
...
spec:
  volumes:
    - name: logs
      emptyDir: {}
  containers:
    - name: app-container
      image: ubuntu
      command: ["/bin/bash"]
      args: ["-c", "while true; do echo 'This is a log line' >> /var/log/app.log; sleep 2;done"]
      volumeMounts:
        - name: logs
          mountPath: /var/log
    - name: log-adapter
      image: ubuntu
      command: ["/bin/bash"]
      args: ["-c", "apt update -y && apt install -y moreutils && tail -f /var/log/app.log | ts '[%Y-%m-%d %H:%M:%S]' > /var/log/out.log"]
      volumeMounts:
        - name: logs
          mountPath: /var/log
```

The pod contains two containers – the app container, which is a simple Ubuntu container that outputs `This is a log line` every 2 seconds, and the log adapter, which continuously tails the `app.log` file, adds a timestamp at the beginning of the line, and sends the resulting output to `/var/log/out.log`. Both containers share the `/var/log` volume, which is mounted as an `emptyDir` volume on both containers.

Now, let's apply this manifest using the following command:

```
$ kubectl apply -f app-adapter.yaml
```

Let's wait a while and check whether the pod is running by using the following command:

```
$ kubectl get pod app-adapter
NAME        READY   STATUS    RESTARTS   AGE
app-adapter  2/2     Running   0          8s
```

As the pod is running, we can now get a shell into the log adapter container by using the following command:

```
$ kubectl exec -it app-adapter -c log-adapter -- bash
```

When we get into the shell, we can `cd` into the `/var/log` directory and list its contents using the following command:

```
root@app-adapter:/# cd /var/log/ && ls
app.log apt/ dpkg.log out.log
```

As we can see, we get `app.log` and `out.log` as two files. Now, let's use the `cat` command to print both of them to see what we get.

First, `cat` the `app.log` file using the following command:

```
root@app-adapter:/var/log# cat app.log
This is a log line
This is a log line
This is a log line
```

Here, we can see that a series of log lines are being printed.

Now, `cat` the `out.log` file to see what we get using the following command:

```
root@app-adapter:/var/log# cat out.log
[2023-06-18 16:35:25] This is a log line
[2023-06-18 16:35:27] This is a log line
[2023-06-18 16:35:29] This is a log line
```

Here, we can see timestamps in front of the log line. This means that the adapter pattern is working correctly. You can then export this log file to your log analytics tool.

Summary

We have reached the end of this critical chapter. We've covered enough ground to get you started with Kubernetes and understand and appreciate the best practices surrounding it.

We started with Kubernetes and why we need it and then discussed bootstrapping a Kubernetes cluster using Minikube and KinD. Then, we looked at the pod resource and discussed creating and managing pods, troubleshooting them, ensuring your application's reliability using probes, and multi-container design patterns to appreciate why Kubernetes uses pods in the first place instead of containers.

In the next chapter, we will deep dive into the advanced aspects of Kubernetes by covering controllers, services, ingresses, managing a stateful application, and Kubernetes command-line best practices.

Questions

Answer the following questions to test your knowledge of this chapter:

1. All communication with Kubernetes happens via which of the following?
 - A. Kubelet
 - B. API server
 - C. Etcd
 - D. Controller manager
 - E. Scheduler
2. Which of the following is responsible for ensuring that the cluster is in the desired state?
 - A. Kubelet
 - B. API server
 - C. Etcd
 - D. Controller manager
 - E. Scheduler
3. Which of the following is responsible for storing the desired state of the cluster?
 - A. Kubelet
 - B. API server
 - C. Etcd
 - D. Controller manager
 - E. Scheduler
4. A pod can contain more than one container. (True/False)

5. You can use port-forwarding for which of the following use cases? (Choose two)
 - A. For troubleshooting a misbehaving pod
 - B. For exposing a service to the internet
 - C. For accessing a system service such as the Kubernetes dashboard
6. Using a combination of which two probes can help you ensure that your application is reliable even when your application has some intermittent issues? (Choose two.)
 - A. Startup probe
 - B. Liveness probe
 - C. Readiness probe
7. We may use KinD in production. (True/False)
8. Which of the following multi-container patterns is used as a forward proxy?
 - A. Ambassador
 - B. Adapter
 - C. Sidecar
 - D. Init containers

Answers

Here are the answers to this chapter's questions:

1. B
2. D
3. C
4. True
5. A, C
6. B, C
7. False
8. A

6

Managing Advanced Kubernetes Resources

In the previous chapter, we covered Kubernetes and why we need it and then discussed bootstrapping a Kubernetes cluster using MiniKube and KinD. We then looked at the Pod resource and discussed how to create and manage pods, how to troubleshoot them, and how to ensure your application's reliability using probes, along with multi-container design patterns to appreciate why Kubernetes uses pods in the first place instead of containers. We also looked at Secrets and ConfigMaps.

Now, we will dive deep into the advanced aspects of Kubernetes and Kubernetes command-line best practices.

In this chapter, we're going to cover the following main topics:

- The need for advanced Kubernetes resources
- Kubernetes Deployments
- Kubernetes Services and Ingresses
- Horizontal pod autoscaling
- Managing stateful applications
- Kubernetes command-line best practices, tips, and tricks

So, let's dive in!

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine (GKE)**, for the exercises. That is because you will not be able to spin up load balancers and PersistentVolumes within your local system, and therefore, we cannot use KinD and MiniKube in this chapter.

Currently, **Google Cloud Platform (GCP)** provides a free \$300 trial for 90 days, so you can go ahead and sign up for one at <https://cloud.google.com/free>.

Spinning up GKE

Once you've signed up and logged in to your console, you can open the Google Cloud Shell CLI to run the commands.

You need to enable the GKE API first using the following command:

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

To create a three-node GKE cluster, run the following command:

```
$ gcloud container clusters create cluster-1 --zone us-central1-a
```

And that's it! The cluster is up and running.

You will also need to clone the following GitHub repository for some exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

Run the following command to clone the repository into your home directory and cd into the ch6 directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch6
```

Now, let's understand why we need advanced Kubernetes resources.

The need for advanced Kubernetes resources

In the last chapter, we looked at pods, the basic building blocks of Kubernetes that provide everything for your containers to run within a Kubernetes environment. However, pods on their own are not that effective. The reason is that while they define a container application and its specification, they do not replicate, auto-heal, or maintain a particular state. When you delete a pod, the pod is gone. You cannot maintain multiple versions of your code or roll out and roll back releases using a pod. You also cannot autoscale your application with traffic with pods alone. Pods do not allow you to expose your containers to the outside world, and they do not provide traffic management capabilities such as load balancing, content and path-based routing, storing persistent data to externally attached storage,

and so on. To solve these problems, Kubernetes provides us with specific advanced resources, such as Deployments, Services, Ingresses, PersistentVolumes and claims, and StatefulSets. Let's start with Kubernetes Deployments in the next section.

Kubernetes Deployments

Let's understand Kubernetes Deployments using a simple analogy.

Imagine you're a chef preparing a particular dish in your kitchen. You want to make sure that it is consistently perfect every time you serve it, and you want to be able to change the recipe without causing a chaotic mess.

In the world of Kubernetes, a “Deployment” is like your sous chef. It helps you create and manage copies of your pods effortlessly.

Here's how it works:

- **Creating consistency:** You want to serve your dish to many guests. Therefore, instead of cooking each plate separately, you prepare a bunch of them at once. All of them should taste the same and strictly as intended. A Kubernetes Deployment does the same for your pod. It creates multiple identical copies of your pod, ensuring they all have the same setup.
- **Updating safely:** Now, imagine you have a new twist for your dish. You want to try it out, but you want your guests only to eat something if it turns out right. Similarly, when you want to update your app in Kubernetes, the Deployment resource slowly and carefully replaces old copies with new ones individually, so your app is always available, and your guests (or users) don't notice any hiccups.
- **Rolling back gracefully:** Sometimes, experiments don't go as planned, and you must revert to the original recipe. Just as in your kitchen, Kubernetes lets you roll back to the previous version of your pod if things don't work out with the new one.
- **Scaling easily:** Imagine your restaurant suddenly gets a rush of customers, and you need more plates for your special dish. A Kubernetes Deployment helps with that, too. It can quickly create more copies of your pod to handle the increased demand and remove them when things quieten down.
- **Managing multiple kitchens:** If you have multiple restaurants, you'd want your signature dish to taste the same in all of them. Similarly, if you're using Kubernetes across different environments such as testing, development, and production, Deployments help keep things consistent.

In essence, Kubernetes Deployments help manage your pod, like a sous chef manages the dishes served from a kitchen. They ensure consistency, safety, and flexibility, ensuring your application runs smoothly and can be updated without causing a mess in your *software kitchen*.

Container application Deployments within Kubernetes are done through Deployment resources. Deployment resources employ ReplicaSet resources behind the scenes, so it would be good to look at ReplicaSet resources before we move on to understand Deployment resources.

ReplicaSet resources

ReplicaSet resources are Kubernetes controllers that help you run multiple pod replicas at a given time. They provide horizontal scaling for your container workloads, forming the basic building block of a horizontal scale set for your containers, a group of similar containers tied together to run as a unit.

ReplicaSet resources define the number of pod replicas to run at a given time. The Kubernetes controller then tries to maintain the replicas and recreates a pod if it goes down.

You should never use ReplicaSet resources on their own, but instead, they should act as a backend to a Deployment resource.

For understanding, however, let's look at an example. To access the resources for this section, cd into the following:

```
$ cd ~/modern-devops/ch6/deployments/
```

The ReplicaSet resource manifest, nginx-replica-set.yaml, looks like this:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
```

The resource manifest includes apiVersion and kind, as with any other resource. It also contains a metadata section that defines the resource's name and labels attributes, similar to any other Kubernetes resource.

The `spec` section contains the following attributes:

- `replicas`: This defines the number of pod replicas matched by the selector to run at a given time.
- `selector`: This defines the basis on which the `ReplicaSet` resource will include pods.
- `selector.matchLabels`: This defines labels and their values to select pods. Therefore, the `ReplicaSet` resource will select any pod with the `app: nginx` label.
- `template`: This is an optional section that you can use to define the pod template. This section's contents are very similar to defining a pod, except it lacks the `name` attribute, as the `ReplicaSet` resource will generate dynamic names for pods. If you don't include this section, the `ReplicaSet` resource will still try to acquire existing pods with matching labels. However, it cannot create new pods because of the missing template. Therefore, it is best practice to specify a template for a `ReplicaSet` resource.

Let's go ahead and apply this manifest to see what we get:

```
$ kubectl apply -f nginx-replica-set.yaml
```

Now, let's run the following command to list the `ReplicaSet` resources:

```
$ kubectl get replicaset
NAME      DESIRED      CURRENT      READY      AGE
nginx     3            3            0          9s
```

Right—so, we see that there are three desired replicas. Currently, 3 replicas are running, but 0 are ready. Let's wait for a while and then rerun the following command:

```
$ kubectl get replicaset
NAME      DESIRED      CURRENT      READY      AGE
nginx     3            3            3          1m10s
```

And now, we see 3 ready pods that are awaiting a connection. Now, let's list the pods and see what the `ReplicaSet` resource has done behind the scenes using the following command:

```
$ kubectl get pod
NAME        READY      STATUS      RESTARTS      AGE
nginx-6qr9j  1/1      Running    0           1m32s
nginx-7hkqv  1/1      Running    0           1m32s
nginx-9kvkj  1/1      Running    0           1m32s
```

There are three `nginx` pods, each with a name that starts with `nginx` but ends with a random hash. The `ReplicaSet` resource has appended a random hash to generate unique pods at the end of the `ReplicaSet` resource name. Yes—the name of every resource of a particular kind in Kubernetes should be unique.

Let's go ahead and use the following command to delete one of the pods from the ReplicaSet resource and see what we get:

```
$ kubectl delete pod nginx-9kvkj && kubectl get pod
pod "nginx-9kvkj" deleted
NAME      READY   STATUS    RESTARTS   AGE
nginx-6qr9j 1/1     Running   0          8m34s
nginx-7hkqv 1/1     Running   0          8m34s
nginx-9xbdf 1/1     Running   0          5s
```

We see that even though we deleted the `nginx-9kvkj` pod, the ReplicaSet resource has replaced it with a new pod, `nginx-9xbdf`. That is how ReplicaSet resources work.

You can delete a ReplicaSet resource just like any other Kubernetes resource. You can run the `kubectl delete replicaset <ReplicaSet name>` command for an imperative approach or use `kubectl delete -f <manifest_file>` for a declarative approach.

Let's use the former approach and delete the ReplicaSet resource by using the following command:

```
$ kubectl delete replicaset nginx
```

Let's check whether the ReplicaSet resource has been deleted by running the following command:

```
$ kubectl get replicaset
No resources found in default namespace.
```

We don't have anything in the `default` namespace. This means that the ReplicaSet resource is deleted.

As we discussed, ReplicaSet resources should not be used on their own but should instead be the backend of Deployment resources. Let's now look at Kubernetes Deployment resources.

Deployment resources

Kubernetes Deployment resources help to manage deployments for container applications. They are typically used for managing stateless workloads. You can still use them to manage stateful applications, but the recommended approach for stateful applications is to use StatefulSet resources.

Kubernetes Deployments use ReplicaSet resources as a backend, and the chain of resources looks like what's shown in the following diagram:



Figure 6.1 – Deployment chain

Let's take the preceding example and create an nginx Deployment resource manifest—nginx-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
```

The manifest is very similar to the ReplicaSet resource, except for the kind attribute—Deployment, in this case.

Let's apply the manifest by using the following command:

```
$ kubectl apply -f nginx-deployment.yaml
```

So, as the Deployment resource has been created, let's look at the chain of resources it created. Let's run `kubectl get` to list the Deployment resources using the following command:

```
$ kubectl get deployment
NAME      READY     UP-TO-DATE   AVAILABLE   AGE
nginx    3/3       3           3           6s
```

And we see there is one Deployment resource called nginx, with 3/3 ready pods and 3 up-to-date pods. As Deployment resources manage multiple versions, UP-TO-DATE signifies whether the latest Deployment resource has rolled out successfully. We will look into the details of this in the subsequent sections. It also shows 3 available pods at that time.

As we know Deployment resources create ReplicaSet resources in the background, let's get the ReplicaSet resources using the following command:

```
$ kubectl get replicaset
NAME             DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8   3         3         3      11s
```

And we see that the Deployment resource has created a ReplicaSet resource, which starts with nginx and ends with a random hash. That is required as a Deployment resource might contain one or more ReplicaSet resources. We will look at how in the subsequent sections.

Next in the chain are pods, so let's get the pods using the following command to see for ourselves:

```
$ kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
nginx-6799fc88d8-d52mj  1/1     Running   0          15s
nginx-6799fc88d8-dmpbn  1/1     Running   0          15s
nginx-6799fc88d8-msvxkw 1/1     Running   0          15s
```

And, as expected, we have three pods. Each begins with the ReplicaSet resource name and ends with a random hash. That's why you see two hashes in the pod name.

Let's assume you have a new release and want to deploy a new version of your container image. So, let's update the Deployment resource with a new image using the following command:

```
$ kubectl set image deployment/nginx nginx=nginx:1.16.1
deployment.apps/nginx image updated
```

To check the deployment status, run the following command:

```
$ kubectl rollout status deployment nginx
deployment "nginx" successfully rolled out
```

Imperative commands such as those just shown are normally not used in production environments because they lack the audit trail you would get with declarative manifests using Git to version them. However, if you do choose to use imperative commands, you can always record the change cause of the previous rollout using the following command:

```
$ kubectl annotate deployment nginx kubernetes.io/change-cause \
= "Updated nginx version to 1.16.1" --overwrite=true
deployment.apps/nginx annotated
```

To check the Deployment history, run the following command:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1          <none>
2          Updated nginx version to 1.16.1
```

As we can see, there are two revisions in the Deployment history. Revision 1 was the initial Deployment, and revision 2 was because of the kubectl set image command we ran, as evident from the CHANGE-CAUSE column.

Let's say you find an issue after Deployment and want to roll it back to the previous version. To do so and also recheck the status of the Deployment, run the following command:

```
$ kubectl rollout undo deployment nginx && kubectl rollout status deployment nginx
deployment.apps/nginx rolled back
Waiting for deployment "nginx" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "nginx" rollout to finish: 1 old replicas are pending
termination...
deployment "nginx" successfully rolled out
```

And finally, let's recheck the deployment history using the following command:

```
$ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
2          Updated nginx version to 1.16.1
3          <none>
```

And we get revision 3 with a CHANGE-CAUSE value of <none>. In this case, we did not annotate the rollback as in the last command.

Tip

Always annotate Deployment updates as it becomes easier to peek into the history to see what got deployed.

Now, let's look at some common Kubernetes Deployment strategies to understand how to use Deployments effectively.

Kubernetes Deployment strategies

Updating an existing Deployment requires you to specify a new container image. That is why we version container images in the first place so that you can roll out, and roll back, application changes as required. As we run everything in containers—and containers, by definition, are ephemeral—this enables a host of different deployment strategies that we can implement. There are several deployment strategies, and some of these are set out as follows:

- **Recreate:** This is the simplest of all. Delete the old pod and deploy a new one.
- **Rolling update:** Slowly roll out the pods of the new version while still running the old version, and slowly remove the old pods as the new pods get ready.
- **Blue/green:** This is a derived deployment strategy where we keep both versions running simultaneously and switch the traffic to the newer version when we want.

- **Canary:** This applies to Blue/Green Deployments where we switch a percentage of traffic to the newer version of the application before fully rolling out the release.
- **A/B testing:** A/B testing is more of a technique to apply to Blue/Green Deployments. This is when you want to roll out the newer version to a subset of willing users and study the usage patterns before completely rolling out the newer version. You do not get A/B testing out of the box with Kubernetes but instead should rely on service mesh technologies that plug in well with Kubernetes, such as **Istio**, **Linkerd**, and **Traefik**.

Kubernetes provides two deployment strategies out of the box—`Recreate` and `RollingUpdate`.

Recreate

The `Recreate` strategy is the most straightforward deployment strategy. When you update the `Deployment` resource with the `Recreate` strategy, Kubernetes immediately spins down the old `ReplicaSet` resource and creates a new one with the required number of replicas along the lines of the following diagram:

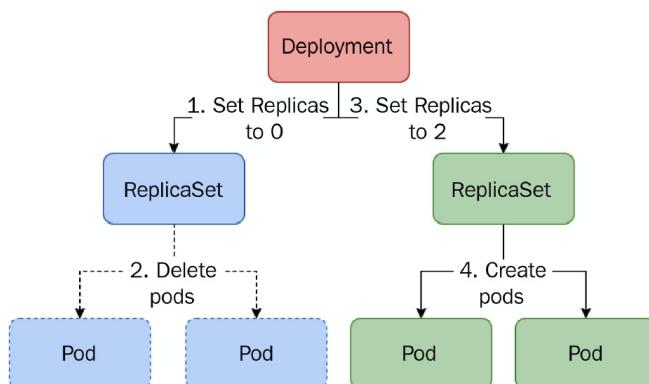


Figure 6.2 – Recreate strategy

Kubernetes does not delete the old `ReplicaSet` resource but instead sets `replicas` to 0. That is required to roll back to the old version quickly. This approach results in downtime and is something you want to use only in case of a constraint. Thus, this strategy isn't the default deployment strategy in Kubernetes.

Tip

You can use the `Recreate` strategy if your application does not support multiple replicas, if it does not support more than a certain number of replicas (such as applications that need to maintain a quorum), or if it does not support multiple versions at once.

Let's update `nginx-deployment` with the `Recreate` strategy. Let's look at the `nginx-recreate.yaml` file:

```
...
spec:
  replicas: 3
  strategy:
    type: Recreate
...

```

The YAML file now contains a `strategy` section with a `Recreate` type. Now, let's apply the `nginx-recreate.yaml` file and watch the `ReplicaSet` resources using the following command:

```
$ kubectl apply -f nginx-recreate.yaml && kubectl get replicaset -w
deployment.apps/nginx configured
NAME          DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8  0         0         0       0s
nginx-6889dfcccd5 0         3         3       7m42s
nginx-6889dfcccd5  0         0         0       7m42s
nginx-6799fc88d8  3         0         0       1s
nginx-6799fc88d8  3         3         0       2s
nginx-6799fc88d8  3         3         3       6s
```

The Deployment resource creates a new `ReplicaSet` resource—`nginx-6799fc88d8`—with 0 desired replicas. It then sets 0 desired replicas to the old `ReplicaSet` resource and waits for the old `ReplicaSet` resource to be completely evicted. It then starts automatically rolling out the new `ReplicaSet` resource to the desired images.

RollingUpdate

When you update the Deployment with a `RollingUpdate` strategy, Kubernetes creates a new `ReplicaSet` resource, and it simultaneously spins up the required number of pods on the new `ReplicaSet` resource while slowly spinning down the old `ReplicaSet` resource, as evident from the following diagram:

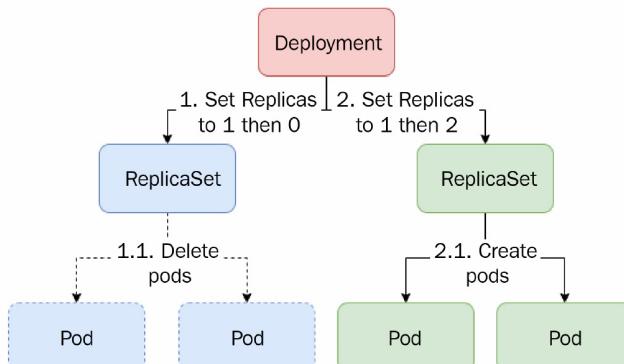


Figure 6.3 – RollingUpdate strategy

`RollingUpdate` is the default deployment strategy. You can use the `RollingUpdate` strategy in most applications, apart from ones that cannot tolerate more than one version of the application at a given time.

Let's update the `nginx` Deployment resource using the `RollingUpdate` strategy. We will reuse the standard `nginx-deployment.yaml` file that we used before. Use the following command and see what happens to the ReplicaSet resources:

```
$ kubectl apply -f nginx-deployment.yaml && kubectl get replicaset -w
deployment.apps/nginx configured
NAME          DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8  3         3         3      49s
nginx-6889dfcccd5 1         1         1      4s
nginx-6799fc88d8  2         2         2      53s
nginx-6889dfcccd5 2         2         2      8s
nginx-6799fc88d8  1         1         1      57s
nginx-6889dfcccd5 3         3         3      11s
nginx-6799fc88d8  0         0         0      60s
```

As we see, the old ReplicaSet resource—`nginx-6799fc88d8`—is being rolled down, and the new ReplicaSet resource—`nginx-6889dfcccd5`—is being rolled out simultaneously.

The `RollingUpdate` strategy also has two options—`maxUnavailable` and `maxSurge`.

While `maxSurge` defines the maximum number of additional pods we can have at a given time, `maxUnavailable` defines the maximum number of unavailable pods we can have at a given time.

Tip

Set `maxSurge` to 0 if your application cannot tolerate more than a certain number of replicas. Set `maxUnavailable` to 0 if you want to maintain reliability and your application can tolerate more than the set replicas. You cannot specify 0 for both parameters, as that will make any rollout attempts impossible. While setting `maxSurge`, ensure your cluster has spare capacity to spin up additional pods, or the rollout will fail.

Using these settings, we can create different kinds of custom rollout strategies—some popular ones are discussed in the following sections.

Ramped slow rollout

If you have numerous replicas but want to roll out the release slowly, observe the application for any issues, and roll back your deployment if needed, you should use this strategy.

Let's create an `nginx` deployment, `nginx-ramped-slow-rollout.yaml`, using the **ramped slow rollout** strategy:

```
...
spec:
```

```
replicas: 10
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 0
...
```

The manifest is very similar to the generic Deployment, except that it contains 10 replicas and a `strategy` section.

The `strategy` section contains the following:

- `type: RollingUpdate`
- `rollingUpdate`: The section describing rolling update attributes `-maxSurge` and `maxUnavailable`

Now, let's apply the YAML file and wait for the deployment to completely roll out to 10 replicas using the following commands:

```
$ kubectl apply -f nginx-ramped-slow-rollout.yaml \
&& kubectl rollout status deployment nginx
deployment.apps/nginx configured
...
deployment "nginx" successfully rolled out
```

As we see, the pods have rolled out completely. Let's now update the Deployment resource with a different `nginx` image version and see what we get using the following command:

```
$ kubectl set image deployment nginx nginx=nginx:1.16.1 \
&& kubectl get replicaset -w
deployment.apps/nginx image updated
NAME      DESIRED   CURRENT   READY   AGE
nginx-6799fc88d8  10        10        10     3m51s
nginx-6889dfcccd5  1         1         0      0s
nginx-6799fc88d8  9        10        10     4m
. . . . .
nginx-6889dfcccd5  8         8         8      47s
nginx-6799fc88d8  2         3         3      4m38s
nginx-6889dfcccd5  9         9         8      47s
nginx-6799fc88d8  2         2         2      4m38s
nginx-6889dfcccd5  9         9         9      51s
nginx-6889dfcccd5  10        9         9      51s
nginx-6799fc88d8  1         2         2      4m42s
nginx-6889dfcccd5  10        10        10     55s
nginx-6799fc88d8  0         1         1      4m46s
nginx-6799fc88d8  0         0         0      4m46s
```

So, we see two `ReplicaSet` resources here—`nginx-6799fc88d8` and `nginx-6889dfcccd5`. While the `nginx-6799fc88d8` pod is slowly rolling down from 10 pods to 0, one at a time, simultaneously, the `nginx-6889dfcccd5` pod is slowly rolling up from 0 pods to 10. At any given time, the number of pods never exceeds 11. That is because `maxSurge` is set to 1, and `maxUnavailable` is 0. This is a slow rollout in action.

Tip

Ramped slow rollout is useful when we want to be cautious before we impact many users, but this strategy is extremely slow and may only suit some applications.

Let's look at the best-effort controlled rollout strategy for a faster rollout without compromising application availability.

Best-effort controlled rollout

Best-effort controlled rollout helps you roll out your deployment on a best-effort basis, and you can use it to roll out your release faster and ensure that your application is available. It can also help with applications that do not tolerate more than a certain number of replicas at a given point.

We will set `maxSurge` to 0 and `maxUnavailable` to any percentage we find suitable for remaining unavailable at a given time to implement this. It can be specified using the number of pods or as a percentage.

Tip

Using a percentage is a better option since, with this, you don't have to recalculate your `maxUnavailable` parameter if the replicas change.

Let's look at the manifest—`nginx-best-effort-controlled-rollout.yaml`:

```
...
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 25%
...
```

Let's now apply the YAML file and see what we get:

```
$ kubectl apply -f nginx-best-effort-controlled-rollout.yaml \
& kubectl get replicaset -w
deployment.apps/nginx configured
```

NAME	DESIRED	CURRENT	READY	AGE
nginx-6799fc88d8	2	0	0	20m
nginx-6889dfcccd5	8	8	8	16m
nginx-6799fc88d8	2	2	1	20m
nginx-6889dfcccd5	7	8	8	16m
...
nginx-6889dfcccd5	1	1	1	16m
nginx-6799fc88d8	9	9	8	20m
nginx-6889dfcccd5	0	1	1	16m
nginx-6799fc88d8	10	9	8	20m
nginx-6889dfcccd5	0	0	0	16m
nginx-6799fc88d8	10	10	10	20m

So, we see the `ReplicaSet` resource rolling out such that the total pods are at most 10 at any point, and the total unavailable pods are never more than 25%. You may also notice that instead of creating a new `ReplicaSet` resource, the `Deployment` resource uses an old `ReplicaSet` resource containing the `nginx:latest` image. Remember when I said the old `ReplicaSet` resource is not deleted when you update a `Deployment` resource?

Deployment resources on their own are great ways of scheduling and managing pods. However, we have overlooked an essential part of running containers in Kubernetes—exposing them to the internal or external world. Kubernetes provides several resources to help expose your workloads appropriately—primarily, `Service` and `Ingress` resources. Let’s have a look at these in the next section.

Kubernetes Services and Ingresses

It’s story time! Let’s simplify Kubernetes Services.

Imagine you have a group of friends who love to order food from your restaurant. Instead of delivering each order to their houses separately, you set up a central delivery point in their neighborhood. This delivery point (or hub) is your “service.”

In Kubernetes, a `Service` is like that central hub. It’s a way for the different parts of your application (such as your website, database, or other things) to talk to each other, even if they’re in separate containers or machines. It gives them easy-to-remember addresses to find each other without getting lost.

The `Service` resource helps expose Kubernetes workloads to the internal or external world. As we know, pods are ephemeral resources—they can come and go. Every pod is allocated a unique IP address and hostname, but once a pod is gone, the pod’s IP address and hostname change. Consider a scenario where one of your pods wants to interact with another. However, because of its transient nature, you cannot configure a proper endpoint. If you use the IP address or the hostname as the endpoint of a pod and the pod is destroyed, you will no longer be able to connect to it. Therefore, exposing a pod on its own is not a great idea.

Kubernetes provides the `Service` resource to provide a static IP address to a group of pods. Apart from exposing the pods on a single static IP address, it also provides load balancing of traffic between

pods in a round-robin configuration. It helps distribute traffic equally between the pods and is the default method of exposing your workloads.

Service resources are also allocated a static **fully qualified domain name (FQDN)** (based on the Service name). Therefore, you can use the Service resource FQDN instead of the IP address within your cluster to make your endpoints fail-safe.

Now, coming back to Service resources, there are multiple Service resource types—`ClusterIP`, `NodePort`, and `LoadBalancer`, each having its own respective use case:

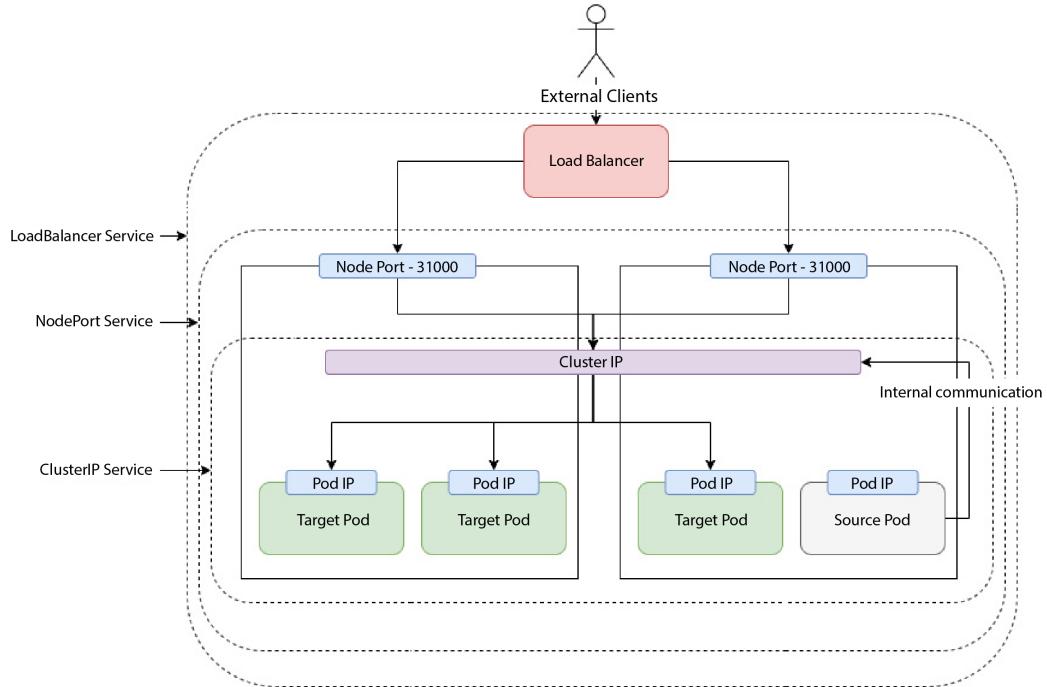


Figure 6.4 – Kubernetes Services

Let's understand each of these with the help of examples.

ClusterIP Service resources

`ClusterIP` Service resources are the default Service resource type that exposes pods within the Kubernetes cluster. It is not possible to access `ClusterIP` Service resources outside the cluster; therefore, they are never used to expose your pods to the external world. `ClusterIP` Service resources generally expose backend apps such as data stores and databases—the business and data layers—in a three-tier architecture.

Tip

When choosing between `Service` resource types, as a general rule of thumb, always start with the `ClusterIP` Service resource and change it if needed. This will ensure that only the required Services are exposed externally.

To understand `ClusterIP` Service resources better, let's create a `redis` Deployment resource first using the imperative method with the following command:

```
$ kubectl create deployment redis --image=redis
```

Let's try exposing the `redis` deployment pods using a `ClusterIP` Service resource. To access the resources for this section, cd into the following:

```
$ cd ~/modern-devops/ch6/services/
```

Let's look at the `Service` resource manifest, `redis-clusterip.yaml`, first:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: redis
  name: redis
spec:
  ports:
  - port: 6379
    protocol: TCP
    targetPort: 6379
  selector:
    app: redis
```

The `Service` resource manifest starts with `apiVersion` and `kind` as any other resource. It has a `metadata` section that contains `name` and `labels`.

The `spec` section contains the following:

- `ports`: This section includes a list of ports that we want to expose via the `Service` resource:
 - `port`: The port we wish to expose.
 - `protocol`: The protocol of the port we expose (TCP/UDP).
 - `targetPort`: The target container port where the exposed port will forward the connection. This allows us to have a port mapping similar to Docker.
- `selector`: This section contains `labels` based on which pod group is selected.

Let's apply the `Service` resource manifest using the following command and see what we get:

```
$ kubectl apply -f redis-clusterip.yaml
```

Let's run `kubectl get` to list the `Service` resource and get the cluster IP:

```
$ kubectl get service redis
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
redis    ClusterIP  10.12.6.109    <none>        6379/TCP    16s
```

We see a `redis` `Service` resource running with a `ClusterIP` type. But as this pod is not exposed externally, the only way to access it is through a second pod running within the cluster.

Let's create a `busybox` pod in interactive mode to inspect the `Service` resource and run some tests using the following command:

```
$ kubectl run busybox --rm --restart Never -it --image=busybox
/ #
```

And with this, we see a prompt. We have launched the `busybox` container and are currently within that. We will use the `telnet` application to check the connectivity between pods.

Let's telnet the cluster IP and port to see whether it's reachable using the following command:

```
/ # telnet 10.96.118.99 6379
Connected to 10.96.118.99
```

The IP/port pair is reachable from there. Kubernetes also provides an internal DNS to promote service discovery and connect to the `Service` resource. We can do a reverse nslookup on the cluster IP to get the `Service` resource's FQDN using the following command:

```
/ # nslookup 10.96.118.99
Server:          10.96.0.10
Address:         10.96.0.10:53
99.118.96.10.arpa name = redis.default.svc.cluster.local
```

As we can see, the IP address is accessible from the FQDN—`redis.default.svc.cluster.local`. We can use the entire domain or parts of it based on our location. The FQDN is formed of these parts: `<service_name>. <namespace>. svc. <cluster-domain>. local`.

Kubernetes uses **namespaces** to segregate resources. You can visualize namespaces as multiple virtual clusters within the same physical Kubernetes cluster. You can use them if many users work in multiple teams or projects. We have been working in the `default` namespace till now and will continue doing so. If your source pod is located in the same namespace as the `Service` resource, you can use `service_name` to connect to your `Service` resource—something like the following example:

```
/ # telnet redis 6379
Connected to redis
```

If you want to call a `Service` resource from a pod situated in a different namespace, you can use `<service_name>. <namespace>` instead—something like the following example:

```
/ # telnet redis.default 6379
Connected to redis.default
```

Some service meshes, such as Istio, allow multi-cluster communication. In that situation, you can also use the cluster name for connecting to the `Service` resource, but as this is an advanced topic, it is beyond the scope of this discussion.

Tip

Always use the shortest domain name possible for endpoints, as it allows for more flexibility in moving your Kubernetes resources across your environments.

`ClusterIP` Services work very well for exposing internal pods, but what if we want to expose our pods to the external world? Kubernetes offers various `Service` resource types for that; let's look at the `NodePort` `Service` resource type first.

NodePort Service resources

`NodePort` `Service` resources are used to expose your pods to the external world. Creating a `NodePort` `Service` resource spins up a `ClusterIP` `Service` resource and maps the `ClusterIP` port to a random high port number (default: 30000-32767) on all cluster nodes. You can also specify a static `NodePort` number if you so desire. So, with a `NodePort` `Service` resource, you can access your pods using the IP address of any node within your cluster and the `NodePort` of the service.

Tip

Though it is possible to specify a static `NodePort` number, you should avoid using it. That is because you might end up in port conflicts with other `Service` resources and put a high dependency on config and change management. Instead, keep things simple and use dynamic ports.

Going by the Flask application example, let's create a `flask-app` pod with the `redis` `Service` resource we created before, acting as its backend, and then we will expose the pod on `NodePort`.

Use the following command to create a pod imperatively:

```
$ kubectl run flask-app --image=<your_dockerhub_user>/python-flask-redis
```

Now, as we've created the `flask-app` pod, let's check its status using the following command:

```
$ kubectl get pod flask-app
NAME      READY   STATUS    RESTARTS   AGE
flask-app  1/1     Running   0          19s
```

The `flask-app` pod is running successfully and is ready to accept requests. It's time to understand the resource manifest for the `NodePort` Service resource, `flask-nodeport.yaml`:

```
...
spec:
  ports:
    - port: 5000
      protocol: TCP
      targetPort: 5000
  selector:
    run: flask-app
  type: NodePort
```

The manifest is similar to the `ClusterIP` manifest but contains a `type` attribute specifying the `Service` resource type—`NodePort`.

Let's apply this manifest to see what we get using the following command:

```
$ kubectl apply -f flask-nodeport.yaml
```

Now, let's list the `Service` resource to get the `NodePort` Service using the following command:

```
$ kubectl get service flask-app
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
flask-app  NodePort  10.3.240.246  <none>          5000:32618/TCP  9s
```

And we see that the type is now `NodePort`, and the container port 5000 is mapped to node port 32618.

If you are logged in to any Kubernetes node, you can access the `Service` resource using `localhost:32618`. But as we are using Google Cloud Shell, we need to SSH into a node to access the `Service` resource.

Let's list the nodes first using the following command:

```
$ kubectl get nodes
NAME      STATUS      ROLES      AGE      VERSION
gke-node-1dhh  Ready      <none>      17m      v1.26.15-gke.4901
gke-node-7lhl  Ready      <none>      17m      v1.26.15-gke.4901
gke-node-zwg1  Ready      <none>      17m      v1.26.15-gke.4901
```

And as we can see, we have three nodes. Let's SSH into the `gke-node-1dhh` node using the following command:

```
$ gcloud compute ssh gke-node-1dhh
```

Now, as we are within the `gke-node-1dhh` node, let's curl `localhost:32618` using the following command:

```
$ curl localhost:32618
Hi there! This page was last visited on 2023-06-26, 08:37:50.
```

And we get a response back! You can SSH into any node and curl the endpoint and should get a similar response.

To exit from the node and get back to the Cloud Shell prompt, run the following command:

```
$ exit
Connection to 35.202.82.74 closed.
```

And you are back at the Cloud Shell prompt.

Tip

A `NodePort Service` resource is an intermediate kind of resource. This means that while it forms an essential building block of providing external services, it is not used on its own most of the time. When you are running on the cloud, you can use `LoadBalancer Service` resources instead. Even for an on-premises setup, it makes sense not to use `NodePort` for every `Service` resource and instead use `Ingress` resources.

Now, let's look at the `LoadBalancer Service` resource used extensively to expose your Kubernetes workloads externally.

LoadBalancer Service resources

`LoadBalancer Service` resources help expose your pods on a single load-balanced endpoint. These `Service` resources can only be used within cloud platforms and platforms that provide Kubernetes controllers with access to spin up external network resources. A `LoadBalancer Service` practically spins up a `NodePort Service` resource and then requests the Cloud API to spin up a load balancer in front of the node ports. That way, it provides a single endpoint to access your `Service` resource from the external world.

Spinning up a `LoadBalancer Service` resource is simple—just set the type to `LoadBalancer`.

Let's expose the Flask application as a load balancer using the following manifest—`flask-loadbalancer.yaml`:

```
...
spec:
  type: LoadBalancer
...
```

Now, let's apply the manifest using the following command:

```
$ kubectl apply -f flask-loadbalancer.yaml
```

Let's get the Service resource to notice the changes using the following command:

```
$ kubectl get svc flask-app
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
flask-app  LoadBalancer  10.3.240.246  34.71.95.96  5000:32618
```

The Service resource type is now LoadBalancer. As you can see, it now contains an external IP along with the cluster IP.

You can then curl on the external IP on port 5000 using the following command:

```
$ curl 34.71.95.96:5000
Hi there! This page was last visited on 2023-06-26, 08:37:50.
```

And you get the same response as before. Your Service resource is now running externally.

Tip

LoadBalancer Service resources tend to be expensive as every new resource spins up a network load balancer within your cloud provider. If you have HTTP-based workloads, use Ingress resources instead of LoadBalancer to save on resource costs and optimize traffic as they spin up an application load balancer instead.

While Kubernetes Services form the basic building block of exposing your container applications internally and externally, Kubernetes also provides Ingress resources for additional fine-grained control over traffic. Let's have a look at this in the next section.

Ingress resources

Imagine you have a beautiful front entrance to your restaurant where customers come in. They walk through this main entrance to reach different parts of your restaurant, such as the dining area or the bar. This entrance is like your "ingress."

In Kubernetes, an Ingress is like that front entrance. It helps manage external access to the Services inside your cluster. Instead of exposing each Service individually, you can use an Ingress to decide how people from the outside can reach different parts of your application.

In simple terms, a Kubernetes Service is like a central delivery point for your application's different parts, and an Ingress is like a front entrance that helps people from the outside find and access those parts easily.

Ingress resources act as reverse proxies into Kubernetes. You don't need a load balancer for every application you run within your estate, as load balancers normally forward traffic and don't require high levels of computing power. Therefore, spinning up a load balancer for everything does not make sense.

Therefore, Kubernetes provides a way of routing external traffic into your cluster via Ingress resources. These resources help you subdivide traffic according to multiple conditions. Some of these are set out here:

- Based on the URL path
- Based on the hostname
- A combination of the two

The following diagram illustrates how Ingress resources work:

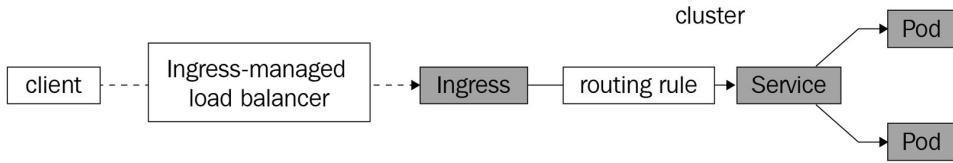


Figure 6.5 – Kubernetes Ingress resources

Ingress resources require an ingress controller to work. While most cloud providers have a controller installed, you must install an ingress controller on-premises or in a self-managed Kubernetes cluster. For more details on installing an ingress controller, refer to <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>. You can install more than one ingress controller, but you will have to annotate your manifests to denote explicitly which controller the Ingress resource should use.

For this chapter, we will use the **nginx ingress controller** (<https://github.com/kubernetes/ingress-nginx>), which is actively maintained by the Kubernetes open source community. The reason we use this instead of the native GKE ingress controller is that we want to make our setup as cloud-agnostic as possible. The nginx ingress controller is also feature-packed and runs the same irrespective of the environment. So, if we want to migrate to another cloud provider, we will retain all the features we had with Ingress resources before and do an exact like-for-like migration.

To understand how the nginx ingress controller works on GKE (or any other cloud), let's look at the following diagram:

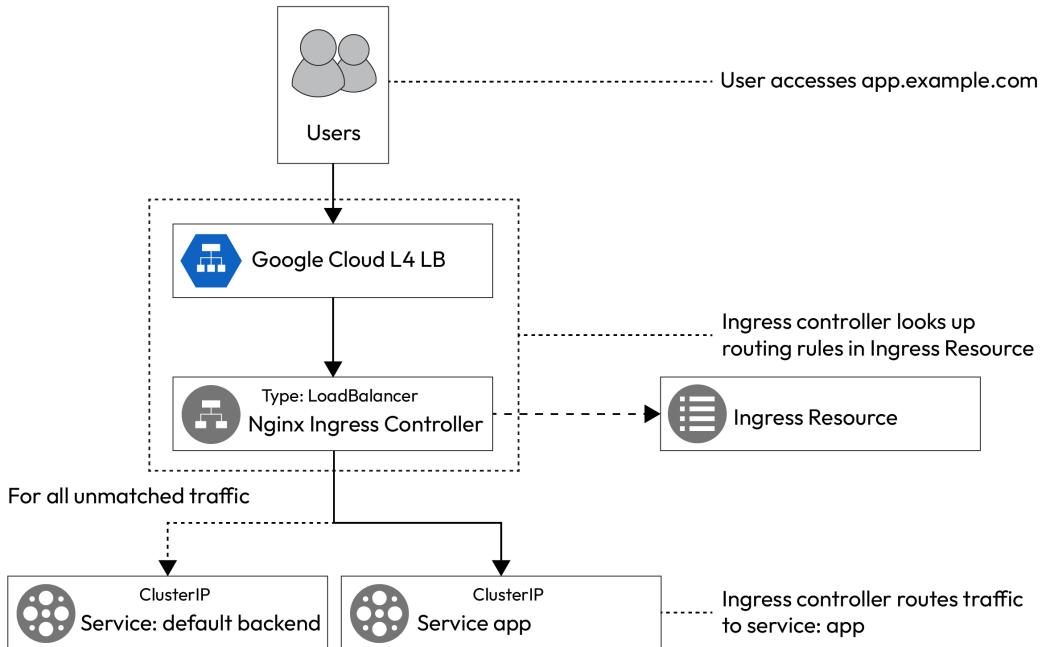


Figure 6.6 – nginx ingress controller on GKE

The client connects to the `Ingress` resource via an ingress-managed load balancer, and the traffic moves to the ingress controllers that act as the load balancer's backend. The ingress controllers then route the traffic to the correct `Service` resource via routing rules defined on the `Ingress` resource.

Now, let's go ahead and install the `nginx` ingress controller using the following command:

```
$ kubectl apply -f \
https://raw.githubusercontent.com/kubernetes/ingress-nginx\
/controller-v1.8.0/deploy/static/provider/cloud/deploy.yaml
```

This will boot up several resources under the `ingress-nginx` namespace. Most notable is the `ingress-nginx-controller` Deployment, which is exposed via the `ingress-nginx-controller` LoadBalancer Service.

Let's now expose the `flask-app` Service via an `Ingress` resource, but before we do that, we will have to expose the `flask-app` Service on a ClusterIP instead, so let's apply the relevant manifest using the following command:

```
$ kubectl apply -f flask-clusterip.yaml
```

The next step is to define an `Ingress` resource. Remember that as GKE is running on a public cloud, it has the ingress controllers installed and running. So, we can simply go and create an ingress manifest—`flask-basic-ingress.yaml`:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-app
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  defaultBackend:
    service:
      name: flask-app
      port:
        number: 5000
```

This resource defines a default backend that passes all traffic to the `flask-app` pod, so it is counter-productive, but let's look at it for simplicity.

Apply the manifest using the following command:

```
$ kubectl apply -f flask-basic-ingress.yaml
```

Now, let's list the `Ingress` resources using the following command:

```
$ kubectl get ingress flask-app
NAME      CLASS      HOSTS      ADDRESS      PORTS      AGE
flask-app  <none>    *          80          40s
```

We can see that the `flask-app` `Ingress` resource is now listed with `HOSTS *`. That means that this would listen on all hosts on all addresses. So, anything that does not match other `Ingress` rules will be routed here. As mentioned, we need the `nginx-ingress-controller` Service external IP address to invoke all Services exposed via `Ingress`. To get the external IP of the `nginx-ingress-controller` Service, run the following command:

```
$ kubectl get svc ingress-nginx-controller -n ingress-nginx
NAME                  TYPE      EXTERNAL-IP
ingress-nginx-controller LoadBalancer 34.120.27.34
```

We see an external IP allocated to it, which we will use further.

Important note

Remember that `Ingress` rules take a while to propagate across the cluster, so if you receive an error initially when you curl the endpoint, wait for 5 minutes, and you should get the response back.

Let's curl this IP and see what we get using the following command:

```
$ curl 34.120.27.34
Hi there! This page was last visited on 2023-06-26, 09:28:26.
```

Now, let's clean up the `Ingress` resource using the following command:

```
$ kubectl delete ingress flask-app
```

The simple Ingress rule is counterproductive as it routes all traffic to a single `Service` resource. The idea of Ingress is to use a single load balancer to route traffic to multiple targets. Let's look at two ways to do this—**path-based** and **name-based** routing.

Path-based routing

Let's consider an application with two versions, v1 and v2, and you want both to co-exist on a single endpoint. You can use **path-based routing** for such a scenario.

Let's create the two application versions first using the imperative method by running the following commands:

```
$ kubectl run nginx-v1 --image=bharamicrosystems/nginx:v1
$ kubectl run nginx-v2 --image=bharamicrosystems/nginx:v2
```

Now, expose the two pods as ClusterIP Service resources using the following commands:

```
$ kubectl expose pod nginx-v1 --port=80
$ kubectl expose pod nginx-v2 --port=80
```

We will then create an Ingress resource using the following manifest file, `nginx-app-path-ingress.yaml`, which will expose two endpoints—`<external-ip>/v1`, which routes to the v1 Service resource, and `<external-ip>/v2`, which routes to the v2 Service resource:

```
...
spec:
  rules:
  - http:
      paths:
      - path: /v1
        pathType: Prefix
        backend:
          service:
            name: nginx-v1
            port:
              number: 80
      - path: /v2
        pathType: Prefix
        backend:
          service:
            name: nginx-v2
```

```
port:  
  number: 80
```

The Ingress manifest contains several rules. The `http` rule has two paths—`/v1` and `/v2`, having the `pathType` value set to `Prefix`. Therefore, any traffic arriving on a URL that starts with `/v1` is routed to the `nginx-v1` Service resource on port 80, and traffic arriving on `/v2` is routed to the `nginx-v2` Service resource on port 80.

Let's apply the manifest by using the following command:

```
$ kubectl apply -f nginx-app-path-ingress.yaml
```

Now, let's list the Ingress resources by running the following command:

```
$ kubectl get ingress nginx-app -w  
NAME      CLASS      HOSTS      ADDRESS          PORTS      AGE  
nginx-app  <none>    *          34.120.27.34   80        114s
```

Now that we have the external IP, we can `curl` both endpoints to see what we get using the following commands:

```
$ curl 34.120.27.34/v1/  
This is version 1  
$ curl 34.120.27.34/v2/  
This is version 2
```

Sometimes, a path-based route is not always feasible, as you might not want your users to remember the path of multiple applications. However, you can still run multiple applications using a single Ingress endpoint—that is, by using **name-based routing**.

Name-based routing

Name-based or **FQDN-based routing** relies on the `host` header we pass while making an HTTP request. The Ingress resource can route based on the header. For example, if we want to access the `v1` Service resource, we can use `v1.example.com`, and for the `v2` Service resource, we can use the `v2.example.com` URL.

Let's now have a look at the `nginx-app-host-ingress.yaml` manifest to understand this concept further:

```
...  
spec:  
  rules:  
  - host: v1.example.com  
    http:  
      paths:  
      - path: "/"  
        pathType: Prefix  
      backend:
```

```

service:
  name: nginx-v1
  port:
    number: 80
- host: v2.example.com
  http:
    paths:
    - path: "/"
      pathType: Prefix
      backend:
        service:
          name: nginx-v2
          port:
            number: 80

```

The manifest now contains multiple hosts—v1.example.com routing to nginx-v1, and v2.example.com routing to nginx-v2.

Now, let's apply this manifest and get the Ingress using the following commands:

```

$ kubectl apply -f nginx-app-host-ingress.yaml
$ kubectl get ingress
NAME      HOSTS                      ADDRESS      PORTS
nginx-app  v1.example.com,v2.example.com  34.120.27.34  80

```

This time, we can see that two hosts are defined, v1.example.com and v2.example.com, running on the same address. Before we hit those endpoints, we need to make an entry on the /etc/hosts file to allow our machine to resolve v1.example.com and v2.example.com to the Ingress address.

Edit the /etc/hosts file and add the following entry at the end:

```
<Ingress_External_IP> v1.example.com v2.example.com
```

Now, let's curl both endpoints and see what we get:

```

$ curl v1.example.com
This is version 1
$ curl v2.example.com
This is version 2

```

And, as we can see, the name-based routing is working correctly! You can create a more dynamic setup by combining multiple hosts and path-based routing.

Service, Ingress, Pod, Deployment, and ReplicaSet resources help us to maintain a set number of replicas within Kubernetes and help to serve them under a single endpoint. As you may have noticed, they are linked together using a combination of labels and matchLabels attributes. The following diagram will help you visualize this:

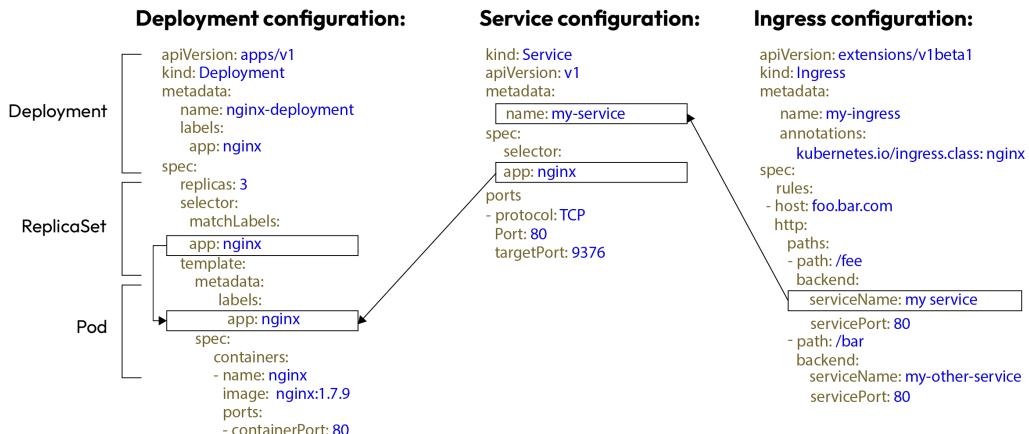


Figure 6.7 – Linking Deployment, Service, and Ingress

Till now, we have been scaling our pods manually, but a better way would be to autoscale the replicas based on resource utilization and traffic. Kubernetes provides a resource called `HorizontalPodAutoscaler` to handle that requirement.

Horizontal Pod autoscaling

Imagine you're the manager of a snack bar at a park. On a sunny day, lots of people come to enjoy the park, and they all want snacks. Now, you have a few workers at your snack bar who make and serve the snacks.

Horizontal Pod autoscaling in Kubernetes is like having magical helpers who adjust the number of snack makers (pods) based on how many people want snacks (traffic).

Here's how it works:

- **Average days:** You might only need one or two snack makers on regular days with fewer people. In Kubernetes terms, you have a few pods running your application.
- **Busy days:** But when it's a sunny weekend, and everyone rushes to the park, more people want snacks. Your magical helpers (Horizontal Pod autoscaling) notice the increase in demand. They say, "*We need more snack makers!*" So, more snack makers (pods) are added automatically to handle the rush.
- **Scaling down:** Once the sun sets and the crowd leaves, you don't need as many snack makers anymore. Your magical helpers see the decrease in demand and say, "*We can have fewer snack makers now.*" So, extra snack makers (pods) are removed, saving resources.

- **Automatic adjustment:** These magical helpers monitor the crowd and adjust the number of snack makers (pods) in real time. When the demand goes up, they deploy more. When it goes down, they remove some.

In the same way, Kubernetes Horizontal pod autoscaling watches how busy your application is. If there's more traffic (more people wanting your app), it automatically adds more pods. If things quieten down, it scales down the number of pods. This helps your app handle varied traffic without you manually doing everything.

So, Horizontal pod autoscaling is like having magical assistants that ensure your application has the correct number of workers (pods) to handle the crowd (traffic) efficiently.

`HorizontalPodAutoscaler` is a Kubernetes resource that helps you to update replicas within your `ReplicaSet` resources based on defined factors, the most common being CPU and memory.

To understand this better, let's create an `nginx` Deployment, and this time, we will set the resource limits within the pod. Resource limits are a vital element that enables `HorizontalPodAutoscaler` resources to function. It relies on the percentage utilization of the limits to decide when to spin up a new replica. We will use the following `nginx-autoscale-deployment.yaml` manifest under `~/modern-devops/ch6/deployments` for this exercise:

```
...
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            limits:
              cpu: 200m
              memory: 200Mi
...
...
```

Use the following command to perform a new deployment:

```
$ kubectl apply -f nginx-autoscale-deployment.yaml
```

Let's expose this deployment with a `LoadBalancer` Service resource and get the external IP:

```
$ kubectl expose deployment nginx --port 80 --type LoadBalancer
$ kubectl get svc nginx
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
nginx    LoadBalancer  10.3.243.225  34.123.234.57  80:30099/TCP
```

Now, let's autoscale this deployment. The Deployment resource needs at least 1 pod replica and can have a maximum of 5 pod replicas while maintaining an average CPU utilization of 25%. Use the following command to create a HorizontalPodAutoscaler resource:

```
$ kubectl autoscale deployment nginx --cpu-percent=25 --min=1 --max=5
```

Now that we have the HorizontalPodAutoscaler resource created, we can load test the application using the `hey` load testing utility preinstalled in Google Cloud Shell. But before you fire the load test, open a duplicate shell session and watch the Deployment resource using the following command:

```
$ kubectl get deployment nginx -w
```

Open another duplicate shell session and watch the HorizontalPodAutoscaler resource using the following command:

```
$ kubectl get hpa nginx -w
```

Now, in the original window, run the following command to fire a load test:

```
$ hey -z 120s -c 100 http://34.123.234.57
```

It will start a load test for 2 minutes, with 10 concurrent users continuously hammering the Service. You will see the following output if you open the window where you're watching the HorizontalPodAutoscaler resource. As soon as we start firing the load test, the average utilization reaches 46%. The HorizontalPodAutoscaler resource waits for some time, then it increases the replicas, first to 2, then to 4, and finally to 5. When the test is complete, the utilization drops quickly to 27%, 25%, and finally, 0%. When the utilization goes to 0%, the HorizontalPodAutoscaler resource spins down the replicas from 5 to 1 gradually:

```
$ kubectl get hpa nginx -w
  NAME   REFERENCE      TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
  nginx  deployment/nginx <unknown>/25%  1          5          1          32s
  nginx  deployment/nginx 46%/25%    1          5          1          71s
  nginx  deployment/nginx 46%/25%    1          5          2          92s
  nginx  deployment/nginx 92%/25%    1          5          4          2m2s
  nginx  deployment/nginx 66%/25%    1          5          5          2m32s
  nginx  deployment/nginx 57%/25%    1          5          5          2m41s
  nginx  deployment/nginx 27%/25%    1          5          5          3m11s
  nginx  deployment/nginx 23%/25%    1          5          5          3m41s
  nginx  deployment/nginx 0%/25%     1          5          4          4m23s
  nginx  deployment/nginx 0%/25%     1          5          2          5m53s
  nginx  deployment/nginx 0%/25%     1          5          1          6m30s
```

Likewise, we will see the replicas of the Deployment changing when the HorizontalPodAutoscaler resource actions the changes:

```
$ kubectl get deployment nginx -w
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx  1/1     1           1           18s
nginx  1/2     1           1           77s
nginx  2/2     2           2           79s
nginx  2/4     2           2           107s
nginx  3/4     4           3           108s
nginx  4/4     4           4           109s
nginx  4/5     4           4           2m17s
nginx  5/5     5           5           2m19s
nginx  4/4     4           4           4m23s
nginx  2/2     2           2           5m53s
nginx  1/1     1           1           6m30s
```

Besides CPU and memory, you can use other parameters to scale your workloads, such as network traffic. You can also use external metrics such as latency and other factors that you can use to decide when to scale your traffic.

Tip

While you should use the `HorizontalPodAutoscaler` resource with CPU and memory, you should also consider scaling on external metrics such as response time and network latency. That will ensure better reliability as they directly impact customer experience and are crucial to your business.

Till now, we have been dealing with stateless workloads. However, pragmatically speaking, some applications need to save the state. Let's look at some considerations for managing stateful applications.

Managing stateful applications

Imagine you're a librarian in a magical library. You have a bunch of enchanted books that store valuable knowledge. Each book has a unique story and is kept in a specific spot on the shelf. These books are like your "stateful applications," and managing them requires extra care.

Managing stateful applications in the world of technology is like taking care of these magical books in your library.

Here's how it works:

- **Stateful books:** Some books in your library are “stateful.” This means they hold vital information that changes over time, such as bookmarks or notes from readers.
- **Fixed locations:** Just as each book has a specific place on the shelf, stateful applications must also be in particular locations. They might need to be on certain machines or use specific storage to keep their data safe.
- **Maintaining inventory:** You must remember where each book is placed. Similarly, managing stateful applications means remembering their exact locations and configurations.
- **Careful handling:** When someone borrows a stateful book, you must ensure they return it in good condition. With stateful applications, you must handle updates and changes carefully to avoid losing important data.
- **Backup spells:** Sometimes, you cast a spell to create a copy of a book, just in case something happens to the original. With stateful applications, you back up your data to restore it if anything goes wrong.
- **Moving with caution:** If you need to rearrange the library, you move books one at a time so that nothing gets lost. Similarly, with stateful applications, if you need to move them between machines or storage, it's done cautiously to avoid data loss.

In the world of technology, managing stateful applications means taking extra care of applications that hold important data. You ensure they're placed in the right spots, handle updates carefully, and create backups to keep valuable information safe, just like how you protect your enchanted books in the magical library!

Deployment resources are beneficial for stateless workloads, as they do not need to add any state considerations while updating ReplicaSet resources, but they cannot work effectively with stateful workloads. To manage such workloads, you can use a StatefulSet resource.

StatefulSet resources

StatefulSet resources help manage stateful applications. They are similar to Deployment resources, but unlike a Deployment resource, they also keep track of state and require Volume and Service resources to operate. StatefulSet resources maintain a sticky identity for each pod. This means that the volume mounted on one pod cannot be used by the other. In a StatefulSet resource, Kubernetes orders pods by numbering them instead of generating a random hash. Pods within a StatefulSet resource are also rolled out and scaled-in in order. If a particular pod goes down and is recreated, the same volume is mounted to the pod.

The following diagram illustrates a `StatefulSet` resource:

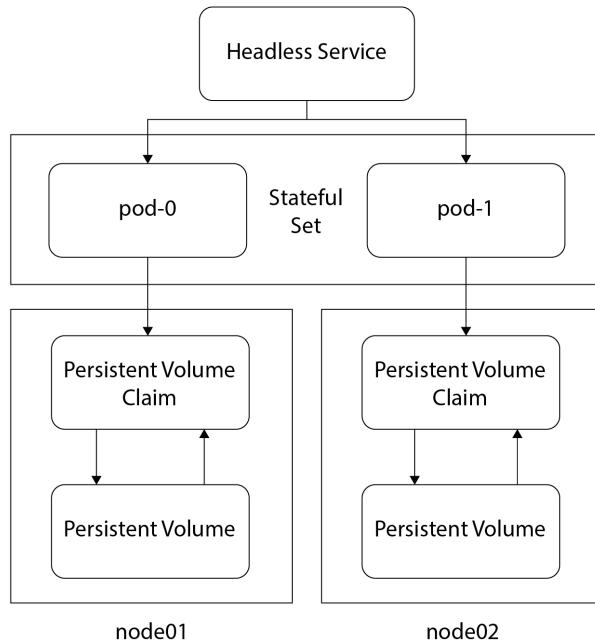


Figure 6.8 – StatefulSet resource

A `StatefulSet` resource has a stable and unique network identifier, therefore, it requires a headless `Service` resource. Headless Services are `Service` resources that do not have a cluster IP. Instead, the Kubernetes DNS resolves the `Service` resource's FQDN directly to the pods.

As a `StatefulSet` resource is supposed to persist data, it requires Persistent Volumes to operate. Therefore, let's look at how to manage volumes using Kubernetes.

Managing Persistent Volumes

Persistent Volumes are Kubernetes resources that deal with storage. They can help you manage and mount **hard disks**, **SSDs**, **filestores**, and other block and network storage entities. You can provision Persistent Volumes manually or use dynamic provisioning within Kubernetes. When you use dynamic provisioning, Kubernetes will request the cloud provider via the cloud controller manager to provide the required storage. Let's look at both methods to understand how they work.

Static provisioning

Static provisioning is the traditional method of provisioning volumes. It requires someone (typically an administrator) to manually provision a disk and create a `PersistentVolume` resource using the disk information. The developer can then use the `PersistentVolume` resource within their `StatefulSet` resource, as in the following diagram:

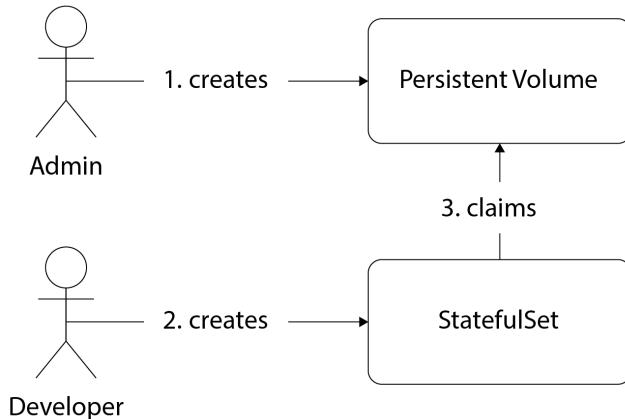


Figure 6.9 – Static provisioning

Let's now look at a static provisioning example.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch6/statefulsets/
```

So, we first need to create a disk within the cloud platform. Since we're using Google Cloud, let's proceed and use the `gcloud` commands to do so.

Use the following command to create a persistent zonal disk. Ensure that you use the same zone as your Kubernetes cluster. As we are using the `us-central1-a` zone for the Kubernetes cluster, we will use the same in the following command:

```
$ gcloud compute disks create nginx-manual \
--size 50GB --type pd-ssd --zone us-central1-a
Created [https://www.googleapis.com/compute/v1/projects/<project_id>/zones/us-central1-a/
disks/nginx-manual].
NAME          ZONE          SIZE_GB   TYPE     STATUS
nginx-manual  us-central1-a  50        pd-ssd   READY
```

As the disk is now ready, we can then create a `PersistentVolume` resource from it.

The manifest file, `nginx-manual-pv.yaml`, looks like this:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nginx-manual-pv
  labels:
    usage: nginx-manual-disk
spec:
  capacity:
    storage: 50G
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: nginx-manual
    fsType: ext4
```

The `spec` section contains `capacity`, `accessModes`, and the kind of disk it needs to provision. You can specify one or more access modes to a `PersistentVolume`:

- `ReadWriteOnce`: Only one pod can read and write to the disk at a time; therefore, you cannot mount such a volume to multiple pods
- `ReadOnlyMany`: Multiple pods can read from the same volume simultaneously, but no pod can write to the disk
- `ReadWriteMany`: Multiple pods can read and write to the same volume at once

Tip

Not all kinds of storage support all access modes. You need to decide the volume type during the initial requirement analysis and architectural assessment phase.

OK—let's now go and apply the manifest to provision the `PersistentVolume` resource using the following command:

```
$ kubectl apply -f nginx-manual-pv.yaml
```

Let's now check whether the Persistent Volume is available by using the following command:

```
$ kubectl get pv
NAME          CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS
nginx-manual-pv  50G       RWO          Retain        Available
```

As the Persistent Volume is now available, we must create a headless Service resource to help maintain network identity in the StatefulSet resource. The following `nginx-manual-service.yaml` manifest describes it:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-manual
  labels:
    app: nginx-manual
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx-manual
```

It is very similar to the regular Service resource, except that we have specified `clusterIP` as `None`.

Now, let's go and apply the manifest using the following command:

```
$ kubectl apply -f nginx-manual-service.yaml
```

As the Service resource is created, we can create a StatefulSet resource that uses the created PersistentVolume and Service resources. The StatefulSet resource manifest, `nginx-manual-statefulset.yaml`, looks like this:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-manual
spec:
  selector:
    matchLabels:
      app: nginx-manual
  serviceName: "nginx-manual"
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-manual
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
```

```

  name: html
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 40Gi
    selector:
      matchLabels:
        usage: nginx-manual-disk

```

The manifest contains various sections. While most are similar to the Deployment resource manifest, this requires a volume definition and a separate volumeClaimTemplates section. The volumeClaimTemplates section consists of the accessModes, resources, and selector sections. The selector section defines the matchLabels attribute, which helps to select a particular PersistentVolume resource. In this case, it selects the PersistentVolume resource we defined previously. It also contains the serviceName attribute that defines the headless Service resource it will use.

Now, let's go ahead and apply the manifest using the following command:

```
$ kubectl apply -f nginx-manual-statefulset.yaml
```

Now, let's inspect a few elements to see where we are. The StatefulSet resource creates a PersistentVolumeClaim resource to claim the PersistentVolume resource we created before.

Get the PersistentVolumeClaim resource using the following command:

```
$ kubectl get pvc
NAME           STATUS  VOLUME          CAPACITY  ACCESS MODES
html-nginx-manual-0  Bound  nginx-manual-pv  50G       RWO
```

As we can see, the StatefulSet resource has created a PersistentVolumeClaim resource called html-nginx-manual-0 that is bound to the nginx-manual-pv PersistentVolume resource. Therefore, manual provisioning has worked correctly.

If we query the PersistentVolume resource using the following command, we will see that the status is now showing as Bound:

```
$ kubectl get pv
NAME          CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS
nginx-manual-pv  50G       RWO          Retain        Bound
```

Now, let's have a look at the pods using the following command:

```
$ kubectl get pod
NAME        READY  STATUS   RESTARTS  AGE
nginx-manual-0  1/1   Running  0         14s
```

As we see, the `StatefulSet` resource has created a pod and appended it with a serial number instead of a random hash. It wants to maintain ordering between the pods and mount the same volumes to the pods they previously mounted.

Now, let's open a shell into the pod and create a file within the `/usr/share/nginx/html` directory using the following commands:

```
$ kubectl exec -it nginx-manual-0 -- /bin/bash
root@nginx-manual-0:/# cd /usr/share/nginx/html/
root@nginx-manual-0:/usr/share/nginx/html# echo 'Hello, world' > index.html
root@nginx-manual-0:/usr/share/nginx/html# exit
```

Great! So, let's go ahead and delete the pod and see whether we can get the file in the same location again using the following commands:

```
$ kubectl delete pod nginx-manual-0
$ kubectl get pod
NAME           READY   STATUS      RESTARTS   AGE
nginx-manual-0  1/1     Running    0          3s
$ kubectl exec -it nginx-manual-0 -- /bin/bash
root@nginx-manual-0:/# cd /usr/share/nginx/html/ && cat index.html
Hello, world
root@nginx-manual-0:/usr/share/nginx/html# exit
```

And, as we can see, the file still exists, even after we deleted the pod.

Static provisioning isn't one of the best ways of doing things, as you must manually keep track and provision volumes. That involves a lot of manual activities and may be error-prone. Some organizations that want to keep a line between Dev and Ops can use this technique. Kubernetes allows this provision. However, for more DevOps-friendly organizations, **dynamic provisioning** is a better way of doing it.

Dynamic provisioning

Dynamic provisioning is when Kubernetes provides storage resources for you by interacting with the cloud provider. When we provisioned the disk manually, we interacted with the cloud APIs using the `gcloud` command line. What if your organization decides to move to some other cloud provider later? That would break many existing scripts, and you would have to rewrite the storage provisioning steps. Kubernetes is inherently portable and platform-independent. You can provision resources in the same way on any cloud platform.

But then, different cloud providers have different storage offerings. How would Kubernetes know what kind of storage it needs to provision? Well, Kubernetes uses `StorageClass` resources for that. `StorageClass` resources are Kubernetes resources that define the type of storage they need to provide when someone uses it.

The following diagram illustrates dynamic provisioning:

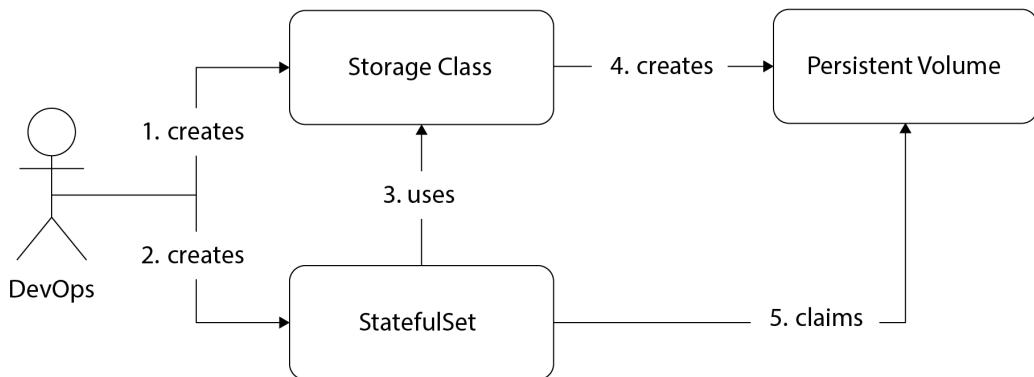


Figure 6.10 – Dynamic provisioning

Let's see an example storage class manifest, `fast-storage-class.yaml`, that provisions an SSD within GCP:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  
```

The `StorageClass` resource contains a provisioner and any parameters the provisioner requires. You may have noticed that I have kept the name `fast` instead of `gce-ssd` or similar. That is because we want to keep the names as generic as possible.

Tip

Keep generic storage class names such as `fast`, `standard`, `block`, and `shared`, and avoid names specific to the cloud platform. Because storage class names are used in Persistent Volume claims, if you migrate to another cloud provider, you may end up changing a lot of manifests just to avoid confusion.

Let's go ahead and apply the manifest using the following command:

```
$ kubectl apply -f fast-storage-class.yaml
```

As the `StorageClass` resource is created, let's use it to provision an `nginx` `StatefulSet` resource dynamically.

We need to create a `Service` resource manifest, `nginx-dynamic-service.yaml`, first:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-dynamic
  labels:
    app: nginx-dynamic
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx-dynamic
```

The manifest is very similar to the manual `Service` resource. Let's go ahead and apply it using the following command:

```
$ kubectl apply -f nginx-dynamic-service.yaml
```

Now, let's look at the `StatefulSet` resource manifest, `nginx-dynamic-statefulset.yaml`:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-dynamic
spec:
...
  serviceName: "nginx-dynamic"
  template:
    spec:
      containers:
      - name: nginx
        image: nginx
        volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
...
  volumeClaimTemplates:
  - metadata:
      name: html
    spec:
      storageClassName: "fast"
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 40Gi
```

The manifest is similar to the manual one, but this one contains the `storageClassName` attribute in the `volumeClaimTemplates` section and lacks the `selector` section, as we are dynamically provisioning the storage. Use the following command to apply the manifest:

```
$ kubectl apply -f nginx-dynamic-statefulset.yaml
```

As the `StatefulSet` resource is created, let's go ahead and check the `PersistentVolumeClaim` and `PersistentVolume` resources using the following commands:

```
$ kubectl get pvc
NAME           STATUS  VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS
html-nginx-dynamic-0  Bound   pvc-6b78  40Gi     RWO          fast
$ kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM
pvc-6b78  40Gi     RWO          Delete        Bound   default/html-nginx-dynamic-0
```

And we can see that the claim is bound to a Persistent Volume that is dynamically provisioned. Now, let's proceed and run the following command to do similar tests with this `StatefulSet` resource.

Let's create a file in the `nginx-dynamic-0` pod using the following command:

```
$ kubectl exec -it nginx-dynamic-0 -- bash
root@nginx-dynamic-0:/# cd /usr/share/nginx/html/
root@nginx-dynamic-0:/usr/share/nginx/html# echo 'Hello, dynamic world' > index.html
root@nginx-dynamic-0:/usr/share/nginx/html# exit
```

Now, delete the pod and open a shell session again to check whether the file exists by using the following commands:

```
$ kubectl delete pod nginx-dynamic-0
$ kubectl get pod nginx-dynamic-0
NAME      READY  STATUS    RESTARTS  AGE
nginx-dynamic-0  1/1   Running   0          13s
$ kubectl exec -it nginx-dynamic-0 -- bash
root@nginx-dynamic-0:/# cd /usr/share/nginx/html/
root@nginx-dynamic-0:/usr/share/nginx/html# cat index.html
Hello, dynamic world
root@nginx-dynamic-0:/usr/share/nginx/html# exit
```

And as we can see, the file exists in the volume, even if the pod was deleted. That is dynamic provisioning in action for you!

You will have observed that we have used the `kubectl` command multiple times throughout this chapter. When you perform activities throughout the day, using shortcuts and best practices wherever you can makes sense. Let's look at some best practices while using `kubectl`.

Kubernetes command-line best practices, tips, and tricks

For seasoned Kubernetes developers and administrators, `kubectl` is a command they run most of the time. The following steps will simplify your life, save you a ton of time, let you focus on more essential activities, and set you apart from the rest.

Using aliases

Most system administrators use aliases for an excellent reason—they save valuable time. Aliases in Linux are different names for commands, and they are mostly used to shorten the most frequently used commands; for example, `ls -l` becomes `ll`.

You can use the following aliases with `kubectl` to make your life easier.

k for kubectl

Yes—that’s right. By using the following alias, you can use `k` instead of typing `kubectl`:

```
$ alias k='kubectl'
$ k get node
NAME           STATUS   ROLES      AGE     VERSION
kind-control-plane  Ready    master    5m7s   v1.26.1
kind-worker      Ready    <none>   4m33s  v1.26.1
```

That will save a lot of time and hassle.

Using kubectl --dry-run

`kubectl --dry-run` helps you to generate YAML manifests from imperative commands and saves you a lot of typing time. You can write an imperative command to generate a resource and append that with a `--dry-run=client -o yaml` string to generate a YAML manifest from the imperative command. The command does not create the resource within the cluster, but instead just outputs the manifest. The following command will generate a Pod manifest using `--dry-run`:

```
$ kubectl run nginx --image=nginx --dry-run=client -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: nginx
    name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    resources: {}
  dnsPolicy: ClusterFirst
```

```
restartPolicy: Always
status: {}
```

And you now have the skeleton YAML file that you can edit according to your liking.

Now, imagine typing this command multiple times during the day! At some point, it becomes tiring. Why not shorten it by using the following alias?

```
$ alias kdr='kubectl --dry-run=client -o yaml'
```

You can then use the alias to generate other manifests.

To generate a Deployment resource manifest, use the following command:

```
$ kdr create deployment nginx --image=nginx
```

You can use the dry run to generate almost all resources from imperative commands. However, some resources do not have an imperative command, such as a DaemonSet resource. You can generate a manifest for the closest resource and modify it for such resources. A DaemonSet manifest is very similar to a Deployment manifest, so you can generate a Deployment manifest and change it to match the DameonSet manifest.

Now, let's look at some of the most frequently used `kubectl` commands and their possible aliases.

kubectl apply and delete aliases

If you use manifests, you will use the `kubectl apply` and `kubectl delete` commands most of the time within your cluster, so it makes sense to use the following aliases:

```
$ alias kap='kubectl apply -f'
$ alias kad='kubectl delete -f'
```

You can then use them to apply or delete resources using the following commands:

```
$ kap nginx-deployment.yaml
$ kad nginx-deployment.yaml
```

While troubleshooting containers, most of us use busybox. Let's see how to optimize it.

Troubleshooting containers with busybox using an alias

We use the following commands to open a busybox session:

```
$ kubectl run busybox-test --image=busybox -it --rm --restart=Never -- <cmd>
```

Now, opening several busybox sessions during the day can be tiring. How about minimizing the overhead by using the following alias?

```
$ alias kbb='kubectl run busybox-test --image=busybox -it --rm --restart=Never --'
```

We can then open a shell session to a new busybox pod using the following command:

```
$ kbb sh  
/ #
```

Now, that is much cleaner and easier. Likewise, you can also create aliases of other commands that you use frequently. Here's an example:

```
$ alias kgp='kubectl get pods'  
$ alias kgn='kubectl get nodes'  
$ alias kgs='kubectl get svc'  
$ alias kdb='kubectl describe'  
$ alias kl='kubectl logs'  
$ alias ke='kubectl exec -it'
```

And so on, according to your needs. You may also be used to completion within bash, where your commands autocomplete when you press *Tab* after typing a few words. `kubectl` also provides completion of commands, but not by default. Let's now look at how to enable `kubectl` completion within bash.

Using `kubectl` bash completion

To enable `kubectl` bash completion, use the following command:

```
$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

The command adds the `kubectl completion bash` command as a source to your `.bashrc` file. So, the next time you log in to your shell, you should be able to use `kubectl` autocomplete. That will save you a ton of time when typing commands.

Summary

We began this chapter by managing pods with Deployment and ReplicaSet resources and discussed some critical Kubernetes deployment strategies. We then looked into Kubernetes service discovery and models and understood why we required a separate entity to expose containers to the internal or external world. We then looked at different Service resources and where to use them. We talked about Ingress resources and how to use them to create reverse proxies for our container workloads. We then delved into Horizontal Pod autoscaling and used multiple metrics to scale our pods automatically.

We looked at state considerations and learned about static and dynamic storage provisioning using `PersistentVolume`, `PersistentVolumeClaim`, and `StorageClass` resources, and talked about some best practices surrounding them. We looked at `StatefulSet` resources as essential resources that help you schedule and manage stateful containers. Finally, we looked at some best practices, tips, and tricks surrounding the `kubectl` command line and how to use them effectively.

The topics covered in this and the previous chapter are just the core of Kubernetes. Kubernetes is a vast tool with enough functionality to write an entire book, so these chapters only give you the gist of what it is all about. Please feel free to read about the resources in detail in the Kubernetes official documentation at <https://kubernetes.io>.

In the next chapter, we will delve into the world of the cloud and look at **Container-as-a-Service (CaaS)** and serverless offerings for containers.

Questions

1. A Kubernetes Deployment deletes an old `ReplicaSet` resource when the image is updated. (True/False)
2. What are the primary deployment strategies supported by Kubernetes? (Choose two)
 - A. Recreate
 - B. Rolling update
 - C. Ramped slow rollout
 - D. Best-effort controlled rollout
3. Which types of resources can you use to expose containers externally? (Choose three)
 - A. `ClusterIP Service`
 - B. `NodePort Service`
 - C. `LoadBalancer Service`
 - D. `Ingress`
4. It is a best practice to start with a `ClusterIP Service` and change the Service type later if needed. (True/False)
5. Deployment resources are suitable for stateful workloads. (True/False)
6. Which kinds of workloads can you run with Ingresses?
 - A. HTTP
 - B. TCP
 - C. FTP
 - D. SMTP

-
7. Which resources would you define for dynamic volume provisioning? (Choose two)
 - A. StorageClass
 - B. PersistentVolumeClaim
 - C. PersistentVolume
 - D. StatefulSet
 8. To make your horizontal scaling more meaningful, what parameters should you use to scale your pods? (Choose three)
 - A. CPU
 - B. Memory
 - C. External metrics, such as response time
 - D. Packets per second (PPS)
 9. What are the forms of routing within an Ingress resource? (Choose two)
 - A. Simple
 - B. Path-based
 - C. Name-based
 - D. Complex

Answers

1. False. An image Deployment just scales the old ReplicaSet resource to 0.
2. A and B
3. B, C, and D
4. True
5. False. Use StatefulSet resources instead.
6. A
7. A and B
8. A, B, and C
9. B and C

7

Containers as a Service (CaaS) and Serverless Computing for Containers

In the last two chapters, we covered Kubernetes and how it helps manage your containers seamlessly. Now, let's look at other ways of automating and managing container deployments—**Containers as a Service (CaaS)** and **serverless computing for containers**. CaaS provides container-based virtualization that abstracts away all management behind the scenes and helps you manage your containers without worrying about the underlying infrastructure and orchestration.

For simple deployments and less complex applications, CaaS can be a savior. Serverless computing is a broad term that encompasses applications that can be run without us having to worry about the infrastructure behind the scenes. It has an additional benefit that you can focus purely on the application. We will discuss CaaS technologies such as **Amazon Elastic Container Service (Amazon ECS)** with **Amazon Web Services Fargate (AWS Fargate)** in detail and briefly discuss other cloud-based CaaS offerings such as **Azure Kubernetes Services (AKS)**, **Google Kubernetes Engine (GKE)**, and **Google Cloud Run**. We will then delve into the popular open source serverless CaaS solution known as **Knative**.

In this chapter, we're going to cover the following main topics:

- The need for serverless offerings
- Amazon ECS with **Elastic Compute Cloud (EC2)** and Fargate
- Other CaaS services
- Open source CaaS with Knative

Technical requirements

You will need an active AWS subscription for this chapter's exercises. AWS is the market's most popular, feature-rich cloud platform. Currently, AWS is offering a free tier for some products. You can sign up for this at <https://aws.amazon.com/free>. This chapter uses some paid services, but we will try to minimize how many we use as much as possible during the exercises.

You will also need to clone the following GitHub repository for some of the exercises:

```
https://github.com/PacktPublishing/Modern-DevOps-Practices-2e
```

Run the following command to clone the repository into your home directory. Then, cd into the ch7 directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch7
```

As the repository contains files with placeholder strings, you must replace the <your_dockerhub_user> string with your actual Docker Hub user. Use the following commands to substitute the placeholders:

```
$ find ./ -type f -exec sed -i -e \
's/<your_dockerhub_user>/<your actual docker hub user>/g' {} \;
```

So, let's get started!

The need for serverless offerings

Numerous organizations, so far, have been focusing a lot on infrastructure provisioning and management. They optimize the number of resources, machines, and infrastructure surrounding the applications they build. However, they should focus on what they do best—software development. Unless your organization wants to invest heavily in an expensive infrastructure team to do a lot of heavy lifting behind the scenes, you'd be better off concentrating on writing and building quality applications rather than focusing on where and how to run and optimize them.

Serverless offerings come as a reprieve for this problem. Instead of concentrating on how to host your infrastructure to run your applications, you can declare what you want to run, and the serverless offering manages it for you. This has become a boon for small enterprises that do not have the budget to invest heavily in infrastructure and want to get started quickly without wasting too much time standing up and maintaining infrastructure to run applications.

Serverless offerings also offer automatic placement and scaling for container and application workloads. You can spin from 0 to 100 instances in minutes, if not seconds. The best part is that you pay for what you use in some services rather than what you allocate.

This chapter will concentrate on a very popular AWS container management offering called **ECS** and AWS's container serverless offering, **AWS Fargate**. We will then briefly examine offerings from other cloud platforms and, finally, the open source container-based serverless solution known as **Knative**.

Now, let's go ahead and look at Amazon ECS.

Amazon ECS with EC2 and Fargate

Amazon ECS is a container orchestration platform that AWS offers. It is simple to use and manage, uses Docker behind the scenes, and can deploy your workloads to **Amazon EC2**, a **virtual machine (VM)**-based solution, or **AWS Fargate**, a serverless offering.

It is a highly scalable solution that deploys containers in seconds. It makes hosting, running, stopping, and starting your containers easy. Just as Kubernetes offers **pods**, ECS offers **tasks** that help you run your container workloads. A task can contain one or more containers grouped according to a logical relationship. You can also group one or more tasks into **services**. Services are similar to Kubernetes controllers, which manage tasks and can ensure that the required number of replicas of your tasks are running in the right place at the right time. ECS uses simple API calls to provide many functionalities, such as creating, updating, reading, and deleting tasks and services.

ECS also allows you to place your containers according to multiple placement strategies while keeping **high availability (HA)** and resource optimization in mind. You can tweak the placement algorithm according to your priority—cost, availability, or a mix of both. So, you can use ECS to run one-time batch workloads or long-running microservices, all using a simple-to-use API interface.

ECS architecture

Before we explore the ECS architecture, it is important to understand some common AWS terminologies to follow it. Let's look at some AWS resources:

- **AWS Regions:** An AWS Region is a geographical region where AWS provides its services. It is normally a city or a metropolitan region but can sometimes span multiple cities. It comprises multiple **Availability Zones (AZs)**. Some examples of AWS Regions are `us-east-1`, `us-west-1`, `ap-southeast-1`, `eu-central-1`, and so on.
- **AWS AZs:** AWS AZs are data centers within an AWS Region connected with low-latency, high-bandwidth networks. Most resources run within AZs. Examples of AZs are `us-east-1a`, `us-east-1b`, and so on.
- **AWS virtual private cloud (VPC):** An AWS VPC is an isolated network resource you create within AWS. You associate a dedicated private IP address range to it from which the rest of your resources, such as EC2 instances, can derive their IP addresses. An AWS VPC spans an AWS Region.
- **Subnet:** A subnet, as the name suggests, is a subnetwork within the VPC. You must subdivide the IP address ranges you provided to the VPC and associate them with subnets. Resources normally reside within subnets, and each subnet spans an AZ.

- **Route table:** An AWS route table routes traffic within the VPC subnets and to the internet. Every AWS subnet is associated with a route table through **subnet route table associations**.
- **Internet gateways:** An internet gateway allows connection to and from the internet to your AWS subnets.
- **Identity Access Management (IAM):** AWS IAM helps you control access to resources by users and other AWS resources. They help you implement **role-based access control (RBAC)** and the **principle of least privilege (PoLP)**.
- **Amazon EC2:** EC2 allows you to spin up VMs within subnets, also known as instances.
- **AWS Auto Scaling groups (ASGs):** An AWS ASG works with Amazon EC2 to provide HA and scalability to your instances. It monitors your EC2 instances and ensures that a defined number of healthy instances are always running. It also takes care of autoscaling your instances with increasing load in your machines to allow for handling more traffic. It uses the **instance profile** and **launch configuration** to decide on the properties of new EC2 instances it spins up.
- **Amazon CloudWatch:** Amazon CloudWatch is a monitoring and observability service. It allows you to collect, track, and monitor metrics, log files, and set alarms to take automated actions on specific conditions. CloudWatch helps understand application performance, health, and resource utilization.

ECS is a cloud-based regional service. When you spin up an ECS cluster, the instances span multiple AZs, where you can schedule your tasks and services using simple manifests. ECS manifests are very similar to docker-compose YAML manifests, where we specify which tasks to run and which tasks comprise a service.

You can run ECS within an existing VPC. We can schedule tasks in either Amazon EC2 or AWS Fargate.

Your ECS cluster can have one or more EC2 instances attached to it. You also have the option to attach an existing EC2 instance to a cluster by installing the ECS node agent within your EC2 instance. The agent sends information about your containers' state and tasks to the ECS scheduler. It then interacts with the container runtime to schedule containers within the node. They are similar to kubelet in the Kubernetes ecosystem. If you run your containers within EC2 instances, you pay for the number of EC2 instances you allocate to the cluster.

If you plan to use Fargate, the infrastructure is wholly abstracted from you, and you must specify the amount of CPU and memory your container is set to consume. You pay for the CPU and memory your container consumes rather than the resources you allocate to the machines.

Tip

Although you only pay for the resources you consume in Fargate, it is more expensive than running your tasks on EC2, especially when running long-running services such as a web server. A rule of thumb is to run long-running online tasks within EC2 and batch tasks with Fargate. That will give you the best cost optimization.

When we schedule a task, AWS spins up the container on a managed EC2 or Fargate server by pulling the required container image from a **container registry**. Every **task** has an **elastic network interface (ENI)** attached to it. Multiple tasks are grouped as a **service**, and the service ensures that all the required tasks run at once.

Amazon ECS uses a **task scheduler** to schedule containers on your cluster. It places your containers in an appropriate node of your cluster based on placement logic, availability, and cost requirements. The scheduler also ensures that the desired number of tasks run on the node at a given time.

The following diagram explains the ECS cluster architecture beautifully:

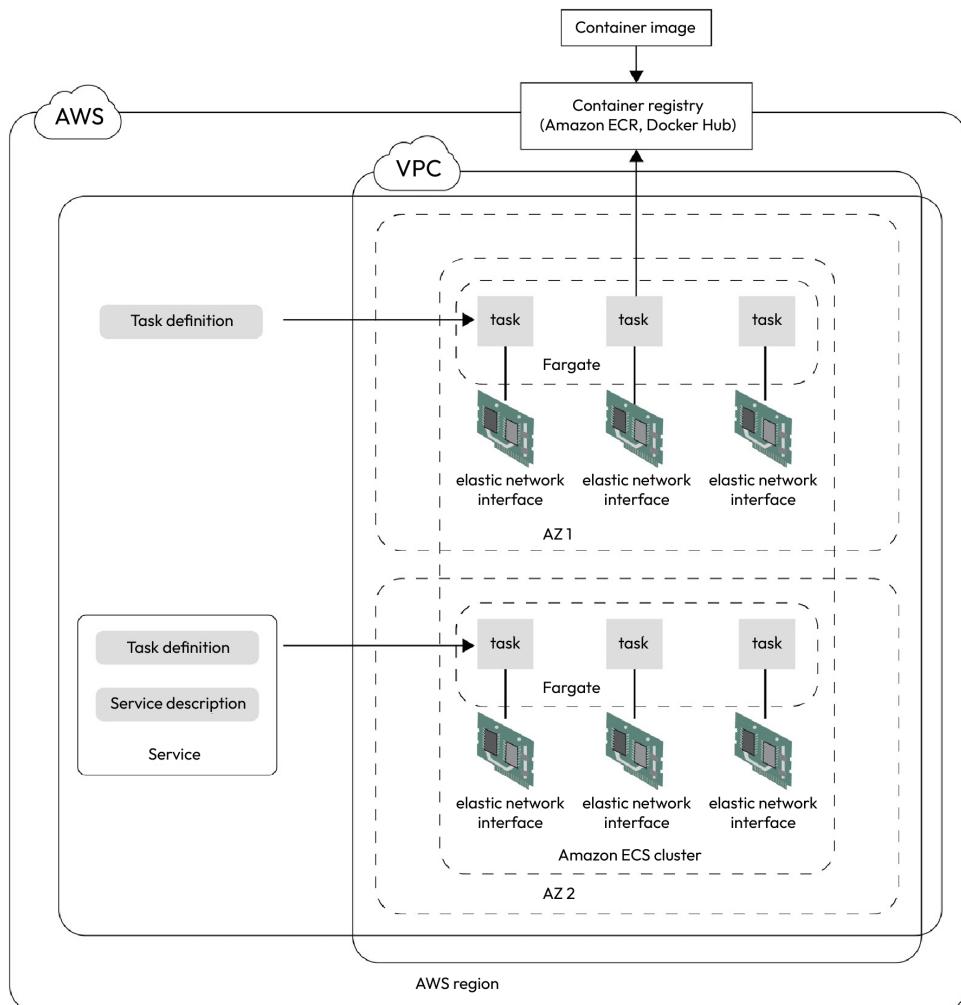


Figure 7.1 – ECS architecture

Amazon provides the ECS **command-line interface (CLI)** for interacting with the ECS cluster. It is a simple command-line tool that you can use to administer an ECS cluster and create and manage tasks and services on the ECS cluster.

Now, let's go ahead and install the ECS CLI.

Installing the AWS and ECS CLIs

The AWS CLI is available as a deb package within the public apt repositories. To install it, run the following commands:

```
$ sudo apt update && sudo apt install awscli -y  
$ aws --version  
aws-cli/1.22.34 Python/3.10.6 Linux/5.19.0-1028-aws botocore/1.23.34
```

Installing the ECS CLI in the Linux ecosystem is simple. We need to download the binary and move to the system path using the following command:

```
$ sudo curl -Lo /usr/local/bin/ecs-cli \  
https://amazon-ecs-cli.s3.amazonaws.com/ecs-cli-linux-amd64-latest  
$ sudo chmod +x /usr/local/bin/ecs-cli
```

Run the following command to check whether `ecs-cli` has been installed correctly:

```
$ ecs-cli --version  
ecs-cli version 1.21.0 (bb0b8f0)
```

As we can see, `ecs-cli` has been successfully installed on our system.

The next step is to allow `ecs-cli` to connect with your AWS API. You need to export your AWS CLI environment variables for this. Run the following commands to do so:

```
$ export AWS_SECRET_ACCESS_KEY=...  
$ export AWS_ACCESS_KEY_ID=...  
$ export AWS_DEFAULT_REGION=...
```

Once we've set the environment variables, `ecs-cli` will use them to authenticate with the AWS API. In the next section, we'll spin up an ECS cluster using the ECS CLI.

Spinning up an ECS cluster

We can use the ECS CLI commands to spin up an ECS cluster. You can run your containers in EC2 and Fargate, so first, we will create a cluster that runs EC2 instances. Then, we will add Fargate tasks within the cluster.

To connect with your EC2 instances, you need to generate a key pair within AWS. To do so, run the following command:

```
$ aws ec2 create-key-pair --key-name ecs-keypair
```

The output of this command will provide the key pair in a JSON file. Extract the JSON file's key material and save that in a separate file called `ecs-keypair.pem`. Remember to replace the \n characters with a new line when you save the file.

Once we've generated the key pair, we can use the following command to create an ECS cluster using the ECS CLI:

```
$ ecs-cli up --keypair ecs-keypair --instance-type t2.micro \
--size 2 --cluster cluster-1 --capability-iam
INFO[0002] Using recommended Amazon Linux 2 AMI with ECS Agent 1.72.0 and Docker version
20.10.23
INFO[0003] Created cluster cluster=cluster-1 region=us-east-1
INFO[0004] Waiting for your cluster resources to be created...
INFO[0130] Cloudformation stack status stackStatus=CREATE_IN_PROGRESS
VPC created: vpc-0448321d209bf75e2
Security Group created: sg-0e30839477f1c9881
Subnet created: subnet-02200afa6716866fa
Subnet created: subnet-099582f6b0d04e419
Cluster creation succeeded.
```

When we issue this command, in the background, AWS spins up a stack of resources using CloudFormation. CloudFormation is AWS's **Infrastructure-as-Code (IaC)** solution that helps you deploy infrastructure on AWS through reusable templates. The CloudFormation template consists of several resources such as a VPC, a security group, a subnet within the VPC, a route table, a route, a subnet route table association, an internet gateway, an IAM role, an instance profile, a launch configuration, an ASG, a VPC gateway attachment, and the cluster itself. The ASG contains two EC2 instances running and serving the cluster. Keep a copy of the output; we will need the details later during the exercises.

Now that our cluster is up, we will spin up our first task.

Creating task definitions

ECS tasks are similar to Kubernetes pods. They are the basic building blocks of ECS and comprise one or more related containers. Task definitions are the blueprints for ECS tasks and define what the ECS task should look like. They are very similar to `docker-compose` files and are written in YAML format. ECS also uses all versions of `docker-compose` to allow us to define tasks. They help you define containers and their images, resource requirements, where they should run (EC2 or Fargate), volume and port mappings, and other networking requirements.

Tip

Using the `docker-compose` manifest to spin up tasks and services is a great idea, as it will help you align your configuration with an open standard.

A task is a finite process and only runs once. Even if it's a long-running process, such as a web server, the task still runs once as it waits for the long-running process to end (which runs indefinitely in theory). The task's life cycle follows the **Pending** -> **Running** -> **Stopped** states. So, when you schedule your task, the task enters the **Pending** state, attempting to pull the image from the container registry. Then, it tries to start the container. Once the container has started, it enters the **Running** state. When the container has completed executing or errored out, it ends up in the **Stopped** state. A container with startup errors directly transitions from the **Pending** state to the **Stopped** state.

Now, let's go ahead and deploy an `nginx` web server task within the ECS cluster we just created.

To access the resources for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch7/ECS/tasks/EC2/
```

We'll use `docker-compose` task definitions here. So, let's start by defining the following `docker-compose.yml` file:

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "80:80"
    logging:
      driver: awslogs
      options:
        awslogs-group: /aws/webserver
        awslogs-region: us-east-1
        awslogs-stream-prefix: ecs
```

The YAML file defines a web container with an `nginx` image with host port 80 mapped to container port 80. It uses the `awslogs` logging driver, which streams logs into Amazon CloudWatch. It will stream the logs to the `/aws/webserver` log group in the `us-east-1` region with the `ecs` stream prefix.

The task definition also includes the resource definition—that is, the amount of resources we want to reserve for our task. Therefore, we will have to define the following `ecs-params.yaml` file:

```
version: 1
task_definition:
  services:
    web:
      cpu_shares: 100
      mem_limit: 524288000
```

This YAML file defines `cpu_shares` in millicores and `mem_limit` in bytes for the container we plan to fire. Now, let's look at scheduling this task as an EC2 task.

Scheduling EC2 tasks on ECS

Let's use `ecs-cli` to apply the configuration and schedule our task using the following command:

```
$ ecs-cli compose up --create-log-groups --cluster cluster-1 --launch-type EC2
```

Now that the task has been scheduled and the container is running, let's list all the tasks to get the container's details and find out where it is running. To do so, run the following command:

```
$ ecs-cli ps --cluster cluster-1
Name           State    Ports          TaskDefinition
cluster-1/feelcf28/web  RUNNING  34.237.218.7:80->80  EC2:1
```

As we can see, the web container is running on `cluster-1` on `34.237.218.7:80`. Now, use the following command to curl this endpoint to see what we get:

```
$ curl 34.237.218.7:80
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

Here, we get the default `nginx` home page! We've successfully scheduled a container on ECS using the EC2 launch type. You might want to duplicate this task to handle more traffic. This is known as horizontal scaling. We'll see how in the next section.

Scaling tasks

We can easily scale tasks using `ecs-cli`. Use the following command to scale the tasks to 2:

```
$ ecs-cli compose scale 2 --cluster cluster-1 --launch-type EC2
```

Now, use the following command to check whether two containers are running on the cluster:

```
$ ecs-cli ps --cluster cluster-1
Name           State    Ports          TaskDefinition
cluster-1/b43bdec7/web  RUNNING  54.90.208.183:80->80  EC2:1
cluster-1/feelcf28/web  RUNNING  34.237.218.7:80->80  EC2:1
```

As we can see, two containers are running on the cluster. Now, let's query CloudWatch to get the logs of the containers.

Querying container logs from CloudWatch

To query logs from CloudWatch, we must list the log streams using the following command:

```
$ aws logs describe-log-streams --log-group-name /aws/webserver \
--log-stream-name-prefix ecs | grep logStreamName
"logStreamName": "ecs/web/b43bdec7",
"logStreamName": "ecs/web/feelcf28",
```

As we can see, there are two log streams for this—one for each task. `logStreamName` follows the convention `<log_stream_prefix>/<task_name>/<task_id>`. So, to get the logs for `ecs/b43bdec7/web`, run the following command:

```
$ aws logs get-log-events --log-group-name /aws/webserver \
--log-stream ecs/web/b43bdec7
```

Here, you will see a stream of logs in JSON format in the response. Now, let's look at how we can stop running tasks.

Stopping tasks

`ecs-cli` uses the friendly `docker-compose` syntax for everything. Use the following command to stop the tasks in the cluster:

```
$ ecs-cli compose down --cluster cluster-1
```

Let's list the containers to see whether the tasks have stopped by using the following command:

```
$ ecs-cli ps --cluster cluster-1
INFO[0001] Stopping container... container=cluster-1/b43bdec7/web
INFO[0001] Stopping container... container=cluster-1/feelcf28/web
INFO[0008] Stopped container... container=cluster-1/b43bdec7/web
desiredStatus=STOPPED lastStatus=STOPPED taskDefinition="EC2:1"
INFO[0008] Stopped container... container=cluster-1/feelcf28/web
desiredStatus=STOPPED lastStatus=STOPPED taskDefinition="EC2:1"
```

As we can see, both containers have stopped.

Running tasks on EC2 is not a serverless way of doing things. You still have to provision and manage the EC2 instances, and although ECS manages workloads on the cluster, you still have to pay for the amount of resources you've provisioned in the form of EC2 instances. AWS offers Fargate as a serverless solution where you pay per resource consumption. Let's look at how we can create the same task as a Fargate task.

Scheduling Fargate tasks on ECS

Scheduling tasks on Fargate is very similar to EC2. Here, we need to specify the `launch-type` value as `FARGATE`.

To schedule the same task on Fargate, run the following command:

```
$ ecs-cli compose up --create-log-groups --cluster cluster-1 --launch-type FARGATE
FATA[0001] ClientException: Fargate only supports network mode 'awsvpc'.
```

Oops! We have a problem! Well, it's complaining about the network type. For a Fargate task, we must supply the network type as awsvpc instead of the default bridge network. The awsvpc network is an overlay network that implements the **Container Network Interface (CNI)**. To understand more about Docker networking, please refer to *Chapter 1, The Modern Way of DevOps*. For now, let's go ahead and configure the awsvpc network type. But before that, the Fargate task requires a few configurations.

To access the resources for this section, cd into the following directory:

```
$ cd ~/modern-devops/ch7/ECS/tasks/FARGATE
```

First, we'll have to assume a task execution role for the ECS agent to authenticate with the AWS API and interact with Fargate.

To do so, create the following `task-execution-assume-role.json` file:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Then, use the following command to assume the task execution role:

```
$ aws iam --region us-east-1 create-role --role-name ecsTaskExecutionRole \
--assume-role-policy-document file://task-execution-assume-role.json
```

ECS provides a default role policy called `AmazonECSTaskExecutionRolePolicy`, which contains various permissions that help you interact with CloudWatch and **Elastic Container Registry (ECR)**. The following JSON code outlines the permission that the policy has:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/*"
    }
  ]
}
```

```

        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "logs>CreateLogStream",
        "logs:PutLogEvents"
    ],
    "Resource": "*"
}
]
}

```

We have to assign this role policy to the `ecsTaskExecution` role we assumed previously by using the following command:

```
$ aws iam attach-role-policy \
--policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy \
--role-name ecsTaskExecutionRole
```

Once we've assigned the policy to the `ecsTaskExecution` role, we need to source the ID of both subnets and the security group of the ECS cluster when we created it. You can find those details in the command-line output from when we created the cluster. We will use these details in the following `ecs-params.yml` file:

```

version: 1
task_definition:
  task_execution_role: ecsTaskExecutionRole
  ecs_network_mode: awsvpc
  task_size:
    mem_limit: 0.5GB
    cpu_limit: 256
run_params:
  network_configuration:
    awsvpc_configuration:
      subnets:
        - "subnet-088b52c91a6f40fd7"
        - "subnet-032cd63290da67271"
      security_groups:
        - "sg-097206175813aa7e7"
      assign_public_ip: ENABLED

```

The `ecs-params.yml` file consists of `task_execution_role`, which we created, and `ecs_network_mode` set to `awsvpc`, as Fargate requires. We've defined `task_size` to have 0.5GB of memory and 256 millicores of CPU. So, since Fargate is a serverless solution, we only pay for the CPU cores and memory we consume. The `run_params` section consists of `network_configuration`, which contains `awsvpc_configuration`. Here, we specify both subnets created when we created the ECS cluster. We must also specify `security_groups`, which we created with the ECS cluster.

Note

Use the subnets and security groups of your ECS cluster instead of copying the ones in this example.

Now that we're ready to fire the task on Fargate, let's run the following command:

```
$ ecs-cli compose up --create-log-groups --cluster cluster-1 --launch-type FARGATE
```

Now, let's check whether the task is running successfully by using the following command:

```
$ ecs-cli ps --cluster cluster-1
Name           State    Ports          TaskDefinition
cluster-1/8717a149/web  RUNNING  3.80.173.230:80  FARGATE:1
```

As we can see, the task is running on 3.80.173.230:80 as a Fargate task. Let's curl this URL to see whether we get a response by using the following command:

```
$ curl 3.80.173.230:80
<html>
<head>
<title>Welcome to nginx!</title>
...
</body>
</html>
```

As we can see, we get the default nginx home page.

Now, let's go ahead and delete the task we created by using the following command:

```
$ ecs-cli compose down --cluster cluster-1
```

As we already know, tasks have a set life cycle, and once they stop, they stop. You cannot start the same task again. Therefore, we must create a **service** to ensure that a certain number of tasks are always running. We'll create a service in the next section.

Scheduling services on ECS

ECS services are similar to Kubernetes **ReplicaSets**. They ensure that a certain number of tasks are always running at a particular time. To schedule a service, we can use the `ecs-cli` command line.

Tip

Always use services for applications that are long-running, such as web servers. For batch jobs, always use tasks, as we don't want to recreate the job after it ends.

To run the nginx web server as a service, we can use the following command:

```
$ ecs-cli compose service up --create-log-groups \
--cluster cluster-1 --launch-type FARGATE
INFO[0001] Using ECS task definition TaskDefinition="FARGATE:1"
INFO[0002] Auto-enabling ECS Managed Tags
INFO[0013] (service FARGATE) has started 1 tasks: (task 9b48084d). timestamp="2023-07-03 11:24:42 +0000 UTC"
INFO[0029] Service status desiredCount=1 runningCount=1 serviceName=FARGATE
INFO[0029] (service FARGATE) has reached a steady state. timestamp="2023-07-03 11:25:00 +0000 UTC"
INFO[0029] (service FARGATE) (deployment ecs-svc/94284856) deployment completed. timestamp="2023-07-03 11:25:00 UTC"
INFO[0029] ECS Service has reached a stable state desiredCount=1 runningCount=1 serviceName=FARGATE
INFO[0029] Created an ECS service service=FARGATE taskDefinition="FARGATE:1"
```

Looking at the logs, we can see that the service is trying to ensure that the task's desired count matches the task's running count. If your task is deleted, ECS will replace it with a new one.

Let's list the tasks and see what we get by using the following command:

```
$ ecs-cli ps --cluster cluster-1
Name           State    Ports          TaskDefinition
cluster-1/9b48084d/web  RUNNING  18.234.123.71:80  FARGATE:1
```

As we can see, the service has created a new task that is running on 18.234.123.71:80. Let's try to access this URL using the following command:

```
$ curl 18.234.123.71
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

We get the default nginx home page in the response. Now, let's try to browse the logs of the task.

Browsing container logs using the ECS CLI

Apart from using Amazon CloudWatch, you can also use the friendly ECS CLI to do this, irrespective of where your logs are stored. This helps us see everything from a single pane of glass.

Run the following command to do so:

```
$ ecs-cli logs --task-id 9b48084d --cluster cluster-1
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform
configuration
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/
default.conf
```

```
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/07/03 11:24:57 [notice] 1#1: nginx/1.25.1
2023/07/03 11:24:57 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2023/07/03 11:24:57 [notice] 1#1: OS: Linux 5.10.184-175.731.amzn2.x86_64
2023/07/03 11:24:57 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 65535:65535
2023/07/03 11:24:57 [notice] 1#1: start worker processes
2023/07/03 11:24:57 [notice] 1#1: start worker process 29
2023/07/03 11:24:57 [notice] 1#1: start worker process 30
13.232.8.130 - - [03/Jul/2023:11:30:38 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.81.0"
"-"
```

As we can see, we can browse the logs for the particular task this service is running. Now, let's go ahead and delete the service.

Deleting an ECS service

To delete the service, run the following command:

```
$ ecs-cli compose service down --cluster cluster-1
INFO[0001] Deleted ECS service service=FARGATE
INFO[0001] Service status desiredCount=0 runningCount=1 serviceName=FARGATE
INFO[0006] Service status desiredCount=0 runningCount=0 serviceName=FARGATE
INFO[0006] (service FARGATE) has stopped 1 running tasks: (task
9b48084d11cf49be85141fd9bfe9e1c3). timestamp="2023-07-03 11:34:10 +0000 UTC"
INFO[0006] ECS Service has reached a stable state desiredCount=0 runningCount=0
serviceName=FARGATE
```

As we can see, the service has been deleted.

Note that even if we create multiple instances of tasks, they run on different IP addresses and can be accessed separately. However, tasks need to be load-balanced, and we need to provide a single endpoint. Let's look at a solution we can use to manage this.

Load balancing containers running on ECS

Load balancing is an essential functionality of multi-instance applications. They help us serve the application on a single endpoint. Therefore, you can have multiple instances of your applications running simultaneously, and the end user doesn't need to worry about which instance they're calling. AWS provides two main load-balancing solutions—**Layer 4** with the **Network Load Balancer (NLB)** and **Layer 7** with the **Application Load Balancer (ALB)**.

Tip

While both load balancers have their use cases, a Layer 7 load balancer provides a significant advantage for HTTP-based applications. It offers advanced traffic management, such as path-based and host-based routing.

So, let's go ahead and create an ALB to frontend our tasks using the following command:

```
$ aws elbv2 create-load-balancer --name ecs-alb --subnets <SUBNET-1> <SUBNET-2> \
--security-groups <SECURITY_GROUP_ID> --region us-east-1
```

The output of the preceding command contains values for LoadBalancerARN and DNSName. We will need to use them in the subsequent steps, so keep a copy of the output safe.

The next step will be to create a **target group**. The target group defines the group of tasks and the port they will be listening to, and the load balancer will forward traffic to it. Use the following command to define a target group:

```
$ aws elbv2 create-target-group --name target-group --protocol HTTP \
--port 80 --target-type ip --vpc-id <VPC_ID> --region us-east-1
```

You will get the targetGroupARN value in the response. Keep it safe, as we will need it in the next step.

Next, we will need a **listener** running on the load balancer. This should forward traffic from the load balancer to the target group. Use the following command to do so:

```
$ aws elbv2 create-listener --load-balancer-arn <LOAD_BALANCER_ARN> \
--protocol HTTP --port 80 \
--default-actions Type=forward,TargetGroupArn=<TARGET_GROUP_ARN> \
--region us-east-1
```

You will get the listenerARN value in response to this command. Please keep that handy; we will need it in the next step.

Now that we've defined the load balancer, we need to run `ecs-cli compose service` up to deploy our service. We will also provide the target group as a parameter to associate our service with the load balancer.

To access the resources for this section, `cd` into the following directory:

```
$ cd ~/modern-devops/ch7/ECS/loadbalancing/
```

Run the following command to do so:

```
$ ecs-cli compose service up --create-log-groups --cluster cluster-1 \
--launch-type FARGATE --target-group-arn <TARGET_GROUP_ARN> \
--container-name web --container-port 80
```

Now that the service and our task are running on Fargate, we can scale our service to three desired tasks. To do so, run the following command:

```
$ ecs-cli compose service scale 3 --cluster cluster-1
```

Since our service has scaled to three tasks, let's go ahead and hit the load balancer DNS endpoint we captured in the first step. This should provide us with the default nginx response. Run the following command to do so:

```
$ curl ecs-alb-1660189891.us-east-1.elb.amazonaws.com
<html>
<head>
<title>Welcome to nginx!</title>
...
</html>
```

As we can see, we get a default nginx response from the load balancer. This shows that load balancing is working well!

ECS provides a host of other features, such as horizontal autoscaling, customizable task placement algorithms, and others, but they are beyond the scope of this book. Please read the ECS documentation to learn more about other aspects of the tool. Now, let's look at other popular CaaS products available on the market.

Other CaaS services

Amazon ECS provides a versatile way of managing your container workloads. It works great when you have a smaller, simpler architecture and don't want to add the additional overhead of using a complex container orchestration engine such as Kubernetes.

Tip

ECS is an excellent tool choice if you run exclusively on AWS and don't have a future multi-cloud or hybrid-cloud strategy. Fargate makes deploying and running your containers easier without worrying about the infrastructure behind the scenes.

ECS is tightly coupled with AWS and its architecture. To solve this problem, we can use managed services within AWS, such as **Elastic Kubernetes Service (EKS)**. It offers the Kubernetes API to schedule your workloads. This makes managing containers even more versatile as you can easily spin up a Kubernetes cluster and use a standard, open source solution that you can install and run anywhere you like. This does not tie you to a particular vendor. However, EKS is slightly more expensive than ECS and adds a \$0.10 per hour cluster management charge. That is nothing in comparison to the benefits you get out of it.

If you aren't running on AWS, there are options from other providers too. The next of the big three cloud providers is Azure, which offers **Azure Kubernetes Service (AKS)**, a managed Kubernetes solution that can help you get started in minutes. AKS provides a fully managed solution with event-driven elastic provisioning for worker nodes as and when required. It also integrates nicely with **Azure DevOps**, giving you a faster **end-to-end (E2E)** development experience. As with AWS, Azure also charges \$0.10 per hour for cluster management.

Google Kubernetes Engine (GKE) is one of the most robust Kubernetes platforms. Since the Kubernetes project came from Google and is the largest contributor to this project in the open source community, GKE is generally quicker to roll out newer versions and is the first to release security patches into the solution. Also, it is one of the most feature-rich with customizable solutions and offers several plugins as a cluster configuration. Therefore, you can choose what to install on Bootstrap and further harden your cluster. However, all these come at a cost, as GKE charges a *\$0.10* cluster management charge per hour, just like AWS and Azure.

You can use **Google Cloud Run** if you don't want to use Kubernetes if your architecture is not complicated, and there are only a few containers to manage. Google Cloud Run is a serverless CaaS solution built on the open source **Knative** project. It helps you run your containers without any vendor lock-in. Since it is serverless, you only pay for the number of containers you use and their resource utilization. It is a fully scalable and well-integrated solution with Google Cloud's DevOps and monitoring solutions such as **Cloud Code**, **Cloud Build**, **Cloud Monitoring**, and **Cloud Logging**. The best part is that it is comparable to AWS Fargate and abstracts all infrastructure behind the scenes. So, it's a minimal Ops or NoOps solution.

Now that we've mentioned Knative as an open source CaaS solution, let's discuss it in more detail.

Open source CaaS with Knative

As we've seen, several vendor-specific CaaS services are available on the market. Still, the problem with most of them is that they are tied up to a single cloud provider. Our container deployment specification then becomes vendor-specific and results in vendor lock-in. As modern DevOps engineers, we must ensure that the proposed solution best fits the architecture's needs, and avoiding vendor lock-in is one of the most important requirements.

However, Kubernetes in itself is not serverless. You must have infrastructure defined, and long-running services should have at least a single instance running at a particular time. This makes managing microservices applications a pain and resource-intensive.

But wait! We said that microservices help optimize infrastructure consumption. Yes—that's correct, but they do so within the container space. Imagine that you have a shared cluster of VMs where parts of the application scale with traffic, and each part of the application has its peaks and troughs. Doing this will save a lot of infrastructure by performing this simple multi-tenancy.

However, it also means that you must have at least one instance of each microservice running every time—even if there is zero traffic! Well, that's not the best utilization we have. How about creating instances when you get the first hit and not having any when you don't have traffic? This would save a lot of resources, especially when things are silent. You can have hundreds of microservices making up the application that would not have any instances during an idle period. If you combine it with a managed service that runs Kubernetes and then autoscale your VM instances with traffic, you can have minimal instances during the silent period.

There have been attempts within the open source and cloud-native space to develop an open source, vendor-agnostic, serverless framework for containers. We have Knative for this, which the **Cloud Native Computing Foundation (CNCF)** has adopted.

Tip

The Cloud Run service uses Knative behind the scenes. So, if you use Google Cloud, you can use Cloud Run to use a fully managed serverless offering.

To understand how Knative works, let's look at the Knative architecture.

Knative architecture

The Knative project combines elements of existing CNCF projects such as Kubernetes, **Istio**, **Prometheus**, and **Grafana** and eventing engines such as **Kafka** and **Google Pub/Sub**. Knative runs as a Kubernetes operator using Kubernetes **Custom Resource Definitions (CRDs)**, which help operators administer Knative using the `kubectl` command line. Knative provides its API for developers, which the `kn` command-line utility can use. The users are provided access through Istio, which, with its traffic management features, is a crucial component of Knative. The following diagram describes this graphically:

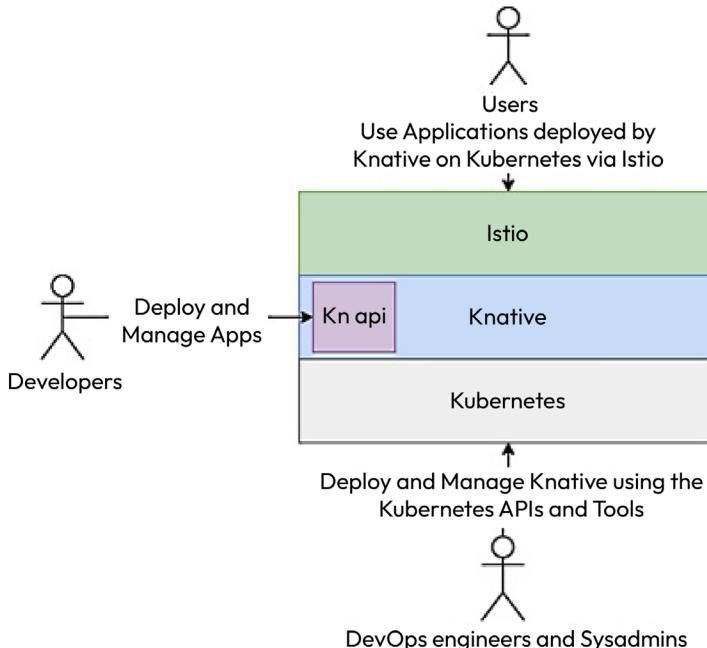


Figure 7.2 – Knative architecture

Knative consists of two main modules—**serving** and **eventing**. While the **serving** module helps us maintain stateless applications using HTTP/S endpoints, the **eventing** module integrates with eventing engines such as Kafka and Google Pub/Sub. As we've discussed mostly HTTP/S traffic, we will scope our discussion to Knative serving for this book.

Knative maintains serving pods, which help route traffic within workload pods and act as proxies using the **Istio Ingress Gateway** component. It provides a virtual endpoint for your service and listens to it. When it discovers a hit on the endpoint, it creates the required Kubernetes components to serve that traffic. Therefore, Knative has the functionality to scale from zero workload pods as it will spin up a pod when it receives traffic for it. The following diagram shows how:

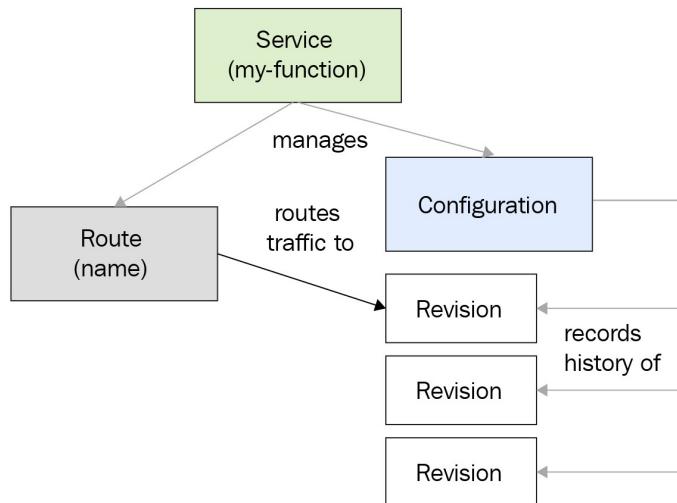


Figure 7.3 – Knative serving architecture

Knative endpoints are made up of three basic parts—`<app-name>`, `<namespace>`, and `<custom-domain>`. While name and namespace are similar to Kubernetes Services, custom-domain is defined by us. It can be a legitimate domain for your organization or a **MagicDNS** solution, such as **sslip.io**, which we will use in our hands-on exercises. If you are using your organization domain, you must create your DNS configuration to resolve the domain to the Istio Ingress Gateway IP addresses.

Now, let's go ahead and install Knative.

For the exercises, we will use GKE. Since GKE is a highly robust Kubernetes cluster, it is a great choice for integrating with Knative. As mentioned previously, Google Cloud provides a free trial of \$300 for 90 days. You can sign up at <https://cloud.google.com/free> if you've not done so already.

Spinning up GKE

Once you've signed up and are on your console, you can open the Google Cloud Shell CLI to run the following commands.

You need to enable the GKE API first using the following command:

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

To create a two-node autoscaling GKE cluster that scales from 1 node to 5 nodes, run the following command:

```
$ gcloud container clusters create cluster-1 --num-nodes 2 \
--enable-autoscaling --min-nodes 1 --max-nodes 5 --zone us-central1-a
```

And that's it! The cluster is up and running.

You will also need to clone the following GitHub repository for some of the exercises:

<https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>

Run the following command to clone the repository into your home directory. Then, cd into the ch7 directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
```

Now that the cluster is up and running, let's go ahead and install Knative.

Installing Knative

We will install the CRDs that define Knative resources as Kubernetes API resources.

To access the resources for this section, cd into the following directory:

```
$ cd ~/modern-devops/ch7/knative/
```

Run the following command to install the CRDs:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-crds.yaml
```

As we can see, Kubernetes has installed some CRDs. Next, we must install the core components of the Knative serving module. Use the following command to do so:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-core.yaml
```

Now that the core serving components have been installed, the next step is installing Istio within the Kubernetes cluster. To do so, run the following commands:

```
$ curl -L https://istio.io/downloadIstio | sh -
$ sudo mv istio-* /bin/istioctl /usr/local/bin
$ istioctl install --set profile=demo -y
```

Now that Istio has been installed, we will wait for the Istio Ingress Gateway component to be assigned an external IP address. Run the following command to check this until you get an external IP in the response:

```
$ kubectl -n istio-system get service istio-ingressgateway
NAME           TYPE      EXTERNAL-IP     PORT(S)
istio-ingressgateway   LoadBalancer   35.226.198.46  15021,80,443
```

As we can see, we've been assigned an external IP—`35.226.198.46`. We will use this IP for the rest of this exercise.

Now, we will install the Knative Istio controller by using the following command:

```
$ kubectl apply -f \
https://github.com/knative/net-istio/releases/download/knative-v1.10.1/net-istio.yaml
```

Now that the controller has been installed, we must configure the DNS so that Knative can provide custom endpoints. To do so, we can use the MagicDNS solution known as `sslip.io`, which you can use for experimentation. The MagicDNS solution resolves any endpoint to the IP address present in the subdomain. For example, `35.226.198.46.sslip.io` resolves to `35.226.198.46`.

Note

Do not use MagicDNS in production. It is an experimental DNS service and should only be used for evaluating Knative.

Run the following command to configure the DNS:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2\
/serving-default-domain.yaml
```

As you can see, it provides a batch job that gets fired whenever there is a DNS request.

Now, let's install the **HorizontalPodAutoscaler (HPA)** add-on to automatically help us autoscale pods on the cluster with traffic. To do so, run the following command:

```
$ kubectl apply -f \
https://github.com/knative/serving/releases/download/knative-v1.10.2/serving-hpa.yaml
```

That completes our Knative installation.

Now, we need to install and configure the `kn` command-line utility. Use the following commands to do so:

```
$ sudo curl -Lo /usr/local/bin/kn \
https://github.com/knative/client/releases/download/knative-v1.10.0/kn-linux-amd64
$ sudo chmod +x /usr/local/bin/kn
```

In the next section, we'll deploy our first application on Knative.

Deploying a Python Flask application on Knative

To understand Knative, let's try to build and deploy a Flask application that outputs the current timestamp in the response. Let's start by building the app.

Building the Python Flask app

We will have to create a few files to build such an app.

The `app.py` file looks like this:

```
import os
import datetime
from flask import Flask
app = Flask(__name__)
@app.route('/')
def current_time():
    ct = datetime.datetime.now()
    return 'The current time is : {}!\n'.format(ct)
if __name__ == "__main__":
    app.run(debug=True,host='0.0.0.0')
```

We will need the following Dockerfile to build this application:

```
FROM python:3.7-slim
ENV PYTHONUNBUFFERED True
ENV APP_HOME /app
WORKDIR $APP_HOME
COPY . .
RUN pip install Flask gunicorn
CMD exec gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0 app:app
```

Now, let's go ahead and build the Docker container using the following command:

```
$ docker build -t <your_dockerhub_user>/py-time .
```

Now that the image is ready, let's push it to Docker Hub by using the following command:

```
$ docker push <your_dockerhub_user>/py-time
```

As we've successfully pushed the image, we can run this on Knative.

Deploying the Python Flask app on Knative

We can use the `kn` command line or create a manifest file to deploy the app. Use the following command to deploy the application:

```
$ kn service create py-time --image <your_dockerhub_user>/py-time
Creating service 'py-time' in namespace 'default':
  9.412s Configuration "py-time" is waiting for a Revision to become ready.
  9.652s Ingress has not yet been reconciled.
  9.847s Ready to serve.
Service 'py-time' created to latest revision 'py-time-00001' is available at URL:
http://py-time.default.35.226.198.46.sslip.io
```

As we can see, Knative has deployed the app and provided a custom endpoint. Let's `curl` the endpoint to see what we get:

```
$ curl http://py-time.default.35.226.198.46.sslip.io
The current time is : 2023-07-03 13:30:20.804790!
```

We get the current time in the response. As we already know, Knative should detect whether there is no traffic coming into the pod and delete it. Let's watch the pods for some time and see what happens:

```
$ kubectl get pod -w
NAME                      READY   STATUS    RESTARTS   AGE
py-time-00001-deployment-jqrk 2/2     Running   0          5s
py-time-00001-deployment-jqrk 2/2     Terminating   0          64s
```

As we can see, just after 1 minute of inactivity, Knative starts terminating the pod. Now, that's what we mean by scaling from zero.

To delete the service permanently, we can use the following command:

```
$ kn service delete py-time
```

We've just looked at the imperative way of deploying and managing the application. But what if we want to declare the configuration as we did previously? We can create a CRD manifest with the `Service` resource provided by `apiVersion=serving.knative.dev/v1`.

We will create the following manifest file, called `py-time-deploy.yaml`, for this:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: py-time
```

```
spec:  
  template:  
    spec:  
      containers:  
        - image: <your_dockerhub_user>/py-time
```

As we've created this file, we will use the `kubectl` CLI to apply it. It makes deployment consistent with Kubernetes.

Note

Though it is a `service` resource, don't confuse this with the typical Kubernetes `Service` resource. It is a custom resource provided by `apiVersion serving.knative.dev/v1`. That is why `apiVersion` is very important.

Let's go ahead and run the following command to do so:

```
$ kubectl apply -f py-time-deploy.yaml  
service.serving.knative.dev/py-time created
```

With that, the service has been created. To get the service's endpoint, we will have to query the `ksvc` resource using `kubectl`. Run the following command to do so:

```
$ kubectl get ksvc py-time  
NAME      URL  
py-time   http://py-time.default.35.226.198.46.sslip.io
```

The URL is the custom endpoint we have to target. Let's `curl` the custom endpoint using the following command:

```
$ curl http://py-time.default.35.226.198.46.sslip.io  
The current time is : 2023-07-03 13:30:23.345223!
```

We get the same response this time as well! So, if you want to keep using `kubectl` for managing Knative, you can easily do so.

Knative helps scale applications based on the load it receives—automatic horizontal scaling. Let's run load testing on our application to see that in action.

Load testing your app on Knative

We will use the `hey` utility to perform load testing. Since your application has already been deployed, run the following command to do the load test:

```
$ hey -z 30s -c 500 http://py-time.default.35.226.198.46.sslip.io
```

Once the command has executed, run the following command to get the currently running instances of the `py-time` pods:

```
$ kubectl get pod
NAME                      READY   STATUS    RESTARTS   AGE
py-time-00001-deployment-52vjv 2/2   Running   0          44s
py-time-00001-deployment-bhhvm 2/2   Running   0          44s
py-time-00001-deployment-h6qr5  2/2   Running   0          42s
py-time-00001-deployment-h92jp  2/2   Running   0          40s
py-time-00001-deployment-p27gl  2/2   Running   0          88s
py-time-00001-deployment-tdwrh  2/2   Running   0          38s
py-time-00001-deployment-zsgcg 2/2   Running   0          42s
```

As we can see, Knative has created seven instances of the `py-time` pod. This is horizontal autoscaling in action.

Now, let's look at the cluster nodes by running the following command:

```
$ kubectl get nodes
NAME                  STATUS   AGE
gke-cluster-1-default-pool-353b3ed4-js71  Ready   3m17s
gke-cluster-1-default-pool-353b3ed4-mx83  Ready   106m
gke-cluster-1-default-pool-353b3ed4-vf7q  Ready   106m
```

As we can see, GKE has created another node in the node pool because of the extra burst of traffic it received. This is phenomenal, as we have the Kubernetes API to do what we want. We have automatically horizontally autoscaled our pods. We have also automatically horizontally autoscaled our cluster worker nodes. This means we have a fully automated solution for running containers without worrying about the management nuances! That is open source serverless in action for you!

Summary

This chapter covered CaaS and serverless CaaS services. These help us manage container applications with ease without worrying about the underlying infrastructure and managing them. We used Amazon's ECS as an example and deep-dived into it. Then, we briefly discussed other solutions that are available on the market.

Finally, we looked at Knative, an open source serverless solution for containers that run on top of Kubernetes and use many other open source CNCF projects.

In the next chapter, we will delve into IaC with Terraform.

Questions

1. ECS allows us to deploy to which of the following? (Choose two)
 - A. EC2
 - B. AWS Lambda

- C. Fargate
 - D. Amazon Lightsail
2. ECS uses Kubernetes in the background. (True/False)
 3. We should always use services in ECS instead of tasks for batch jobs. (True/False)
 4. We should always use Fargate for batch jobs as it runs for a short period, and we only pay for the resources that are consumed during that time. (True/False)
 5. Which of the following are CaaS services that implement the Kubernetes API? (Choose three)
 - A. GKE
 - B. AKS
 - C. EKS
 - D. ECS
 6. Google Cloud Run is a serverless offering that uses Knative behind the scenes. (True/False)
 7. Which one of the following is offered as a Knative module? (Choose two)
 - A. Serving
 - B. Eventing
 - C. Computing
 - D. Containers

Answers

1. A, C
2. False
3. False
4. True
5. A, B, C
6. True
7. A, B

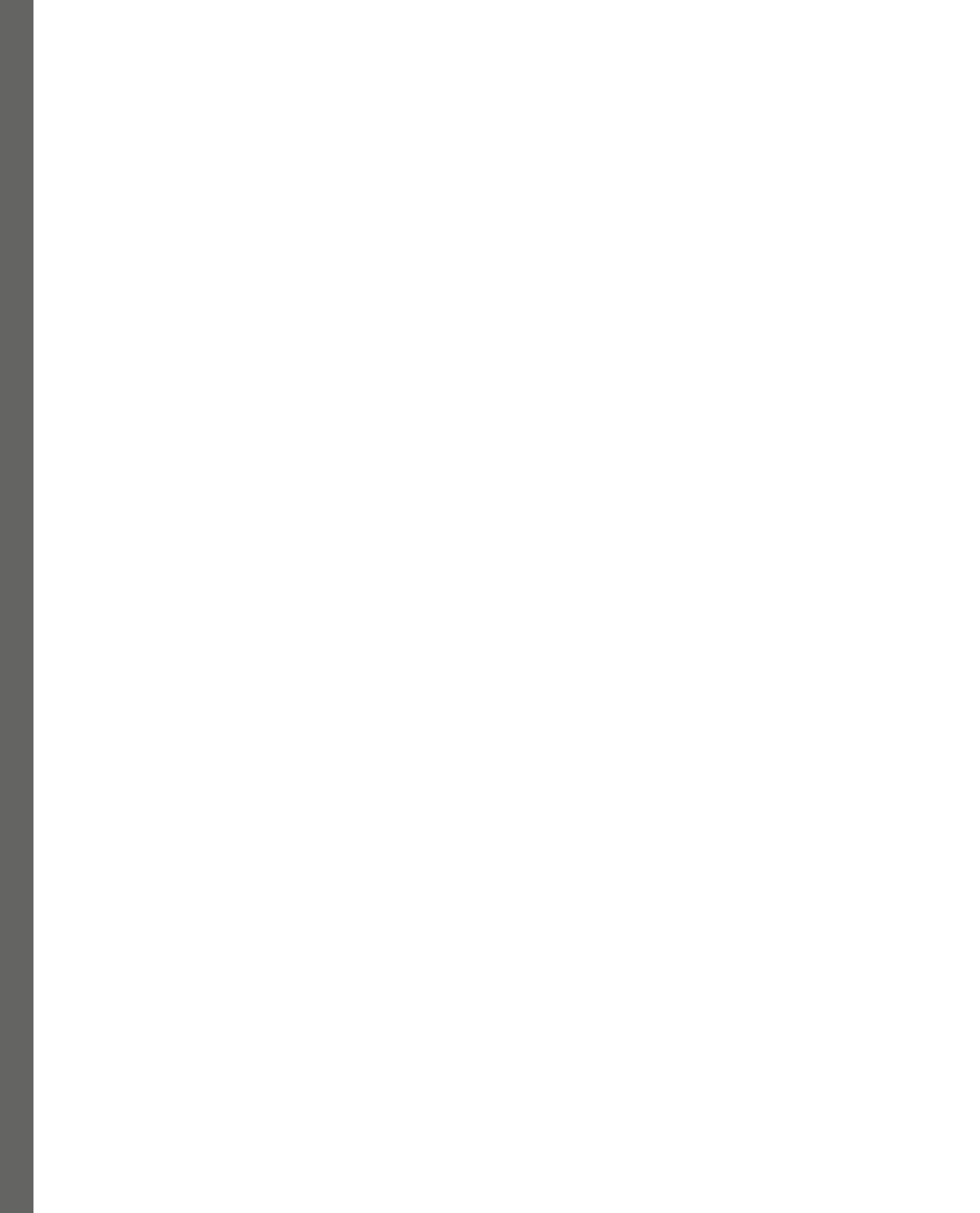
Part 3:

Managing Config and Infrastructure

This part takes a deep dive into infrastructure and configuration management in the public cloud, exploring various tools that enable infrastructure automation, configuration management, and immutable infrastructure.

This part has the following chapters:

- *Chapter 8, Infrastructure as Code (IaC) with Terraform*
- *Chapter 9, Configuration Management with Ansible*
- *Chapter 10, Immutable Infrastructure with Packer*



8

Infrastructure as Code (IaC) with Terraform

Cloud computing is one of the primary factors of DevOps enablement today. The initial apprehensions about the cloud are a thing of the past. With an army of security and compliance experts manning cloud platforms 24x7, organizations are now trusting the *public cloud* like never before. Along with cloud computing, another buzzword has taken the industry by storm – **Infrastructure as Code (IaC)**. This chapter will focus on IaC with **Terraform**, and by the end of this chapter, you will understand the concept and have enough hands-on experience with Terraform to get you started on your journey.

In this chapter, we're going to cover the following main topics:

- Introduction to IaC
- Setting up Terraform and Azure providers
- Understanding Terraform workflows and creating your first resource using Terraform
- Terraform modules
- Terraform state and backends
- Terraform workspaces
- Terraform outputs, state, console, and graphs

Technical requirements

For this chapter, you can use any machine to run Terraform. Terraform supports many platforms, including Windows, Linux, and macOS.

You will need an active Azure subscription to follow the exercises. Currently, Azure is offering a free trial for 30 days with \$200 worth of free credits; you can sign up at <https://azure.microsoft.com/en-in/free>.

You will also need to clone the following GitHub repository for some of the exercises:

<https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>

Run the following command to clone the repository into your home directory, and `cd` into the `ch8` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch8
```

So, let's get started!

Introduction to IaC

IaC is the concept of using code to define infrastructure. While most people can visualize infrastructure as something tangible, virtual infrastructure is already commonplace and has existed for around two decades. Cloud providers provide a web-based console through which you can manage your infrastructure intuitively. But the process is not repeatable or recorded.

If you spin up a set of infrastructure components using the console in one environment and want to replicate it in another, it is a duplication of effort. To solve this problem, cloud platforms provide APIs to manipulate resources within the cloud and some command-line tools that can help trigger the APIs. You can start writing scripts using commands to create the infrastructure and parameterize them to use the same scripts in another environment. Well, that solves the problem, right?

Not really! Writing scripts is an imperative way of managing infrastructure. Though you can still call it IaC, its problem is that it does not effectively manage infrastructure changes. Let me give you a few examples:

- What would happen if you needed to modify something already in the script? Changing the script somewhere in the middle and rerunning the entire thing may create havoc with your infrastructure. Imperative management of infrastructure is not idempotent. So, managing changes becomes a problem.
- What if someone manually changes the script-managed infrastructure using the console? Will your script be able to detect it correctly? What if you want to change the same thing using a script? It will soon start to get messy.
- With the advent of hybrid cloud architecture, most organizations use multiple cloud platforms for their needs. When you are in such a situation, managing multiple clouds with imperative scripts soon becomes a problem. Different clouds have different ways of interacting with their APIs and have distinct command-line tools.

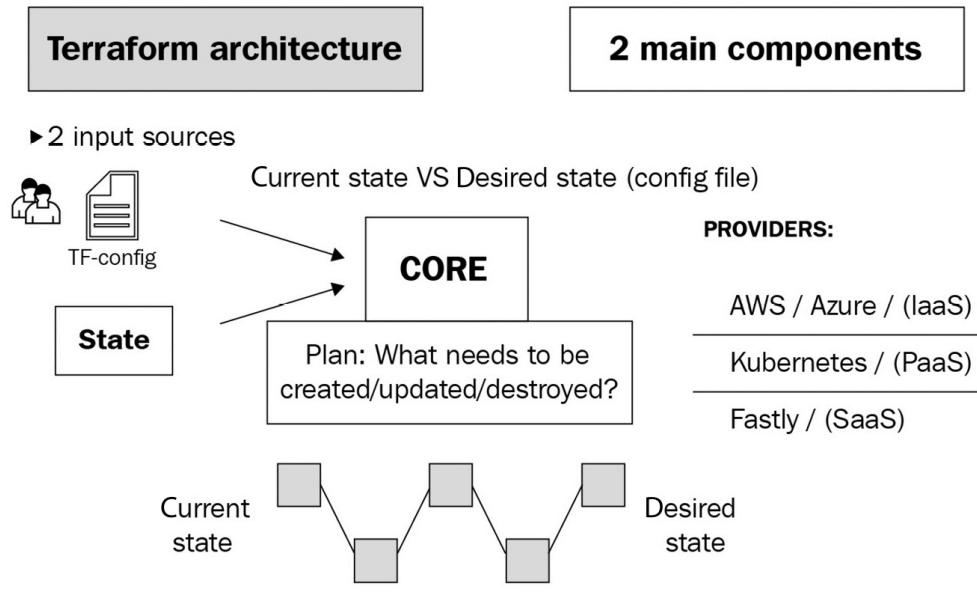
The solution to all these problems is a declarative IaC solution such as Terraform. HashiCorp's Terraform is the most popular IaC tool available on the market. It helps you automate and manage your infrastructure using code and can run on various platforms. As it is declarative, you just need to define what you need (the desired end state) instead of describing how to achieve it. It has the following features:

- It supports multiple cloud platforms via providers and exposes a single declarative **HashiCorp Configuration Language (HCL)**-based interface to interact with it. Therefore, it allows you to manage various cloud platforms using a similar language and syntax. So, having a few Terraform experts within your team can handle all your IaC needs.
- It tracks the state of the resources it manages using state files and supports local and remote backends to store and manage them. That helps in making the Terraform configuration idempotent. So, if someone manually changes a Terraform-managed resource, Terraform can detect the difference in the next run and prompt corrective action to bring it to the defined configuration. The admin can then absorb the change or resolve any conflicts before applying it.
- It enables GitOps in infrastructure management. With Terraform, you can have the infrastructure configuration alongside application code, making versioning, managing, and releasing infrastructure the same as managing code. You can also include code scanning and gating using pull requests so that someone can review and approve the changes to higher environments before you apply them. A great power indeed!

Terraform has multiple offerings – **open source**, **cloud**, and **enterprise**. The open source offering is a simple **command-line interface (CLI)**-based tool that you can download on any supported **operating system (OS)** and use. The cloud and enterprise offerings are more of a wrapper on top of the open source one. They provide a web-based GUI and advanced features such as **policy as code** with **Sentinel**, **cost analysis**, **private modules**, **GitOps**, and **CI/CD pipelines**.

This chapter will discuss the open source offering and its core functions.

Terraform open source is divided into two main parts – **Terraform Core** and **Terraform providers**, as seen in the following diagram:



Let's look at the functions of both components:

- **Terraform Core** is the CLI that we will use to interact with Terraform. It takes two main inputs – your Terraform configuration files and the existing state. It then takes the difference in configuration and applies it.
- **Terraform providers** are plugins that Terraform uses to interact with cloud providers. The providers translate the Terraform configuration into the respective cloud's REST API calls so that Terraform can manage its associated infrastructure. For example, if you want Terraform to manage AWS infrastructure, you must use the Terraform AWS provider.

Now let's see how we can install open source Terraform.

Installing Terraform

Installing Terraform is simple; go to <https://www.terraform.io/downloads.html> and follow the instructions for your platform. Most of it will require you to download a binary and move it to your system path.

Since we've been using Ubuntu throughout this book, I will show the installation on Ubuntu. Use the following commands to use the apt package manager to install Terraform:

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | \
  sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
```

```
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list
$ sudo apt update && sudo apt install terraform
```

Check whether Terraform has been installed successfully with the following command:

```
$ terraform version
Terraform v1.5.2
```

It shows that Terraform has been installed successfully. Terraform uses Terraform providers to interact with cloud providers, so let's look at those in the next section.

Terraform providers

Terraform has a decentralized architecture. While the Terraform CLI contains Terraform's core functionality and provides all functionalities not related to any specific cloud provider, Terraform providers provide the interface between the Terraform CLI and the cloud providers themselves. This decentralized approach has allowed public cloud vendors to offer their Terraform providers so that their customers can use Terraform to manage infrastructure in their cloud. Such is Terraform's popularity that it has now become an essential requirement for every public cloud provider to offer a Terraform provider.

We will interact with Azure for this chapter's entirety and use the Azure Terraform provider for our activity.

To access the resources for this section, cd into the following:

```
$ cd ~/modern-devops/ch8/terraform-exercise/
```

Before we go ahead and configure the provider, we need to understand how Terraform needs to authenticate and authorize with the Azure APIs.

Authentication and authorization with Azure

The simplest way to authenticate and authorize with Azure is to log in to your account using the Azure CLI. When you use the Azure provider within your Terraform file, it will automatically act as your account and do whatever it needs to. Now, this sounds dangerous. Admins generally have a lot of access, and having a tool that acts as an admin might not be a great idea. What if you want to plug Terraform into your CI/CD pipelines? Well, there is another way to do it – by using **Azure service principals**. Azure service principals allow you to access the required features without using a named user account. You can then apply the **principle of least privilege** to the service principal and provide only the necessary access.

Before configuring the service principal, let's install the Azure CLI on our machine. To do so, run the following command:

```
$ curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

The preceding command will download a shell script and execute it using `bash`. The script will then automatically download and configure the Azure CLI. To confirm whether the Azure CLI is installed successfully, run the following command:

```
$ az --version
azure-cli          2.49.0
```

We see that the Azure CLI is correctly installed on the system. Now, let's go ahead and configure the service principal.

To configure the Azure service principal, follow these steps.

Log in to Azure using the following command and follow all the steps the command prompts. You must browse to a specified URL and enter the given code. Once you've logged in, you will get a JSON response that will include some details, something like the following:

```
$ az login
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter
the code XXXXXXXX to authenticate:
[
  {
    "id": "00000000-0000-0000-0000-000000000000",
    ...
  }
]
```

Make a note of the `id` attribute, which is the subscription ID, and if you have more than one subscription, you can use the following to set it to the correct one:

```
$ export SUBSCRIPTION_ID=<SUBSCRIPTION_ID>
$ az account set --subscription="$SUBSCRIPTION_ID"
```

Use the following command to create a **service principal** with the `contributor` role to allow Terraform to manage the subscription's infrastructure.

Tip

Follow the principle of least privilege while granting access to the service principal. Do not give privileges thinking you might need them in the future. If any future access is required, you can grant it later.

We use contributor access for simplicity, but finer-grained access is possible and should be used:

```
$ az ad sp create-for-rbac --role="Contributor" \
--scopes="/subscriptions/$SUBSCRIPTION_ID"
Creating 'Contributor' role assignment under scope '/subscriptions/<SUBSCRIPTION_ID>'.
The output includes credentials that you must protect. Ensure you do not include these
credentials in your code or check the credentials into your source control (for more
information, see https://aka.ms/azadsp-cli):
{
  "appId": "00000000-0000-0000-0000-000000000000",
  "displayName": "azure-cli-2023-07-02-09-13-40",
  "password": "000000000000.xx-000000000000000000",
  "tenant": "00000000-0000-0000-0000-000000000000"
}
```

We've successfully created the **service principal**. The response JSON consists of `appId`, `password`, and `tenant`. We will need these to configure Terraform to use the service principal. In the next section, let's define the Azure Terraform provider with the details.

Using the Azure Terraform provider

Before we define the Azure Terraform provider, let's understand what makes a Terraform root module. The Terraform root module is just a working directory within your filesystem containing one or more `.tf` files that help you define your configuration and are where you would typically run your Terraform commands.

Terraform scans all your `.tf` files, combines them, and processes them internally as one. Therefore, you can have one or more `.tf` files that you can split according to your needs. While there are no defined standards for naming `.tf` files, most conventions use `main.tf` as the main Terraform file where they define resources, a `vars.tf` file for defining variables, and `outputs.tf` for defining outputs.

For this discussion, let's create a `main.tf` file within our working directory and add a provider configuration like the following:

```
terraform {
  required_providers {
    azurerm = {
      source  = "azurerm"
      version = "=3.55.0"
    }
  }
  provider "azurerm" {
    subscription_id = var.subscription_id
    client_id      = var.client_id
    client_secret   = var.client_secret
    tenant_id       = var.tenant_id
    features {}
  }
}
```

The preceding file contains two blocks. The `terraform` block contains the `required_providers` block, which declares the `version` constraint for the `azurerm` provider. The `provider` block declares an `azurerm` provider, which requires four parameters.

Tip

Always constrain the provider version, as providers are released without notice, and if you don't include the version number, something that works on your machine might not work on someone else's machine or the CI/CD pipeline. Using a version constraint avoids breaking changes and keeps you in control.

You might have noticed that we have declared several variables within the preceding file instead of directly inputting the values. There are two main reasons for that – we want to make our template as generic as possible to promote reuse. So, suppose we want to apply a similar configuration in another subscription or use another service principal; we should be able to change it by changing the variable values. Secondly, we don't want to check `client_id` and `client_secret` in source control. It is a bad practice as we expose our service principal to users beyond those who need to know about it.

Tip

Never store sensitive data in source control. Instead, use a `tfvars` file to manage sensitive information and keep it in a secret management system such as HashiCorp's Vault.

Okay, so as we've defined the provider resource and the attribute values are sourced from variables, the next step would be to declare variables. Let's have a look at that now.

Terraform variables

To declare variables, we will need to create a `vars.tf` file with the following data:

```
variable "subscription_id" {
  type     = string
  description = "The azure subscription id"
}
variable "app_id" {
  type     = string
  description = "The azure service principal appId"
}
variable "password" {
  type     = string
  description = "The azure service principal password"
  sensitive  = true
}
variable "tenant" {
  type     = string
  description = "The azure tenant id"
}
```

So, we've defined four variables here using `variable` blocks. Variable blocks typically have a `type` and a `description`. The `type` attribute defines the data type of the variable we declare and defaults to the `string` data type. It can be a primitive data type such as `string`, `number`, or `bool`, or a complex data structure such as `list`, `set`, `map`, `object`, or `tuple`. We will look at types in detail when we use them later in the exercises. The `description` attribute provides more information regarding the variable so users can refer to it for better understanding.

Tip

Always set the `description` attribute right from the beginning, as it is user-friendly and promotes the reuse of your template.

The `client_secret` variable also contains a third attribute called `sensitive`, a Boolean attribute set to `true`. When the `sensitive` attribute is `true`, the Terraform CLI does not display it in the screen's output. This attribute is highly recommended for sensitive variables such as passwords and secrets.

Tip

Always declare a sensitive variable as `sensitive`. This is because if you use Terraform within your CI/CD pipelines, unprivileged users might access sensitive information by looking at the logs.

Apart from the other three, an attribute called `default` will help you specify default variable values. The default values help you provide the best possible value for a variable, which your users can override if necessary.

Tip

Always use default values where possible, as they allow you to provide users with soft guidance about your enterprise standard and save them time.

The next step would be to provide variable values. Let's have a look at that.

Providing variable values

There are a few ways to provide variable values within Terraform:

- **Via the console using `-var` flags:** You can use multiple `-var` flags with the `variable_name=variable_value` string to supply the values.
- **Via a variable definition file (the `.tfvars` file):** You can use a file containing the list of variables and their values ending with an extension of `.tfvars` (if you prefer HCL) or `.tfvars.json` (if you prefer JSON) via the command line with the `-var-file` flag.

- **Via default variable definition files:** If you don't want to supply the variable definition file via the command line, you can create a file with the name `terraform.tfvars` or end it with an extension of `.auto.tfvars` within the Terraform workspace. Terraform will automatically scan these files and take the values from there.
- **Environment variables:** If you don't want to use a file or pass the information via the command line, you can use environment variables to supply it. You must create environment variables with the `TF_VAR_<var-name>` structure containing the variable value.
- **Default:** When you run a Terraform plan without providing values to variables in any other way, the Terraform CLI will prompt for the values, and you must manually enter them.

If multiple methods are used to provide the same variable's value, the first method in the preceding list has the highest precedence for a specific variable. It overrides anything that is defined in the methods listed later.

We will use the `terraform.tfvars` file for this activity and provide the values for the variables.

Add the following data to the `terraform.tfvars` file:

```
subscription_id = "<SUBSCRIPTION_ID>"  
app_id = "<SERVICE_PRINCIPAL_APP_ID>"  
password = "<SERVICE_PRINCIPAL_PASSWORD>"  
tenant = "<TENANT_ID>"
```

If you are checking the Terraform configuration into source control, add the file to the ignore list to avoid accidentally checking it in.

If you use Git, adding the following to the `.gitignore` file will suffice:

```
*.tfvars  
.terraform*
```

Now, let's go ahead and look at the Terraform workflow to progress further.

Terraform workflow

The Terraform workflow typically consists of the following:

- **init:** Initializes the Terraform **workspace** and **backend** (more on them later) and downloads all required providers. You can run the `init` command multiple times during your build, as it does not change your workspace or state.
- **plan:** It runs a speculative plan on the requested resources. This command typically connects with the cloud provider and then checks whether the objects managed by Terraform exist within the cloud provider and whether they have the same configuration as defined in the Terraform template. It then shows the delta in the plan output that an admin can review and change the

configuration if unsatisfied. If satisfied, they can apply the plan to commit the changes to the cloud platform. The `plan` command does not make any changes to the current infrastructure.

- `apply`: This applies the delta configuration to the cloud platform. When you use `apply` by itself, it runs the `plan` command first and asks for confirmation. If you supply a plan to it, it applies the plan directly. You can also use `apply` without running the plan using the `-auto-approve` flag.
- `destroy`: The `destroy` command destroys the entire infrastructure Terraform manages. It is, therefore, not a very popular command and is rarely used in a production environment. That does not mean that the `destroy` command is not helpful. Suppose you are spinning up a development infrastructure for temporary purposes and don't need it later. In that case, destroying everything you created using this command takes a few minutes.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-exercise
```

Now, let's look at these in detail with hands-on exercises.

terraform init

To initialize a Terraform workspace, run the following command:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "3.63.0"...
- Installing hashicorp/azurerm v3.63.0...
- Installed hashicorp/azurerm v3.63.0 (signed by HashiCorp)
Terraform has created a lock file, .terraform.lock.hcl, to record the provider selections
it made previously. Include this file in your version control repository so that Terraform
can guarantee to make the same selections by default when you run terraform init in the
future.
Terraform has been successfully initialized!
```

As the Terraform workspace has been initialized, we can create an **Azure resource group** to start working with the cloud.

Creating the first resource – Azure resource group

We must use the `azurerm_resource_group` resource within the `main.tf` file to create an Azure resource group. Add the following to your `main.tf` file to do so:

```
resource "azurerm_resource_group" "rg" {
  name      = var.rg_name
  location  = var.rg_location
}
```

As we've used two variables, we've got to declare those, so add the following to the `vars.tf` file:

```
variable "rg_name" {
  type     = string
  description = "The resource group name"
}
variable "rg_location" {
  type     = string
  description = "The resource group location"
}
```

Then, we need to add the resource group name and location to the `terraform.tfvars` file. Therefore, add the following to the `terraform.tfvars` file:

```
rg_name=terraform-exercise
rg_location="West Europe"
```

So, now we're ready to run a plan, but before we do so, let's use `terraform fmt` to format our files into the canonical standard.

terraform fmt

The `terraform fmt` command formats the `.tf` files into a canonical standard. Use the following command to format your files:

```
$ terraform fmt
terraform.tfvars
vars.tf
```

The command lists the files that it formatted. The next step is to validate your configuration.

terraform validate

The `terraform validate` command validates the current configuration and checks whether there are any syntax errors. To validate your configuration, run the following:

```
$ terraform validate
Success! The configuration is valid.
```

The success output denotes that our configuration is valid. If there were any errors, it would have highlighted them in the validated output.

Tip

Always run `fmt` and `validate` before every Terraform plan. It saves you a ton of planning time and helps you keep your configuration in good shape.

As the configuration is valid, we are ready to run a plan.

terraform plan

To run a Terraform plan, use the following command:

```
$ terraform plan
Terraform used the selected providers to generate the following execution plan. Resource
actions are indicated with the following symbols: + create
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id      = (known after apply)
      + location = "westeurope"
      + name    = "terraform-exercise"
  }
Plan: 1 to add, 0 to change, 0 to destroy.
Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to
take exactly these actions if you run terraform apply now.
```

The plan output tells us that if we run `terraform apply` immediately, it will create a single `terraform_exercise` resource group. It also outputs a note that since we did not save this plan, the subsequent application is not guaranteed to result in the same action. Meanwhile, things might have changed; therefore, Terraform will rerun `plan` and prompt us for `yes` when applying. Thus, you should save the plan to a file if you don't want surprises.

Tip

Always save `terraform plan` output to a file and use the file to apply the changes. This is to avoid any last-minute surprises with things that might have changed in the background and `apply` not doing what it is intended to do, especially when your plan is reviewed as a part of your process.

So, let's go ahead and save the plan to a file first using the following command:

```
$ terraform plan -out rg_terraform_exercise.tfplan
```

This time, the plan is saved to a file called `rg_terraform_exercise.tfplan`. We can use this file to apply the changes subsequently.

terraform apply

To apply the changes using the `plan` file, run the following command:

```
$ terraform apply "rg_terraform_exercise.tfplan"
azurerm_resource_group.rg: Creating...
```

```
azurerm_resource_group.rg: Creation complete after 2s [id=/subscriptions/id/resourceGroups/terraform-exercise]
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

And that's it! Terraform has applied the configuration. Let's use the Azure CLI to verify whether the resource group is created.

Run the following command to list all resource groups within your subscription:

```
$ az group list
...
{
  "id": "/subscriptions/id/resourceGroups/terraform-exercise",
  "location": "westeurope",
  "name": "terraform-exercise",
}
```

We see that our resource group is created and within the list.

There might be instances when `apply` is partially successful. In that case, Terraform will automatically taint resources it believes weren't created successfully. Such resources will be recreated automatically in the next run. If you want to taint a resource for recreation manually, you can use the `terraform taint` command:

```
$ terraform taint <resource>
```

Suppose we want to destroy the resource group as we no longer need it. We can use `terraform destroy` for that.

terraform destroy

To destroy the resource group, we can run a speculative plan first. It is always a best practice to run a speculative plan to confirm that what we need to destroy is within the output to have no surprises later. Terraform, like Linux, does not have an undo button.

To run a speculative destroy plan, use the following command:

```
$ terraform plan -destroy
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
- destroy
Terraform will perform the following actions:
# azurerm_resource_group.rg will be destroyed
- resource "azurerm_resource_group" "rg" {
    - id = "/subscriptions/id/resourceGroups/terraform-exercise" -> null
    - location = "westeurope" -> null
    - name = "terraform-exercise" -> null
    - tags = {} -> null
}
Plan: 0 to add, 0 to change, 1 to destroy.
```

As we see, as the resource group was the only resource managed by Terraform, it has listed that as the resource that will be destroyed. There are two ways of destroying the resource: using `terraform destroy` on its own or saving the speculative plan using the `out` parameter and running `terraform apply` on the destroy plan.

Let's use the first method for now.

Run the following command to destroy all resources managed by Terraform:

```
$ terraform destroy
Terraform will perform the following actions:
  # azurerm_resource_group.rg will be destroyed
Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above. There is no undo.
Only yes will be accepted to confirm.
Enter a value:
```

Now, this time, Terraform reruns `plan` and prompts for a value. It will only accept `yes` as confirmation. So, you can review the output, type `yes`, and hit *Enter* to confirm:

```
Enter a value: yes
azurerm_resource_group.rg: Destroying... [id=/subscriptions/id/resourceGroups/terraform-exercise]
azurerm_resource_group.rg: Still destroying... [id=/subscriptions/id/resourceGroups/terraform-exercise, 10s elapsed]
azurerm_resource_group.rg: Destruction complete after 1m20s
```

And it has now destroyed the resource group.

We've looked at a basic root module and explored the Terraform workflow. The basic root module helps us create and manage resources but lacks a very important feature – reusability. Terraform provides us with modules to allow reusability for common templates. Let's look at it in the next section.

Terraform modules

Terraform modules are reusable, repeatable templates. They allow abstraction in provisioning infrastructure, which is much needed if your usage grows beyond just some proof of concept. HashiCorp visualizes modules as designed by experts who know the enterprise standard and used by developers who want to apply the enterprise standard infrastructure in their projects. That way, things are standard across the organization. It saves developers time and avoids duplication of efforts. Modules can be versioned and distributed using a **module repository** or through your version control. That provides infrastructure admins with an ample amount of power and control at the same time.

As we created a resource group in the last section, let's modularize that in the next exercise. To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-modules/
```

Within this directory, we have the following directory structure:

```
.
├── main.tf
└── modules
    └── resource_group
        ├── main.tf
        └── vars.tf
└── terraform.tfvars
└── vars.tf
```

As we can see, we have the `main.tf`, `terraform.tfvars`, and `vars.tf` files in the root directory like before. However, we have included an additional `modules` directory, which contains a `resource_group` subdirectory that contains a `main.tf` file and a `vars.tf` file. Let's look at both.

`modules/resource_group/main.tf` looks like the following:

```
resource "azurerm_resource_group" "rg" {
  name      = var.name
  location  = var.location
}
```

It only contains an `azurerm_resource_group` resource with a name and location sourced from the `name` and `location` variables defined in the following `modules/resource_group/vars.tf` file:

```
variable "name" {
  type     = string
  description = "The resource group name"
}
variable "location" {
  type     = string
  description = "The resource group location"
}
```

In the root module, which is the current directory, we've modified the `main.tf` file to look like the following:

```
terraform {
  required_providers {
    ...
  }
  provider "azurerm" {
    ...
  }
  module "rg" {
    source  = "./modules/resource_group"
    name    = var.rg_name
  }
}
```

```
    location = var.rg_location
}
```

As we can see, instead of defining the resource directly in this file, we have defined a module called rg, whose source is ./modules/resource_group. Note that we pass the value for the variables defined for the module, that is, name, and location, from the variables defined at the root level, that is, var.rg_name and var.rg_location.

Now, let's go ahead and see what happens when we initialize and apply this configuration.

Run the following command to initialize the Terraform workspace:

```
$ terraform init
Initializing the backend...
Initializing modules...
- rg in modules/resource_group
Initializing provider plugins...
...
Terraform has been successfully initialized!
```

As we can see, Terraform has detected the new module and initialized it during init.

Tip

Whenever you define a new module, you must always reinitialize Terraform.

Now, let's go ahead and run a plan using the following command:

```
$ terraform plan
Terraform will perform the following actions:
  # module.rg.azurerm_resource_group.rg will be created
  + resource "azurerm_resource_group" "rg" {
      + id      = (known after apply)
      + location = "westeurope"
      + name     = "terraform-exercise"
    }
Plan: 1 to add, 0 to change, 0 to destroy.
```

As we can see, it will create the resource group. However, this is now a part of the module addressed module.rg.azurerm_resource_group.rg. To apply the plan, let's run the following command:

```
$ terraform apply
module.rg.azurerm_resource_group.rg: Creating...
module.rg.azurerm_resource_group.rg: Creation complete after 4s [id=/subscriptions/id/
resourceGroups/terraform-exercise]
```

And the resource group has been created! To destroy the resource group, let's run the following command:

```
$ terraform destroy
```

By using modules, you can simplify infrastructure creation and management, enhance collaboration among teams, and establish a consistent approach to deploying resources in a scalable and maintainable manner.

Tip

Use Terraform modules to encapsulate and reuse infrastructure configurations, promoting modularity and reusability.

Until now, we've seen Terraform creating and destroying resources, but how does Terraform know what it had created before and what it needs to destroy? Well, it uses a **state file** for that. Let's have a look.

Managing Terraform state

Terraform uses a state file to track what it has deployed and what resources it is managing. The state file is essential as it records all the infrastructure Terraform maintains. If you lose it, Terraform will lose track of what it has done so far and start treating resources as new and needing to be created again. Therefore, you should protect your state as code.

Terraform stores state in backends. By default, Terraform stores the state file as `terraform.tfstate` within the `workspace` directory, which is called the local backend. However, that is not the best way of managing the state. There are a couple of reasons why you should not store state in a local system:

- Multiple admins cannot work on the same infrastructure if the state file is stored within someone's local directory
- Local workstations are not backed up; therefore, the risk of losing the state file is high even if you have a single admin doing the job

You might argue that we can resolve these problems by checking the state file into source control with the `.tf` files. Don't do that! State files are plaintext, and if your infrastructure configuration contains sensitive information such as passwords, anyone can see it. Therefore, you need to store a state file securely. Also, storing state files in source control does not provide state locking, resulting in conflicts if multiple people are simultaneously modifying the state file.

Tip

Never store state files in source control. Use a `.gitignore` file entry to bypass the `terraform.tfstate` file.

The best place to store your Terraform state is on remote cloud storage. Terraform provides a remote backend to store state remotely. There are multiple types of remote backends you can use. When writing

this book, **Azure RM**, **Consul**, **cos**, **gcs**, **http**, **Kubernetes**, **oss**, **pg**, **S3**, and **Remote** were available backends. **Remote** is an enhanced backend type that allows running Terraform `plan` and `apply` within the backend, and only Terraform Cloud and Enterprise support it.

Tip

While choosing the state storage solution, you should prefer storage with state locking. That will allow multiple people to manipulate the resources without stepping on each other's shoes and causing conflict, as once a state file is locked, others cannot acquire it until the lock is released.

As we're using Azure, we can use Azure Storage to store our state. The advantages are three-fold:

- Your state file is centralized. You can have multiple admins working together and managing the same infrastructure.
- The store is encrypted at rest.
- You get automatic backup, redundancy, and high availability.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-backend/
```

Let's now use the `azurerm` backend and use Azure Storage to persist our Terraform state.

Using the Azure Storage backend

As we will end up in a chicken-or-egg situation if we use Terraform to build a backend to store its state, we will have to configure this bit without using Terraform.

Therefore, let's use the `az` command to configure the storage account in a different resource group that Terraform will not manage.

Creating Azure Storage resources

Let's start by defining a few variables:

- `$ RESOURCE_GROUP=tfstate`
- `$ STORAGE_ACCOUNT_NAME=tfstate$RANDOM`
- `$ CONTAINER_NAME=tfstate`

Create a resource group first using the following command:

```
$ az group create --name $RESOURCE_GROUP --location westeurope
```

Now, let's go ahead and create a storage account within the resource group using the following command:

```
$ az storage account create --resource-group $RESOURCE_GROUP \
--name $STORAGE_ACCOUNT_NAME --sku Standard_LRS \
--encryption-services BLOB
```

The next step is to fetch the account key using the following command:

```
$ ACCOUNT_KEY=$(az storage account keys list \
--resource-group tfstate --account-name $STORAGE_ACCOUNT_NAME \
--query '[0].value' -o tsv)
```

Now, we can go ahead and create a Blob Storage container using the following command:

```
$ az storage container create --name $CONTAINER_NAME \
--account-name $STORAGE_ACCOUNT_NAME --account-key $ACCOUNT_KEY
```

If we receive a created response, the storage account is created and ready for use. Now, we can go and define the backend configuration file in Terraform.

Creating a backend configuration in Terraform

Before we create the backend, we will need the `STORAGE_ACCOUNT_NAME` value. To get this, run the following command:

```
$ echo $STORAGE_ACCOUNT_NAME
tfstate28099
```

To create the backend configuration in Terraform, create a file called `backend.tf` within the workspace:

```
terraform {
  backend "azurerm" {
    resource_group_name  = "tfstate"
    storage_account_name = "tfstate28099"
    container_name       = "tfstate"
    key                 = "example.tfstate"
  }
}
```

In the backend configuration, we've defined the `resource_group_name` backend where the Blob Storage instance exists – `storage_account_name`, `container_name`, and `key`. The `key` attribute specifies the filename that we will use to define the state of this configuration. There might be multiple projects that you are managing using Terraform, and all of them will need separate state files. Therefore, the `key` attribute defines the state file's name that we will use for our project. That allows multiple Terraform projects to use the same Azure Blob Storage to store the state.

Tip

Always use the name of the project as the name of the key. For example, if your project name is `foo`, name the key `foo.tfstate`. That will prevent potential conflicts with others and also allow you to locate your state file quickly.

To initialize the Terraform workspace with the new backend configuration, run the following command:

```
$ terraform init
Initializing the backend...
Backend configuration changed!
Terraform has detected that the configuration specified for the backend has changed.
Terraform will now check for existing state in the backends.
Successfully configured the backend azurerm! Terraform will automatically use this backend
unless the backend configuration changes.
```

When we initialize that, Terraform detects that the backend has changed and checks whether anything is available in the existing backend. If it finds something, it asks whether we want to migrate the current state to the new backend. If it does not, it automatically switches to the new backend, as we see here.

Now, let's go ahead and use the `terraform plan` command to run a plan:

```
$ terraform plan
Acquiring state lock. This may take a few moments...
Terraform will perform the following actions:
# azurerm_resource_group.rg will be created
+ resource "azurerm_resource_group" "rg" {
  ...
}
Plan: 1 to add, 0 to change, 0 to destroy.
```

So, as we see, `terraform plan` tells us that it will create a new resource group called `terraform-exercise`. Let's apply the configuration, and this time with an `auto-approve` flag so that the plan does not run again, and Terraform immediately applies the changes using the following command:

```
$ terraform apply -auto-approve
Acquiring state lock. This may take a few moments...
azurerm_resource_group.rg: Creating...
azurerm_resource_group.rg: Creation complete after 2s [id=/subscriptions/id/
resourceGroups/terraform-exercise]
Releasing state lock. This may take a few moments...
```

We now have the resource created successfully.

Now, let's go to Azure Blob Storage and see whether we have a `tfstate` file there, as shown in the following screenshot:

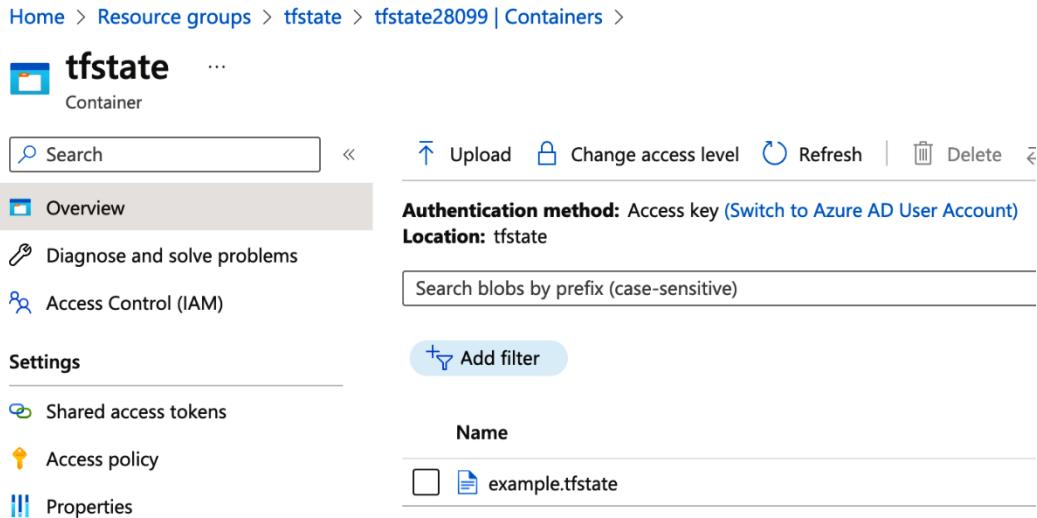


Figure 8.2 – Terraform state

As we see, we have a file called `example.tfstate` within the blob container. That is how remote storage works, and now anyone with access to the Blob Storage instance can use the Terraform configuration and make changes.

So far, we've been managing resources using the default workspace, but what if there are multiple environments that you need to control using the same configuration? Well, Terraform offers workspaces for those scenarios.

Terraform workspaces

Software development requires multiple environments. You develop software within your workspace, deploy it into the development environment, unit test it, and then promote the tested code to a test environment. Your QA team will test the code extensively in the test environment, and once all test cases pass, you can promote your code to production.

That means you must maintain a similar infrastructure in all environments. With an IaC tool such as Terraform, infrastructure is represented as code, and we must manage our code to fit multiple environments. But Terraform isn't just code; it also contains state files, and we must maintain state files for every environment.

Suppose you want to create three resource groups, `terraform-exercise-dev`, `terraform-exercise-test`, and `terraform-exercise-prod`. Each resource group will contain a similar set of infrastructure with similar properties. For example, each resource group includes an Ubuntu **Virtual Machine (VM)**.

A simple method to approach the problem is by creating a structure like the following:

```
└── dev
    ├── backend.tf
    ├── main.tf
    ├── terraform.tfvars
    └── vars.tf
└── prod
    ├── backend.tf
    ├── main.tf
    ├── terraform.tfvars
    └── vars.tf
└── test
    ├── backend.tf
    ├── main.tf
    ├── terraform.tfvars
    └── vars.tf
```

Can you see the duplication? The same files occur multiple times, all containing the same configuration. The only thing that might change is the `terraform.tfvars` file for each environment.

So, this does not sound like a great way to approach this problem, and that's why Terraform provides workspaces for it.

Terraform workspaces are nothing but independent state files. So, you have a single configuration and multiple state files for each environment. Sounds simple, right? Let's have a look.

Another way to represent the same configuration by using Terraform workspaces is the following:

```
└── backend.tf
└── main.tf
└── terraform.tfvars
└── vars.tf
```

Now, this looks simple. It just contains a single set of files. Let's have a look at each of them to understand them better.

To access the resources for this section, `cd` into the following:

```
$ cd ~/modern-devops/ch8/terraform-workspaces/
```

The `main.tf` file contains a `resource_group` resource with a name that includes an environment suffix, along with other resources that we need to create within the resource group, such as the VNet, subnet, and VM, something like the following:

```
...
resource "azurerm_resource_group" "main" {
  name      = "${var.rg_prefix}-${terraform.workspace}"
  location  = var.rg_location
}
resource "azurerm_virtual_network" "main" {
  ...
}
resource "azurerm_subnet" "internal" {
  ...
}
resource "azurerm_network_interface" "main" {
  ...
}
resource "azurerm_virtual_machine" "main" {
  ...
}
...
}
```

To access the name of the workspace, Terraform provides the `terraform.workspace` variable, which we have used to define the `resource_group` name. So, the template is now ready to take configuration for any environment, and we will have a separate resource group for each environment.

Also, update the `backend.tf` file with the `tfstate` container name we created in the last section and initialize the Terraform workspace by using the following command:

```
$ terraform init
```

Now, once Terraform has initialized, let's create a `dev` workspace by using the following command:

```
$ terraform workspace new dev
Created and switched to workspace "dev"!
```

You're now in a new, empty workspace. Workspaces isolate their state, so if you run `terraform plan`, Terraform will not see any existing state for this configuration.

So, as we're in a new, empty workspace called `dev`, let's run a plan.

Use the following command to run a plan on the `dev` environment:

```
$ terraform plan -out dev.tfplan
Acquiring state lock. This may take a few moments...
Terraform will perform the following actions:
+ resource "azurerm_network_interface" "main" {
  ...
}
+ resource "azurerm_resource_group" "main" {
```

```
+ id      = (known after apply)
+ location = "westeurope"
+ name    = "terraform-ws-dev"
}
+ resource "azurerm_subnet" "internal" {
...
}
+ resource "azurerm_virtual_machine" "main" {
...
}
+ resource "azurerm_virtual_network" "main" {
...
}
}
Plan: 5 to add, 0 to change, 0 to destroy.
```

Now, let's go ahead and apply the plan using the following command:

```
$ terraform apply "dev.tfplan"
Acquiring state lock. This may take a few moments...
azurerm_resource_group.main: Creating...
azurerm_virtual_network.main: Creating...
azurerm_subnet.internal: Creating...
azurerm_network_interface.main: Creating...
azurerm_virtual_machine.main: Creating...
Apply complete! Resources: 5 added, 0 changed, 0 destroyed.
Releasing state lock. This may take a few moments...
```

As the dev plan has been applied and the resources are created in the dev resource group, let's create a workspace for testing:

```
$ terraform workspace new test
```

As the new workspace is created, let's run a plan on the test workspace using the following command and save it to the `test.tfplan` file:

```
$ terraform plan -out test.tfplan
...
+ resource "azurerm_resource_group" "main" {
    + id      = (known after apply)
    + location = "westeurope"
    + name    = "terraform-ws-test"
}
...
```

As we can see, the resources will be created in the `terraform-ws-test` resource group. So, let's go ahead and apply the plan using the following command:

```
$ terraform apply test.tfplan
```

The `test` plan has been applied as well. Now let's go ahead and inspect the created resources.

Tip

Terraform workspaces are ideal for maintaining separate infrastructure configurations for different environments, such as development, staging, and production. This helps prevent accidental configuration changes and ensures consistent setups.

Inspecting resources

Let's use the `az` command to list the resource groups. As we know, our resource groups have a resource group prefix of `terraform-ws`. Therefore, use the following command to list all resource groups containing the prefix:

```
$ az group list | grep name | grep terraform-ws
  "name": "terraform-ws-dev",
  "name": "terraform-ws-test",
```

As we can see, we have two resource groups, `terraform-ws-dev` and `terraform-ws-test`. So, two resource groups have been created successfully.

You can also verify this in the Azure portal, as shown in the following screenshot:

The screenshot shows the Azure portal's Resource groups page. At the top, there is a search bar with the text 'terraform-ws'. Below the search bar, there are several filter options: 'Subscription == all' (selected), 'Location == all' (selected), and a 'Add filter' button. The main area displays a table with two rows, each representing a resource group. The first row has a checkbox, the name 'terraform-ws-dev', and a small icon. The second row has a checkbox, the name 'terraform-ws-test', and a small icon. The table has a header row with a 'Name' column and a '↓' symbol.

Name ↓
<input type="checkbox"/> terraform-ws-dev
<input type="checkbox"/> terraform-ws-test

Figure 8.3 – Resource groups

Now, let's go ahead and inspect the resources on the `terraform-ws-dev` resource group using the Azure portal by clicking on `terraform-ws-dev`:

Name	Type
app network	Virtual machine
app-nic	Virtual machine
httpd	Virtual machine
httpd-osdisk	Disk

Figure 8.4 – Terraform dev resource group

We have a virtual network, a network interface, an OS disk, and a VM within the resource group. We should expect similar resources with the same names in the `terraform-ws-test` resource group. Let's go ahead and have a look:

Name	Type
app-network	Virtual machine
app-nic	Virtual machine
httpd	Virtual machine
httpd-osdisk	Disk

Figure 8.5 – Terraform test resource group

As we can see, we also have similar resources in the `terraform-ws-test` resource group.

We did all this using a single configuration, but there should be two state files for each workspace since they are two sets of resources. Let's have a look.

Inspecting state files

If we had used the local backend for the state files, we would get the following structure:

```
|-- terraform.tfstate.d
|-- dev
|  '-- terraform.tfstate
`-- test
    '-- terraform.tfstate
```

So, Terraform creates a directory called `terraform.tfstate.d`; within that, it creates directories for each workspace. Within the directories, it stores the state file for each workspace as `terraform.tfstate`.

But since we are using a remote backend and using Azure Blob Storage for it, let's inspect the files within it using the Azure console:

The screenshot shows the Azure Blob Storage interface for a container named 'tfstate'. At the top, there are navigation buttons for 'Upload', 'Change access level', 'Refresh', and 'Delete'. Below this, the 'Authentication method' is set to 'Access key' with a link to 'Switch to Azure AD User Account'. The 'Location' is specified as 'tfstate'. A search bar contains the text 'ws'. Under the heading 'Name', there are two entries: 'ws.tfstateenv:dev' and 'ws.tfstateenv:test', each preceded by a checkbox.

Name
<input type="checkbox"/> ws.tfstateenv:dev
<input type="checkbox"/> ws.tfstateenv:test

Figure 8.6 – Terraform workspace state

As we see, there are two state files, one for each environment. Therefore, the state files are suffixed with an `env:dev` or `env:test` string. That is how workspaces are managed in Azure Blob Storage. The remote backend's structure for maintaining state files depends on the provider plugins, and therefore, there might be different ways of managing multiple states for various backends. However, the Terraform CLI will interpret workspaces the same way, irrespective of the backends, so nothing changes for the end user from a CLI perspective.

Cleaning up

Now, let's go ahead and clean up both resource groups to avoid unnecessary charges.

As we're already within the test workspace, let's run the following command to destroy resources within the test workspace:

```
$ terraform destroy --auto-approve
```

Now, let's switch to the dev workspace using the following command:

```
$ terraform workspace select dev
Switched to workspace "dev".
```

As we're within the dev workspace, use the following command to destroy all resources within the dev workspace:

```
$ terraform destroy --auto-approve
```

In a while, we should see that both resource groups are gone. Now, let's look at some of the advanced concepts of Terraform in the next section.

Terraform output, state, console, and graphs

While we understand that Terraform uses state files to manage resources, let's look at some advanced commands to help us appreciate and make more sense of the Terraform state concept.

To access the resources for this section, cd into the following:

```
$ cd ~/modern-devops/ch8/terraform-workspaces/
```

Now, let's go ahead and look at our first command – `terraform output`.

terraform output

So far, we've looked at variables but haven't yet discussed outputs. Terraform outputs are return values of a Terraform configuration that allow users to export configuration to users or any modules that might use the current module.

Let's go with the last example and add an output variable that exports the private IP of the network interface attached to the VM in the `outputs.tf` file:

```
output "vm_ip_addr" {
  value = azurerm_network_interface.main.private_ip_address
}
```

Now, let's go ahead and apply the configuration:

```
$ terraform apply --auto-approve  
...  
Outputs:  
vm_ip_addr = "10.0.2.4"
```

After Terraform has applied the configuration, it shows the outputs at the end of the console result. You can run the following to inspect the output anytime later:

```
$ terraform output  
vm_ip_addr = "10.0.2.4"
```

Outputs are stored in the state file like everything else, so let's look at how we can manage Terraform state using the CLI.

Managing Terraform state

Terraform stores the configuration it manages in state files and therefore provides a command for advanced state management. The `terraform state` command helps you manage the state of the current configuration. While the state file is plaintext and you can manually modify it, using the `terraform state` command is recommended.

But before we get into details, we must understand why we want to do that. Things might not always go according to plan, so the state file may have corrupt data. You also might want to see specific attributes of a particular resource after you've applied it. The state file might need to be investigated for the root cause analysis of a specific infrastructure provisioning problem. Let's have a look at the most common use cases.

Viewing the current state

To view the current state, we can run the following command:

```
$ terraform show
```

That will output all resources that Terraform has created and manages, including outputs. Of course, this can be overwhelming for some, and we may want to view the list of resources Terraform manages.

Listing resources in the current state

To list the resources in the Terraform state file, run the following command:

```
$ terraform state list  
azurerm_network_interface.main  
azurerm_resource_group.main  
azurerm_subnet.internal
```

```
azurerm_virtual_machine.main  
azurerm_virtual_network.main
```

And as we see, there are five resources managed by Terraform. You might want to remove a resource from the Terraform state. It might be possible that someone has removed a resource manually as it is no longer required, but it isn't removed from the Terraform configuration.

Removing a resource from the state

To remove a state manually from the Terraform state file, you must use the `terraform state rm <resource>` command. For example, to remove the Azure VM resource from the Terraform state, run the following command:

```
$ terraform state rm azurerm_virtual_machine.main  
Acquiring state lock. This may take a few moments...  
Removed azurerm_virtual_machine.main  
Successfully removed 1 resource instance(s).  
Releasing state lock. This may take a few moments...
```

Bear in mind that this has merely removed the resource from the state file and has not touched the actual resource sitting on Azure.

There might be instances where someone spun up a VM manually within Azure, and we now want Terraform to manage it. This kind of situation happens mostly in brownfield projects. In that case, we must declare the same configuration within Terraform and then import existing resources in the Terraform state. To do so, we can use the `terraform import` command.

Importing existing resources into Terraform state

You can use the `terraform import` command to import existing resources into Terraform state. The `terraform import` command is structured as follows:

```
terraform import <resource> <resource_id>
```

For example, to reimport the `httpd` VM into the state, run the following command:

```
$ terraform import azurerm_virtual_machine.main \  
"/subscriptions/<SUBSCRIPTION_ID>/resourceGroups\  
/terraform-ws-dev/providers/Microsoft.Compute/virtualMachines/httpd"  
Acquiring state lock. This may take a few moments...  
azurerm_virtual_machine.main: Importing from ID "/subscriptions/id/resourceGroups/  
terraform-ws-dev/providers/Microsoft.Compute/virtualMachines/httpd"...  
azurerm_virtual_machine.main: Import prepared!  
  Prepared azurerm_virtual_machine for import  
azurerm_virtual_machine.main: Refreshing state... [id=/subscriptions/1de491b5-f572-  
459b-a568-c4a35d5ac7a9/resourceGroups/terraform-ws-dev/providers/Microsoft.Compute/  
virtualMachines/httpd]  
Import successful!
```

To check whether the resource is imported to the state, we can list the resources again using the following command:

```
$ terraform state list | grep azurerm_virtual_machine  
azurerm_virtual_machine.main
```

As we see, we have the VM within the state file. If we want to dig further into the resources, we can use `terraform console`.

terraform console

The `terraform console` command provides an interactive console to investigate state files, dynamically build paths, and evaluate expressions even before using them in resources. It is a potent tool that most advanced Terraform users use. For example, let's launch the console and look through the configuration of the VM resource we just imported.

Use the following commands to launch the console and get the resource group of the VM and the `id` value:

```
$ terraform console  
Acquiring state lock. This may take a few moments...  
> azurerm_virtual_machine.main.resource_group_name  
"terraform-ws-dev"  
> azurerm_virtual_machine.main.id  
"/subscriptions/id/resourceGroups/terraform-ws-dev/providers/Microsoft.Compute/  
virtualMachines/httpd"  
> exit  
Releasing state lock. This may take a few moments...
```

As we can see, the VM is in the correct resource group, and we're satisfied that the import was correct.

Terraform dependencies and graphs

Terraform uses a dependency model to manage in what order resources are created and destroyed. There are two kinds of dependencies – *implicit* and *explicit*. We've been using implicit dependencies until now, where the VM depended upon the network interface, and the network interface depended upon the subnet. The subnet depended upon the virtual network, and all of these resources depended on the resource group. These dependencies naturally occur when we use one resource's output as another's input.

However, sometimes, we want to define an explicit dependency on a resource, especially when there is no way to define an implicit dependency on it. You can use the `depends_on` attribute for that kind of operation.

Tip

Avoid explicit dependencies unless needed, as Terraform uses parallelism to manage resources. If explicit dependencies are not required, it will slow down Terraform runs because it can process multiple parallel resources.

To visualize the dependencies between resources, we can export a graph from the state file and convert that into a PNG file using a tool such as **Graphviz**.

Run the following command to export the dependency graph:

```
$ terraform graph > vm.dot
```

We can then process the graph file using the Graphviz tool. To install the tool on Ubuntu, run the following command:

```
$ sudo apt install graphviz -y
```

Now run the following command to convert the graph file into a PNG file:

```
$ cat vm.dot | dot -T png -o vm.png
```

The graph is available at <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e/blob/main/ch8/terraform-graph.png>. Now, let's go ahead and see how we can clean up our resources.

Cleaning up resources

As we already know, we run the following command to clean up the resources:

```
$ terraform destroy --auto-approve
```

It will clear resources from the resource group and delete the resource group after that.

While using `terraform destroy` can be an easy way to eliminate resources you don't need, it is best if you stick to this only in the dev environment and never use it in production. Instead, you can remove resources you don't need from the configuration and then run `terraform apply`.

Summary

In this chapter, we've discussed Terraform's core and understood some of the most common commands and functionalities from a hands-on perspective. We started with understanding IaC, introduced Terraform as an IaC tool, installed Terraform, understood Terraform providers, and used the Azure Terraform provider to manage infrastructure in Azure.

We then looked at Terraform variables and multiple ways of supplying values to the variables. We discussed the core Terraform workflow and several commands you would use to manage infrastructure using Terraform. We then looked at Terraform modules and then at Terraform state as an essential component that helps Terraform keep track of the infrastructure it is managing.

We looked at local and remote state storage and used Azure Blob Storage as the remote state backend. We then discussed Terraform workspaces and how they enable us to use the same Terraform configuration to build multiple environments with hands-on exercises.

We then looked at some advanced operations with Terraform state using the `outputs`, `state`, and `console` commands. We finally looked at how Terraform manages dependencies and viewed a dependency graph using the `graph` command.

In the next chapter, we will delve into configuration management using Ansible.

Questions

1. Why should we constrain the provider version?
2. You should always use the `fmt` and `validate` functions before a Terraform plan. (True/False)
3. What does the Terraform `plan` command do? (Choose two)
 - A. Refreshes the current state with the existing infrastructure state
 - B. Gets the delta between the current configuration and the expected configuration
 - C. Applies the configuration to the cloud
 - D. Destroys the configuration in the cloud
4. What does the `terraform apply` command do? (Choose three)
 - A. Refreshes the current state with the existing infrastructure
 - B. Gets the delta between the current configuration and the expected configuration
 - C. Applies the configuration to the cloud
 - D. Destroys the configuration in the cloud

5. Why should you never store state files in source control? (Choose two)
 - A. State files are plaintext, and therefore you expose sensitive information to unprivileged users.
 - B. Source control does not support state locking, and therefore it might result in potential conflicts between users.
 - C. Multiple admins cannot work on the same configuration.
6. Which of the following are valid Terraform remote backends? (Choose five)
 - A. S3
 - B. Azure Blob Storage
 - C. Artifactory
 - D. Git
 - E. HTTP
 - F. Terraform Enterprise
7. Which command will mark a resource for recreation in the next `apply`?
8. Where are state files stored in the local backend if you use workspaces?
9. What command should we use to remove a Terraform resource from the state?
10. What command should we use to import an existing cloud resource within the state?

Answers

1. Because Terraform providers are released separately to the Terraform CLI, different versions might break the existing configuration
2. True
3. A, B
4. A, B, C
5. A, B
6. A, B, C, E, F
7. The `taint` command
8. `terraform.tfstate.d`
9. `terraform state rm <resource>`
10. `terraform import <resource> <id>`

9

Configuration Management with Ansible

In the last chapter, we looked at **Infrastructure as Code (IaC)** with Terraform, its core concepts, IaC workflow, state, and debugging techniques. We will now delve into **configuration management (CM)** and **Configuration as Code (CaC)** with Ansible. Ansible is a CM tool that helps you to define configuration as idempotent chunks of code.

In this chapter, we're going to cover the following main topics:

- Introduction to configuration management
- Setting up Ansible
- Introduction to Ansible playbooks
- Ansible playbooks in action
- Designing for reusability

Technical requirements

You will need an active Azure subscription to follow the exercises for this chapter. Currently, Azure is offering a free trial for 30 days with \$200 worth of free credits, and you can sign up at <https://azure.microsoft.com/en-in/free>.

You will also need to clone the following GitHub repository for some of the exercises:

<https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>

Run the following command to clone the repository into your home directory, and `cd` into the `ch9` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch9
```

You also need to install Terraform on your system. Refer to *Chapter 8, Infrastructure as Code (IaC) with Terraform*, for more details on installing and setting up Terraform.

Introduction to configuration management

CM, in the realm of technology and systems administration, can be compared to the role of a conductor leading an orchestra. Imagine yourself guiding a group of musicians, each playing a unique instrument. Your responsibility is to ensure that everyone is harmoniously in sync, adhering to the correct musical score, and executing their parts at precisely the right moments.

In the context of technology and systems administration, CM is the practice of skilfully orchestrating and overseeing the creation, updates, and maintenance of computer systems and software, much akin to how a conductor directs musicians to produce splendid music.

Here's a breakdown of how it functions:

- **Standardization:** As with musicians employing the same musical notes and scales, CM guarantees that all computers and software within an organization adhere to standardized configurations. This uniformity mitigates errors and bolsters system reliability.
- **Automation:** In an orchestra, musicians don't manually fine-tune their instruments during a performance. Likewise, CM tools automate the configuration and upkeep of computer systems, consistently applying configurations without the need for manual intervention.
- **Version control:** Musicians follow specific sheet music, and if alterations occur, everyone receives updated sheet music. CM maintains a version history of system configurations, simplifying the tracking of changes, reverting to previous versions, and ensuring alignment across the board.
- **Efficiency:** Just as a conductor synchronizes the timing of each instrument, CM optimizes system performance and resource allocation. It guarantees that software and systems operate efficiently and can scale as required.
- **Compliance and security:** Analogous to a conductor enforcing performance guidelines, CM enforces adherence to security policies and best practices. It plays a crucial role in upholding a secure and compliant IT environment.
- **Troubleshooting:** When issues arise during a performance, the conductor swiftly identifies and addresses them. CM tools assist in troubleshooting and rectifying configuration-related problems in IT systems.

To understand CM better, let's first look at the traditional way of hosting and managing applications. We first create a **virtual machine (VM)** from physical infrastructure and then log in manually to VMs. We can then run a set of scripts or do the setup manually. At least, that's what we've been doing till now, even in this book.

There are several problems with this approach. Let's look at some of them:

- If we set up the server manually, the process is not repeatable. For example, if we need to build another server with a similar configuration, we must repeat the entire process to build another server.
- Even if we use scripts, the scripts themselves are not idempotent. This means they cannot identify and apply only the delta configuration if needed.
- Typical production environments consist of many servers; therefore, setting everything up manually is a labor-intensive task and adds to the toil. Software engineers should focus on novel ways of automating processes that cause toil.
- While you can store scripts within source control, they are *imperative*. We always encourage a *declarative* way of managing things.

Modern CM tools such as Ansible solve all these problems by providing the following benefits:

- They manage configuration through a set of declarative code pieces
- You can store code in version control
- You can apply code to multiple servers from a single control node
- As they are idempotent, they only apply the delta configuration
- It is a repeatable process; you can use variables and templates to apply the same configuration to multiple environments
- They provide deployment orchestration and are mostly used within CI/CD pipelines

Although many tools available on the market provide CM, such as **Ansible**, **Puppet**, **Chef**, and **SaltStack**, Ansible is the most popular and straightforward tool used for this. It is more efficient, and its simplicity makes it less time-consuming than others.

It is an open source CM tool built using Python and is owned by **Red Hat**. It provides the following features:

- It helps you to automate routine tasks such as OS upgrades, patches, and backups while also creating all OS-level configurations, such as users, groups, permissions, and others
- The configuration is written using simple YAML syntax
- It uses **Secure Shell (SSH)** to communicate with managed nodes and sends commands
- The commands are executed sequentially within each node in an idempotent manner
- It connects to nodes parallelly to save time

Let's delve into the reasons why using Ansible is a great choice for CM and automation. Here are some compelling factors:

- **Simplicity and user-friendliness:** Ansible boasts an uncomplicated, human-readable YAML syntax that's easy to grasp and employ, even for those with limited coding experience.
- **Agentless approach:** Ansible communicates through SSH or WinRM, eliminating the need to install agents on managed nodes. This reduces overhead and security concerns, a topic we'll explore further when we discuss Ansible architecture.
- **Idempotent operations:** Ansible ensures the desired system state is achieved, even if configurations are applied repeatedly. This minimizes the risk of unintended changes.
- **Broad adoption:** With a thriving and active user community, Ansible offers extensive documentation, modules, and playbooks for various use cases.
- **Cross-platform compatibility:** Ansible can handle diverse environments, managing various operating systems, cloud providers, network devices, and infrastructure components with a single tool.
- **Seamless integration:** Ansible seamlessly integrates with other tools, including **version control systems (VCSS)**, monitoring solutions, and CI/CD pipelines.
- **Scalability:** Ansible scales effortlessly to handle both small and large environments, catering to both enterprises and start-ups.
- **Version control:** Infrastructure configurations are stored in plain text files, simplifying change management, history tracking, and collaboration through Git or similar VCSs.
- **Automation of routine tasks:** Ansible automates repetitive chores such as software installations, configuration updates, and patch management, freeing up time for strategic tasks.
- **Security and compliance:** Implement security policies and compliance standards consistently across your infrastructure using Ansible's **Role-Based Access Control (RBAC)** and integrated security modules.
- **Rollback and recovery:** Ansible enables easy rollback to prior configurations in the case of issues, reducing downtime and minimizing the impact of changes.
- **Modularity and reusability:** Ansible encourages the creation of modular, reusable playbooks and roles, fostering an organized and efficient automation approach.
- **Supportive community:** Benefit from a robust Ansible community that offers support, documentation, and a repository of contributed roles and modules.
- **Cost-effective:** Ansible is open source and free to use, cutting down on licensing expenses compared to other automation tools.
- **Orchestration and workflow automation:** Beyond CM, Ansible can orchestrate intricate workflows, including application deployment and infrastructure provisioning.

- **Immutable infrastructure:** Ansible supports the concept of immutable infrastructure, where changes involve recreating components rather than modifying them in place. This leads to more predictable and dependable deployments.
- **Real-time feedback:** Ansible provides real-time feedback and reporting, simplifying the monitoring and troubleshooting of automation tasks.

These advantages establish Ansible as a popular choice for CM, automation, and orchestration across a wide spectrum of IT environments and industries.

Ansible has a simple architecture. It has a **control node** that takes care of managing multiple **managed nodes**. All you need is a control node server to install Ansible and the nodes to manage using the control node (also known as managed nodes). The managed nodes should allow an SSH connection from the Ansible control node—something like the following diagram:

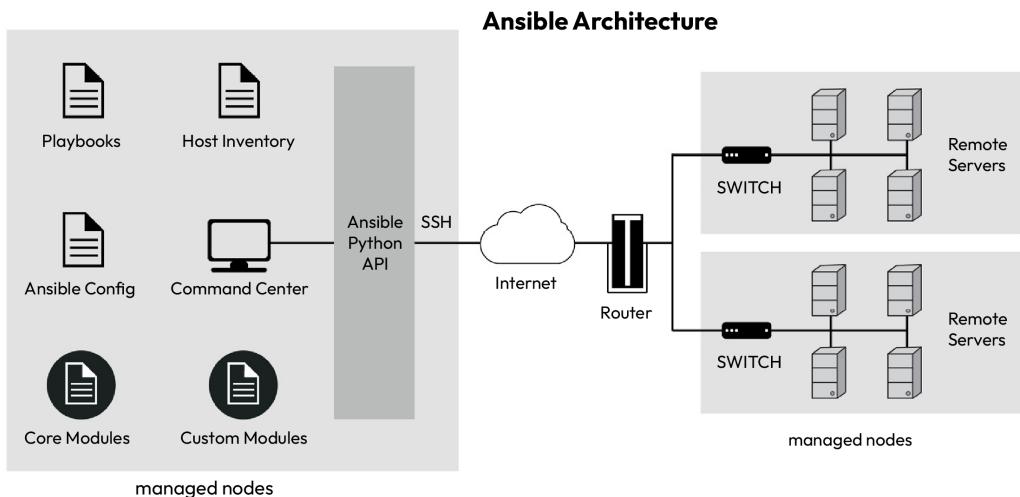


Figure 9.1 – Ansible architecture

Now, let's go ahead and see how we can install and set up the required configuration using Ansible. Let's look at how to install Ansible in the next section.

Setting up Ansible

We need to set up and install Ansible in the control node, but before we do that, we will have to spin three servers to start the activity—an Ansible control node and two managed nodes.

Setting up inventory

The idea is to set up a two-tier architecture with **Apache** and **MySQL**. So, let's use Terraform to spin up the three servers.

Let's first cd into the directory where the Terraform templates are located and then edit the `terraform.tfvars` file to fill in the required details. (Please refer to *Chapter 8, Infrastructure as Code (IaC) with Terraform*, for more details about how to get the attributes):

```
$ cd ~/modern-devops/ch9/setup-ansible-terraform
$ vim terraform.tfvars
```

Then, use the following commands to spin up the servers using Terraform:

```
$ terraform init
$ terraform plan -out ansible.tfplan
$ terraform apply ansible.tfplan
```

Once the `terraform apply` command is completed successfully, we will see three servers—`ansible-control-node`, `web`, and `db`, and the associated resources created within the `ansible-exercise` resource group.

The `terraform apply` output also provides the public IP addresses of the Ansible control node and the web VM. You should see the public IP address we got in the output.

Note

It might take a while for Azure to report the output, and if you did not get the IP addresses during `terraform apply`, you could subsequently run `terraform output` to get the details.

Ansible requires the control node to connect with managed nodes via SSH. Now, let's move on and look at how we can communicate with our managed nodes (also known as inventory servers).

Connecting the Ansible control node with inventory servers

We've already set up **passwordless SSH** between the control node and managed nodes when we provisioned the infrastructure using Terraform. Let's look at how we did that to understand it better.

We created an **Azure Virtual Network (VNet)**, a **subnet**, and three **Azure VMs** called `control-node`, `web`, and `db` within that subnet. If we look at the VM resource configuration, we also have a `custom_data` field that can be used to pass an initialization script to the VM, as follows:

```
resource "azurerm_virtual_machine" "control_node" {
  name          = "ansible-control-node"
  ...
}
```

```
os_profile {
    ...
    custom_data     = base64encode(data.template_file.control_node_init.rendered)
}
}
resource "azurerm_virtual_machine" "web" {
    name                  = "web"
    ...
    os_profile {
        ...
        custom_data     = base64encode(data.template_file.managed_nodes_init.rendered)
    }
}
resource "azurerm_virtual_machine" "db" {
    name                  = "db"
    ...
    os_profile {
        ...
        custom_data     = base64encode(data.template_file.managed_nodes_init.rendered)
    }
}
```

As we can see, the control_node VM refers to a data.template_file.control_node_init resource, and the web and db nodes refer to a data.template_file.managed_nodes_init resource. These are template_file resources that can be used for template files. Let's look at the resources as follows:

```
data "template_file" "managed_nodes_init" {
    template = file("managed-nodes-user-data.sh")
    vars = {
        admin_password = var.admin_password
    }
}
data "template_file" "control_node_init" {
    template = file("control-node-user-data.sh")
    vars = {
        admin_password = var.admin_password
    }
}
```

As we can see, the managed_nodes_init resource points to the managed-nodes-user-data.sh file and passes an admin_password variable to that file. Similarly, the control_node_init resource points to the control-node-user-data.sh file. Let's look at the managed-nodes-user-data.sh file first:

```
#!/bin/sh
sudo useradd -m ansible
echo 'ansible ALL=(ALL) NOPASSWD:ALL' | sudo tee -a /etc/sudoers
sudo su - ansible << EOF
ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
```

```
printf "${admin_password}\n${admin_password}" | sudo passwd ansible
EOF
```

As we can see, it is a shell script that does the following:

1. Creates an `ansible` user.
2. Adds the user to the `sudoers` list.
3. Generates an `ssh` key pair for passwordless authentication.
4. Sets the password for the `ansible` user.

As we've generated the `ssh` key pair, we would need to do the same within the control node with some additional configuration. Let's look at the `control-node-user-data.sh` script, as follows:

```
#!/bin/sh
sudo useradd -m ansible
echo 'ansible ALL=(ALL) NOPASSWD:ALL' | sudo tee -a /etc/sudoers
sudo su - ansible << EOF
ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
sleep 120
ssh-keyscan -H web >> ~/.ssh/known_hosts
ssh-keyscan -H db >> ~/.ssh/known_hosts
sudo apt update -y && sudo apt install -y sshpass
echo "${admin_password}" | sshpass ssh-copy-id ansible@web
echo "${admin_password}" | sshpass ssh-copy-id ansible@db
EOF
```

The script does the following:

1. Creates an `ansible` user
2. Adds the user to the `sudoers` list
3. Generates an `ssh` key pair for passwordless authentication
4. Adds the `web` and `db` VMs to the `known_hosts` file to ensure we trust both hosts
5. Installs the `sshpass` utility to allow for sending the `ssh` public key to the `web` and `db` VMs
6. Copies the `ssh` public key to the `web` and `db` VMs for passwordless connectivity

These files get executed automatically when the VMs are created; therefore, passwordless SSH should already be working. So, let's use an **SSH client** to log in to `ansible-control-node` using the IP address we got in the last step. We will use the username and password we configured in the `terraform.tfvars` file:

```
$ ssh ssh_admin@104.46.61.213
```

Once you are in the control node server, switch the user to `ansible` and try doing an SSH to the web server using the following commands:

```
$ sudo su - ansible  
$ ssh web
```

And if you land on the web server, passwordless authentication is working correctly.

Repeat the same steps to check whether you can connect with the db server.

Exit the prompts until you are in the control node.

Now, as we're in the control node, let's install Ansible.

Installing Ansible in the control node

Ansible requires a Linux/Unix machine (preferably), and you should have Python 2.x or 3.x installed.

As the Ansible control node runs on Ubuntu, Ansible provides a **personal package archive (PPA)** repository that we can configure to download and install Ansible using `apt` commands.

Use the following commands to install Ansible on the server:

```
$ sudo apt update  
$ sudo apt install software-properties-common -y  
$ sudo apt-add-repository --yes --update ppa:ansible/ansible  
$ sudo apt install ansible -y
```

To check whether Ansible has been installed successfully, run the following command:

```
$ ansible --version  
ansible 2.9.27
```

And, as we see, `ansible 2.9.27` is successfully installed on your control node.

Ansible uses an inventory file to manage nodes. Therefore, the next step is to set up an inventory file.

Setting up an inventory file

An inventory file within Ansible is a file that allows you to group your managed nodes according to roles. For example, you can define roles such as `webserver` and `dbserver` and group related servers together. You can use IP addresses, hostnames, or aliases for that.

Tip

Always use aliases because they provide room for IP address and hostname changes.

You can run Ansible commands on hosts or a group of hosts using the role tagged to them. There is no limit to servers that can have a particular role. If your server uses a non-standard SSH port, you can also use that port within the inventory file.

The default location of the Ansible inventory file is `/etc/ansible/hosts`. If you look at the `/etc/ansible` directory ownership, it is owned by the `root` user. We want to use the `ansible` user that we created for security purposes. Therefore, we must change the `/etc/ansible` directory ownership and its subdirectories and files to `ansible`. Use the following command to do so:

```
$ sudo chown -R ansible:ansible /etc/ansible
```

We can then switch the user to `ansible` and clone the Git repository that contains the required files into the control server using the following commands:

```
$ sudo su - ansible
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd ~/modern-devops/ch9/ansible-exercise
```

In our scenario, we have a web server called `web` and a database server called `db`. Therefore, if you check the host file called `hosts` within the repository, you will see the following:

```
[webservers]
web ansible_host=web
[dbservers]
db ansible_host=db
[all:vars]
ansible_python_interpreter=/usr/bin/python3
```

The `[all:vars]` section contains variables applicable to all groups. Here, we're explicitly defining `ansible_python_interpreter` to `python3` so that Ansible uses `python3` instead of `python2`. As we're using Ubuntu, `python3` comes installed as default, and `python2` is deprecated.

We also see that instead of using `web` directly, we've specified an `ansible_host` section. That defines `web` as an alias, pointing to a host with the hostname `web`. You can also use the IP address instead of the hostname if required.

Tip

Always group the inventory according to the function performed. That helps us to apply a similar configuration to a large number of machines with a similar role.

As we want to keep the configuration with code, we would wish to stay within the Git repository itself. So, we must tell Ansible that the inventory file is in a non-standard location. To do so, we will create an Ansible configuration file.

Setting up the Ansible configuration file

The Ansible configuration file defines global properties that are specific to our setup. The following are ways in which you can specify the Ansible configuration file, and the first method overrides the next – the settings are not merged, so keep that in mind:

- By setting an environment variable, `ANSIBLE_CONFIG`, pointing to the Ansible configuration file
- By creating an `ansible.cfg` file in the current directory
- By creating an `ansible.cfg` file in the home directory of the current user
- By creating an `ansible.cfg` file in the `/etc/ansible` directory

Tip

If you manage multiple applications, with each application in its Git repositories, having a local `ansible.cfg` file in every repository will help keep the applications decentralized. It will also enable GitOps and make Git the **single source of truth**.

So, if you check the `ansible.cfg` file in the current directory, you will see the following:

```
[defaults]
inventory = ./hosts
host_key_checking = False
```

Now, to check whether our inventory file is correct, let's list our inventory by using the following command:

```
$ ansible-inventory --list -y
all:
  children:
    dbservers:
      hosts:
        db:
          ansible_host: db
          ansible_python_interpreter: /usr/bin/python3
    ungrouped: {}
    webservers:
      hosts:
        web:
          ansible_host: web
          ansible_python_interpreter: /usr/bin/python3
```

We see that there are two groups—`dbservers` containing `db` and `webservers` containing `web`, each using `python3` as the `ansible_python_interpreter`.

If we want to see all the hosts, we can use the following command:

```
$ ansible --list-hosts all
hosts (2):
```

```
web
db
```

If we want to list all hosts that have the `webservers` role, we can use the following command:

```
$ ansible --list-hosts webservers
hosts (1):
    web
```

Now, let's check whether Ansible can connect to these servers by using the following command:

```
$ ansible all -m ping
web | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
db | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

And, as we can observe, we get a successful response for both servers. So, we're all set up and can start defining the configuration. Ansible offers **tasks** and **modules** to provide CM. Let's look at these in the next section.

Ansible tasks and modules

Ansible tasks form the basic building block of running Ansible commands. Ansible tasks are structured in the following format:

```
$ ansible <options> <inventory>
```

Ansible modules are reusable code for a particular function, such as running a `shell` command or creating and managing users. You can use Ansible modules with Ansible tasks to manage configuration within managed nodes. For example, the following command will run the `uname` command on each managed server:

```
$ ansible -m shell -a "uname" all
db | CHANGED | rc=0 >>
Linux
web | CHANGED | rc=0 >>
Linux
```

So, we get a reply from the `db` server and the `web` server, each providing a return code, 0, and an output, `Linux`. If you look at the command, you will see that we have provided the following flags:

- `-m`: The name of the module (`shell` module here)
- `-a`: The parameters to the module (`uname` in this case)

The command finally ends with where we want to run this task. Since we've specified `all`, it runs the task on all servers. We can run this on a single server, a set of servers, a role, or multiple roles, or use a wildcard to select the combination we want.

The tasks have three possible statuses—`SUCCESS`, `CHANGED`, and `FAILURE`. The `SUCCESS` status denotes that the task was successful, and Ansible took no action. The `CHANGED` status denotes that Ansible had to change the existing configuration to apply the expected configuration, and `FAILURE` denotes an error while executing the task.

Ansible modules are reusable scripts that we can use to define configuration within servers. Each module targets a particular aspect of CM. Modules are used in both Ansible tasks and playbooks. There are many modules available for consumption, and they are available at https://docs.ansible.com/ansible/latest/collections/index_module.html. You can pick and choose modules according to your requirements and use cases.

Tip

As Ansible is idempotent, always use modules specific to your task and avoid using `command` and `shell` modules. For example, use the `apt` module to install a package instead of the `command` module to run `apt install <package> -y`. If your playbook starts to look like code, then you're doing something fundamentally wrong.

Tasks do not make sense when we have a series of steps to follow while setting up a server. Therefore, Ansible provides *playbooks* for this activity. Let's have a look at this in the next section.

Introduction to Ansible playbooks

Imagine you're a conductor leading an orchestra. In this scenario, Ansible playbooks are akin to your musical score, guiding every musician to create a harmonious symphony of automation in the tech world.

In the realm of tech and automation, Ansible playbooks provide the following:

- **Musical score for automation:** Just as a conductor uses a musical score with notations to guide each instrument, an Ansible playbook contains a set of instructions and actions for orchestrating specific IT tasks and configurations, spanning from software deployments to system configurations.
- **Harmonious guidance:** Ansible playbooks take a similar approach. You declare the desired IT state, and Ansible plays the role of the conductor, ensuring that all the necessary steps are followed, much like specifying, "*I want a flawless musical performance*," and Ansible orchestrates the entire process.
- **Tasks and reusability:** Ansible playbooks are organized into tasks and roles, as with musical sheets and instruments. These tasks can be reused across various playbooks, promoting consistency and saving time.

- **Instrument selection and direction:** Just as a conductor selects which instruments play at which times, playbooks specify which servers or machines (the inventory) should execute tasks. You can direct specific server groups or individual machines.
- **Harmonious execution:** Ansible can skilfully coordinate tasks on multiple machines simultaneously, much as a conductor harmonizes the efforts of different musicians to create a beautiful composition.
- **Fine-tuned performance:** If unexpected challenges arise during the performance, a conductor adjusts and guides the musicians to ensure a flawless outcome. Similarly, Ansible playbooks incorporate error-handling strategies to handle unexpected issues during automation.

Ansible playbooks are a collection of tasks that produce the desired configuration within the managed nodes. They have the following features:

- They help in managing configuration within multiple remote servers using declarative steps
- They use a sequential list of idempotent steps, and steps that match the expected configuration are not applied again
- Tasks within the playbook can be synchronous and asynchronous
- They enable GitOps by allowing the steps to be stored using a simple YAML file to keep in source control, providing CaC

Ansible playbooks consist of multiple **plays**, and each play is mapped to a group of **hosts** using a **role** and consists of a series of **tasks** required to achieve them—something like the following diagram:

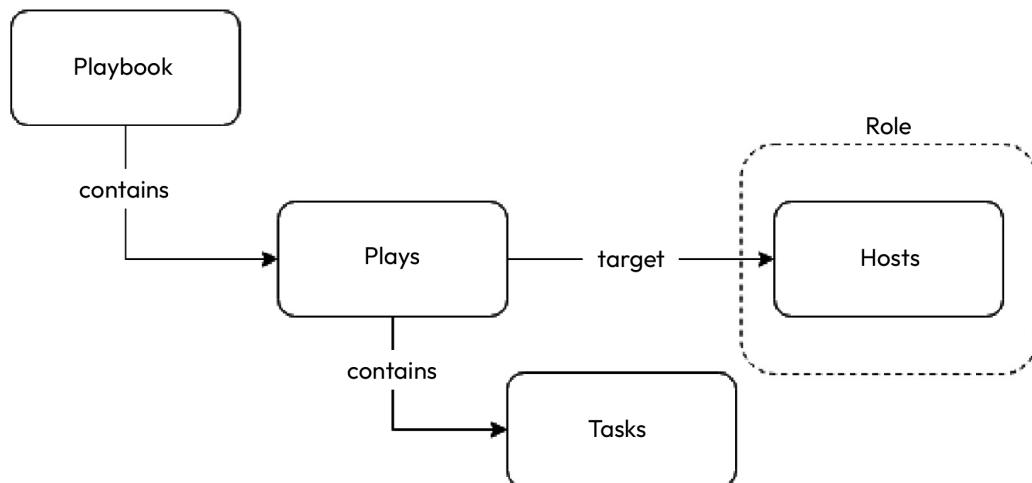


Figure 9.2 – Playbooks

The following `ping.yaml` file is an example of a simple playbook that pings all servers:

```
---
- hosts: all
  tasks:
    - name: Ping all servers
      action: ping
```

The YAML file contains a list of plays, as the list directive shows. Each play consists of a `hosts` attribute that defines the role to which we want to apply the play. The `tasks` section consists of a list of tasks, each with `name` and `action` attributes. In the preceding example, we have a single play with a single task that pings all servers.

Checking playbook syntax

It is a best practice to check playbook syntax before applying it to your inventory. To check your playbook's syntax, run the following command:

```
$ ansible-playbook ping.yaml --syntax-check
playbook: ping.yaml
```

The syntax is correct, as we get a response with the playbook name. Now, let's go ahead and apply the playbook.

Applying the first playbook

To apply the playbook, run the following command:

```
$ ansible-playbook ping.yaml
PLAY [all] ****
TASK [Gathering Facts] ****
ok: [db]
ok: [web]
TASK [Ping all servers] ****
ok: [db]
ok: [web]
PLAY RECAP ****
db : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
web : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

There are three elements of play execution:

- **Gathering facts:** Ansible checks for all hosts that are part of the role, logs in to each instance, and gathers information from each host it uses while executing the tasks from the plays.
- **Run tasks:** Then, it runs the tasks of each play, as defined in the playbook.
- **Play recap:** Ansible then provides a recap of the tasks it executed and the hosts it ran them on. This includes a list of all successful and failed responses.

As we've investigated an elementary example of playbooks, we must understand how to use Ansible playbooks effectively. In the next section, let's look at Ansible playbooks in action with a better example.

Ansible playbooks in action

Let's set up an Apache server for a custom website that connects with a MySQL backend—in short, a **Linux, Apache, MySQL, and PHP (LAMP)** stack using Ansible.

The following directory contains all resources for the exercises in this section:

```
$ cd ~/modern-devops/ch9/lamp-stack
```

We have created the following custom `index.php` page that tests the connection to the MySQL database and displays whether it can connect or not:

```
...
<?php
mysqli_connect('db', 'testuser', 'Password@1') or die('Could not connect the database :
Username or password incorrect');
echo 'Database Connected successfully';
?>
...
```

We create several Ansible playbooks according to the logical steps we follow with CM.

It is an excellent practice to update the packages and repositories at the start of every configuration. Therefore, we need to start our playbook with this step.

Updating packages and repositories

As we're using Ubuntu, we can use the `apt` module to update the packages. We must update packages and repositories to ensure the latest package index is available with all our `apt` repositories and avoid any untoward issues while installing packages. The following playbook, `apt-update.yaml`, performs the update:

```
---
- hosts: webservers:dbservers
  become: true
  tasks:
    - name: Update apt packages
      apt: update_cache=yes cache_valid_time=3600
```

The YAML file begins with a list of plays and contains a single play in this case. The `hosts` attribute defines a colon-separated list of `roles/hosts` inventory to apply the playbook. In this case, we've specified `webservers` and `dbservers`. The `become` attribute specifies whether we want to execute the play as a `root` user. So, as we've set `become` to `true`, Ansible will perform all play tasks with `sudo` privileges. The play contains a single task—`Update apt packages`. The task uses the `apt`

module and consists of `update_cache=yes`. It will run an `apt update` operation on all nodes with the `webservers` and `dbservers` roles. The next step is to install packages and services.

Installing application packages and services

We will use the `apt` module to install the packages on Ubuntu, and the `service` module to start and enable the service.

Let's start by installing Apache on the web servers using the following `install-webserver.yaml` playbook:

```
---
- hosts: webservers
  become: true
  tasks:
    - name: Install packages
      apt:
        name:
          - apache2
          - php
          - libapache2-mod-php
          - php-mysql
        update_cache: yes
        cache_valid_time: 3600
        state: present
    - name: Start and Enable Apache service
      service: name=apache2 state=started enabled=yes
```

As this configuration is for `webservers`, we've specified that within the `hosts` attribute. The `tasks` section defines two tasks—`Install packages` and `Start and Enable Apache service`. The `Install packages` task uses the `apt` module to install `apache2`, `php`, `libapache2-mod-php`, and `php-mysql`. The `Start and Enable Apache service` task will start and enable the `apache2` service.

Similarly, we will install and set up the MySQL service using the following `install-dbserver.yaml` playbook:

```
---
- hosts: dbservers
  become: true
  tasks:
    - name: Install packages
      apt:
        name:
          - python-pymysql
          - mysql-server
        update_cache: yes
        cache_valid_time: 3600
        state: present
```

```

- name: Start and enable MySQL service
  service:
    name: mysql
    state: started
    enabled: true

```

This playbook will run two tasks—Install packages and Start and enable MySQL service. The Install packages task will install the python-mysql and mysql-server packages using the apt module. The Start and enable MySQL service task will start and enable the MySQL service.

Configuring applications

The next step in the chain is to configure the applications. There are two playbooks for this. The first will configure Apache on webservers, and the second will configure MySQL on dbservers.

The following setup-webservers.yaml playbook will configure webservers:

```

---
- hosts: webservers
  become: true
  tasks:
    - name: Delete index.html file
      file:
        path: /var/www/html/index.html
        state: absent
    - name: Upload application file
      copy:
        src: index.php
        dest: /var/www/html
        mode: 0755
      notify:
        - Restart Apache
  handlers:
    - name: Restart Apache
      service: name=apache2 state=restarted

```

This playbook runs on all nodes with the webservers role, and there are three tasks in this playbook. The Delete index.html file task uses the file module to delete the /var/www/html/index.html file from the web server. That is because we are using index.php as the index page and not index.html. The Upload application file task then uses the copy module to copy the index.php file from the Ansible control node to the web server at the /var/www/html destination, with a mode of 0755. The Upload application file task also has a notify action that will call the Restart Apache handler if this task has a status of CHANGED. A handlers section within the playbook defines handlers that listen to notify events. In this scenario, if there is a change in the Upload application file task, the Restart Apache handler will be triggered and will restart the apache2 service.

We will use the following `setup-dbservers.yaml` playbook to configure MySQL on dbservers:

```
---
- hosts: dbservers
  become: true
  vars:
    mysql_root_password: "Password@1"
  tasks:
    - name: Set the root password
      copy:
        src: client.my.cnf
        dest: "/root/.my.cnf"
        mode: 0600
      notify:
        - Restart MySQL
    - name: Create a test user
      mysql_user:
        name: testuser
        password: "Password@1"
        login_user: root
        login_password: "{{ mysql_root_password }}"
        state: present
        priv: '*.*:ALL,GRANT'
        host: '%'
    - name: Remove all anonymous user accounts
      mysql_user:
        name: ''
        host_all: yes
        state: absent
        login_user: root
        login_password: "{{ mysql_root_password }}"
      notify:
        - Restart MySQL
    - name: Remove the MySQL test database
      mysql_db:
        name: test
        state: absent
        login_user: root
        login_password: "{{ mysql_root_password }}"
      notify:
        - Restart MySQL
    - name: Change bind address
      lineinfile:
        path: /etc/mysql/mysql.conf.d/mysqld.cnf
        regexp: ^bind-address
        line: 'bind-address          = 0.0.0.0'
      notify:
        - Restart MySQL
  handlers:
    - name: Restart MySQL
      service: name=mysql state=restarted
```

This playbook is a bit more complicated, but let's break it down into parts to facilitate our understanding.

There is a `vars` section in this playbook that defines a `mysql_root_password` variable. We need this password while executing MySQL tasks. The first task is to set up the root password. The best way to set that up is by defining a `/root/.my.cnf` file within MySQL that contains the root credentials. We are copying the following `client.my.cnf` file to `/root/.my.cnf` using the `copy` module:

```
[client]
user=root
password=Password@1
```

Then, the `Create a test user` task uses the `mysql_user` module to create a user called `testuser`. It requires values for the `login_user` and `login_password` attributes, and we are supplying `root` and `{ mysql_root_password }`, respectively. It then goes ahead and removes all anonymous users and also removes the `test` database. It then changes the bind address to `0.0.0.0` using the `lineinfile` module. The `lineinfile` module is a powerful module that helps manipulate files by first grepping a file using a regex and then replacing those lines with the `line` attribute's value. All these tasks notify the `Restart MySQL` handler that restarts the MySQL database service.

Combining playbooks

As we've written multiple playbooks, we need to execute them in order. We cannot configure the services before installing packages and services, and there is no point in running an `apt update` after installing the packages. Therefore, we can create a playbook of playbooks.

To do so, we've created a YAML file, `playbook.yaml`, that has the following content:

```
---
- import_playbook: apt-update.yaml
- import_playbook: install-webserver.yaml
- import_playbook: install-dbserver.yaml
- import_playbook: setup-webservers.yaml
- import_playbook: setup-dbservers.yaml
```

This YAML file contains a list of plays, and every play contains an `import_playbook` statement. The plays are executed in order as specified in the file. Now, let's go ahead and execute the playbook.

Executing playbooks

Executing the playbook is simple. We will use the `ansible-playbook` command followed by the playbook YAML file. As we've combined playbooks in a `playbook.yaml` file, the following command will run the playbook:

```
$ ansible-playbook playbook.yaml
PLAY [webservers:dbservers] *****
```

```
TASK [Gathering Facts] ****
ok: [web]
ok: [db]
TASK [Update apt packages] ****
ok: [web]
ok: [db]
PLAY [webservers] ****
TASK [Gathering Facts] ****
ok: [web]
TASK [Install packages] ****
changed: [web]
TASK [Start and Enable Apache service] ****
ok: [web]
PLAY [dbservers] ****
TASK [Gathering Facts] ****
ok: [db]
TASK [Install packages] ****
changed: [db]
TASK [Start and enable MySQL service] ****
ok: [db]
PLAY [webservers] ****
TASK [Gathering Facts] ****
ok: [web]
TASK [Delete index.html file] ****
changed: [web]
TASK [Upload application file] ****
changed: [web]
RUNNING HANDLER [Restart Apache] ****
changed: [web]
PLAY [dbservers] ****
TASK [Gathering Facts] ****
ok: [db]
TASK [Set the root password] ****
changed: [db]
TASK [Update the cnf file] ****
changed: [db]
TASK [Create a test user] ****
changed: [db]
TASK [Remove all anonymous user accounts] ****
ok: [db]
TASK [Remove the MySQL test database] ****
ok: [db]
TASK [Change bind address] ****
changed: [db]
RUNNING HANDLER [Restart MySQL] ****
changed: [db]
PLAY RECAP ****
db: ok=13    changed=6      unreachable=0      failed=0
skipped=0    rescued=0     ignored=0
web: ok=9     changed=4      unreachable=0      failed=0
skipped=0    rescued=0     ignored=0
```

As we can see, the configuration is applied on both web servers and db servers, so let's run a curl command to the web server to see what we get:

```
$ curl web
<html>
<head>
<title>PHP to MySQL</title>
</head>
<body>Database Connected successfully</body>
</html>
```

As we can see, the database is connected successfully! That proves that the setup was successful.

There are several reasons why the way we approached the problem was not the best. First, there are several sections within the playbook where we've hardcoded values. While we have used variables in a few playbooks, we've also assigned values to variables within them. That does not make the playbooks a candidate for reuse. The best way to design software is to keep reusability in mind. Therefore, there are many ways in which we can redesign the playbooks to foster reusability.

Designing for reusability

Ansible provides variables for turning Ansible playbooks into reusable templates. You can substitute variables in the right places using **Jinja2** markup, which we've already used in the last playbook. Let's now look at Ansible variables, their types, and how to use them.

Ansible variables

Ansible variables, as with any other variables, are used to manage differences between managed nodes. You can use a similar playbook for multiple servers, but sometimes, there are some differences in configuration. Ansible variables help you template your playbooks so that you can reuse them for a variety of similar systems. There are multiple places where you can define your variables:

- Within the Ansible playbook within the `vars` section
- In your inventory
- In reusable files or roles
- Passing variables through the command line
- Registering variables by assigning the return values of a task

Ansible variable names can include *letters*, *numbers*, and *underscores*. You cannot have a Python *keyword* as a variable, as Ansible uses Python in the background. Also, a variable name cannot begin with a number but can start with an underscore.

You can define variables using a simple key-value pair within the YAML files and following the standard YAML syntax.

Variables can broadly be of three types—*simple variables*, *list variables*, and *dictionary variables*.

Simple variables

Simple variables are variables that hold a single value. They can have *string*, *integer*, *double*, or *boolean* values. To refer to simple Ansible variables within the playbook, use them within Jinja expressions, such as `{ { var_name } }`. You should always quote Jinja expressions, as the YAML files will fail to parse without that.

The following is an example of a simple variable declaration:

```
mysql_root_password: bar
```

And this is how you should reference it:

```
- name: Remove the MySQL test database
  mysql_db:
    name: test
    state: absent
    login_user: root
    login_password: "{{ mysql_root_password }}"
```

Now, let's look at list variables.

List variables

List variables hold a list of values you can reference using an index. You can also use list variables within loops. To define a list variable, you can use the standard YAML syntax for a list, as in the following example:

```
region:
  - europe-west1
  - europe-west2
  - europe-west3
```

To access the variable, we can use the index format, as in this example:

```
region: " {{ region[0] }} "
```

Ansible also supports more complex dictionary variables. Let's have a look.

Dictionary variables

Dictionary variables hold a complex combination of *key-value pairs*, the same as a Python dictionary. You can define dictionary variables using the standard YAML syntax, as in the following example:

```
foo:
  bar: one
  baz: two
```

There are two ways in which to refer to these variables' values. For example, in dot notation, we can write the following:

```
bar: {{ foo.bar }}
```

And in bracket notation, we can depict the same thing using the following expression:

```
bar: {{ foo[bar] }}
```

We can use either dot or bracket notation in the same way as in Python.

Tip

While both dot and bracket notation signify the same thing, bracket notation is better. With dot notation, some keys can collide with the methods and attributes of Python dictionaries.

Now, let's look at ways of sourcing variable values.

Sourcing variable values

While you can manually define variables and provide their values, sometimes we need dynamically generated values; for example, if we need to know the server's hostname where Ansible is executing the playbook or want to use a specific value returned from a task within a variable. Ansible provides a list of variables and system metadata during the gathering facts phase for the former requirement. That helps determine which variables are available and how to use them. Let's understand how we can gather that information.

Finding metadata using Ansible facts

Ansible facts are metadata information associated with the managed nodes. Ansible gets the facts during the *gathering facts* stage, and we can use the `facts` variable directly within the playbook. We can use the `setup` module as an Ansible task to determine the facts. For example, you can run the following command to get the Ansible facts for all nodes with the `webservers` role:

```
$ ansible -m setup webservers
web | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
```

```
    "10.0.2.5"
],
...
"ansible_hostname": "web",
```

So, as we can see, we get `ansible_facts` with multiple variables associated with the inventory item. As we have a single server here, we get web server details. Within the piece, we have an `ansible_hostname` attribute called `web`. We can use that `ansible_hostname` attribute within our playbook if we need to.

Sometimes, we want to source a task's output to a particular variable to use the variable in any subsequent tasks of the playbook. Let's look at how we can do that.

Registering variables

If a task within your playbook, for example, needs a value from the result of a preceding task, we can use the `register` attribute.

The following directory contains all the resources for exercises in this section:

```
$ cd ~/modern-devops/ch9/vars-exercise
```

Let's look at the following example `register.yaml` file:

```
- hosts: webservers
  tasks:
    - name: Get free space
      command: free -m
      register: free_space
      ignore_errors: true
    - name: Print the free space from the previous task
      debug:
        msg: "{{ free_space }}"
```

The playbook contains two tasks. The first task uses the `command` module to execute a command, `free -m`, and registers the result in the `free_space` variable. The subsequent task uses the previous task's output using the `debug` module to print `free_space` as a message to the console.

Let's run the playbook to see for ourselves:

```
$ ansible-playbook register.yaml
PLAY [webservers] ****
TASK [Gathering Facts] ****
ok: [web]
TASK [Get free space] ****
changed: [web]
TASK [Print the free space from the previous task] ***
ok: [web] => {
  "msg": {
```

```

    "stdout": "              total          used
free      shared  buff/cache   available\nMem:           3.3G
170M       2.6G       2.2M       642M       3.0G\nSwap:
 0B        0B        0B",
}
PLAY RECAP ****
web: ok=3    changed=1    unreachable=0    failed=0    skipped=0
      rescued=0   ignored=0

```

Now that we've understood variables, let's look at other aspects that will help us improve the last playbook.

Jinja2 templates

Ansible allows for templating files using dynamic Jinja2 templates. You can use the Python syntax within the file, starting with { { and ending with } }. That will allow you to substitute variables during runtime and run complex computations on variables.

To understand this further, let's modify the `index.php` file to supply the MySQL username and password dynamically during execution:

```

...
<?php
mysqli_connect('db', '{{ mysql_user }}', '{{ mysql_password }}')
or die('Could not connect the database : Username or password
incorrect');
echo 'Database Connected successfully';
?>
...

```

As we can see, instead of hardcoding the username and password, we can use templates to substitute the variable values during runtime. That will make the file more reusable and will fit multiple environments. Ansible provides another important aspect of coding reusability within your playbooks—Ansible **roles**. Let's have a look at this in the next section.

Ansible roles

Well, the last playbook looks a bit cluttered. You have a lot of files within it, and none of them are reusable. The code we've written can only set up the configuration in a particular way. This may work fine for smaller teams with limited configurations to manage, but it is not as simple as it looks for most enterprises.

Ansible roles help to standardize an Ansible setup and promote reusability. With roles, you can automatically load **var files**, **handlers**, **tasks**, and other Ansible artifacts using a standard directory structure relative to your playbooks. The directory structure is as follows:

```

<playbook>.yaml
  roles/
    <role>/
      tasks/

```

```
handlers/
library/
files/
templates/
vars/
defaults/
meta/
```

The `roles` directory contains multiple subdirectories for each role. Each role directory contains multiple standard directories:

- `tasks`: This directory contains a list of tasks' YAML files. It should contain a file called `main.yaml` (or `main.yml` or `main`), containing an entire list of tasks or importing tasks from other files within the directory.
- `handlers`: This directory contains a list of handlers associated with the role within a file called `main.yaml`.
- `library`: This directory contains Python modules that can be used with the role.
- `files`: This directory contains all files that we require for our configuration.
- `templates`: This directory contains the Jinja2 templates that the role deploys.
- `vars`: This directory contains a `main.yaml` file with a list of variables associated with the role.
- `defaults`: This directory contains a `main.yaml` file containing the default variables associated with the role that can be easily overridden by any other variable that includes inventory variables.
- `meta`: This directory contains the metadata and dependencies associated with the role within a `main.yaml` file.

Some best practices revolve around managing your Ansible configuration through the folder structure. Let's look at some of these next.

Tip

While choosing between the `vars` and `defaults` directories, the rule of thumb is to put variables that will not change within the `vars` directory. Put variables that are likely to change within the `defaults` directory.

So, we'll go and use the `defaults` directory as much as we can. There are some best practices regarding roles that we should follow as well. Let's look at some of them.

Tip

Think about the full life cycle of a specific service while designing roles rather than building the entire stack—in other words, instead of using `lamp` as a role, use `apache` and `mysql` roles instead.

We will create three roles for our use—common, apache, and mysql.

Tip

Use specific roles, such as apache or mysql, instead of using webserver or dbserver. Typical enterprises have a mix and match of multiple web servers and database technologies. Therefore, giving a generic name to a role will confuse things.

The following directory contains all the resources for the exercises in this section:

```
$ cd ~/modern-devops/ch9/lamp-stack-roles
```

The following is the directory structure we will follow for our scenario:

```
└── ansible.cfg
└── hosts
└── output.log
└── playbook.yaml
```

There are three roles that we'll create—apache, mysql, and common. Let's look at the directory structure of the apache role first:

```
└── roles
    └── apache
        ├── defaults
        │   └── main.yaml
        ├── handlers
        │   └── main.yaml
        ├── tasks
        │   ├── install-apache.yaml
        │   ├── main.yaml
        │   └── setup-apache.yaml
        └── templates
            └── index.php.j2
```

There is also a common role that will apply to all scenarios. The following directory structure defines that:

```
└── common
    └── tasks
        └── main.yaml
```

Finally, let's define the mysql role through the following directory structure:

```
└── mysql
    ├── defaults
    │   └── main.yaml
    ├── files
    ├── handlers
    │   └── main.yaml
    └── tasks
```

```
|   ├── install-mysql.yaml
|   ├── main.yaml
|   └── setup-mysql.yaml
└── templates
    └── client.my.cnf.j2
```

The apache directory consists of the following:

- We've used the same `index.php` file we created in the last exercise, converted it to a Jinja2 template called `index.php.j2`, and copied it to `roles/apache/templates`.
- The `handlers` directory contains a `main.yaml` file that contains the `Restart Apache` handler.
- The `tasks` directory contains an `install-apache.yaml` file that includes all tasks required to install Apache. The `setup-apache.yaml` file consists of a list of tasks that will set up Apache, similar to what we did in the previous exercise. The `main.yaml` file contains tasks from both files, using `include` directives such as the following:

```
---
- include: install-apache.yaml
- include: setup-apache.yaml
```

- The `defaults` directory contains the `main.yaml` file, which contains the `mysql_username` and `mysql_password` variables and their default values.

Tip

Use as few variables as possible and try to default them. Use defaults for variables in such a way that minimal custom configuration is needed.

The mysql directory consists of the following:

- We've modified `client.my.cnf` and converted that to a `j2` file. The `j2` file is a Jinja2 template file we will use in the role through the `template` module in the `Set the root password` task. The file exists within the `templates` directory:

```
[client]
user=root
password={{ mysql_root_password }}
```

As we can see, we're providing the password through a Jinja2 expression. When we run the `mysql` role through the playbook, the value of `mysql_root_password` will be substituted in the `password` section.

- The `handlers` directory contains the `Restart MySQL` handler.

- The `tasks` directory consists of three files. The `install-mysql.yaml` file contains tasks that install mysql, and the `setup-mysql.yaml` file contains tasks that set up mysql. The `main.yaml` file combines both these files using `include` task directives, as follows:

```
---
- include: install-mysql.yaml
- include: setup-mysql.yaml
```

- The `defaults` directory contains a `main.yaml` file with a list of variables we will use within the role. In this case, it just contains the value of `mysql_root_password`.

The `common` directory contains a single directory called `tasks` that includes a `main.yaml` file with a single task to run an `apt update` operation.

The main directory contains `ansible.cfg`, `hosts`, and `playbook.yaml` files. While the `hosts` and `ansible.cfg` files are the same as the last exercise, the `playbook.yaml` file looks like the following:

```
---
- hosts: webservers
  become: true
  roles:
    - common
    - apache
- hosts: dbservers
  become: true
  roles:
    - common
    - mysql
```

The playbook is now a concise one with a lot of reusable elements. It consists of two plays. The first play will run on all web servers with the `root` user and apply `common` and `apache` roles to them. The second play will run on all nodes with the `dbservers` role with the `root` user and use `common` and `mysql` roles.

Tip

Always keep roles loosely coupled. In the preceding example, the `apache` role has no dependency on `mysql` and vice versa. This will allow us to reuse configuration with ease.

Now, let's go ahead and execute the playbook:

```
$ ansible-playbook playbook.yaml
PLAY [webservers]
...
PLAY [dbservers]
...
PLAY RECAP
```

```
db: ok=10 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
web: ok=7 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

And, as we can see, there are no changes to the configuration. We've applied the same configuration but in a better way. If we want to share our configuration with people within the team, we can share the `roles` directory, and they can apply the role within their playbook.

There may be instances where we want to use a different value for the variable defined in the `roles` section. You can override variables within the playbook by supplying the variable values with the `extra-vars` flag, as follows:

```
$ ansible-playbook playbook.yaml --extra-vars "mysql_user=foo mysql_password=bar@123"
```

When we apply the playbook using the preceding command, we'll see that the user now changes to `foo` and that the password changes to `bar@123` in both the Apache and MySQL configurations:

```
...
PLAY RECAP
db: ok=9 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
web: ok=7 changed=2 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

So, if we run the `curl` command to the web host, we will get the same response as before:

```
...
<body>Database Connected successfully</body>
...
```

Our setup is working correctly with roles. We've set up the Ansible playbook by following all the best practices and using reusable roles and templates. That is the way to go forward in designing powerful Ansible playbooks.

Summary

In this chapter, we've discussed Ansible and its core functionalities from a hands-on perspective. We began by understanding CaC, looked at Ansible and Ansible architecture, installed Ansible, understood Ansible modules, tasks, and playbooks, and then applied our first Ansible configuration. We then looked at fostering reusability with Ansible variables, Jinja2 templates, and roles and reorganized our configuration with reusability in mind. We also looked at several best practices along the way.

In the next chapter, we will combine Terraform with Ansible to spin up something useful and look at HashiCorp's Packer to create immutable infrastructure.

Questions

1. It is a best practice to avoid using `command` and `shell` modules as much as possible. (True/False)
2. Aliases help in keeping your inventory generic. (True/False)
3. What does the `ansible-playbook` command do?
 - A. It runs an ad hoc task on the inventory.
 - B. It runs a series of tasks on the inventory.
 - C. It applies the plays and tasks configured with the playbook.
 - D. It destroys the configuration from managed nodes.
4. Which of the following techniques helps in building reusability within your Ansible configuration? (Choose three)
 - A. Use variables.
 - B. Use Jinja2 templates.
 - C. Use roles.
 - D. Use tasks.
5. While naming roles, what should we consider? (Choose two)
 - A. Name roles as precisely as possible.
 - B. While thinking of roles, think of the service instead of the full stack.
 - C. Use generic names for roles.
6. In which directory should you define variables within roles if the variable's value is likely to change?
 - A. `defaults`
 - B. `vars`
7. Handlers are triggered when the output of the task associated with the handler is ...?
 - A. SUCCESS
 - B. CHANGED
 - C. FAILED
8. Does a SUCCESS status denote that the task did not detect any changed configuration? (True/False)
9. What are the best practices for inventory management? (Choose three)
 - A. Use a separate inventory for each environment.
 - B. Group the inventory by functions.

- C. Use aliases.
- D. Keep the inventory file in a central location.

Answers

1. True
2. True
3. C
4. A, B, C
5. A, B
6. A
7. B
8. True
9. A, B, and C

10

Immutable Infrastructure with Packer

In the previous chapter, we looked at configuration management with Ansible and the tool's core concepts. We also discussed Terraform and IaC in *Chapter 8, Infrastructure as Code (IaC) with Terraform*. In this chapter, we will look at another way of provisioning your infrastructure and configuration using both tools, as well as another one, called **Packer**. With all three tools, let's boot up a scalable **Linux, Apache, MySQL, and PHP (LAMP)** stack on Azure.

In this chapter, we're going to cover the following main topics:

- Immutable infrastructure with HashiCorp's Packer
- Creating the Apache and MySQL playbook
- Building the Apache and MySQL images using Packer and Ansible provisioners
- Creating the required infrastructure with Terraform

Technical requirements

You will need an active Azure subscription to follow the exercises for this chapter. Currently, Azure is offering a free trial for 30 days with \$200 worth of free credits; sign up at <https://azure.microsoft.com/en-in/free>.

You will also need to clone the following GitHub repository for some of the exercises:

<https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>

Run the following command to clone the repository into your home directory, and `cd` into the `ch10` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
  modern-devops
$ cd modern-devops/ch10
```

You also need to install **Terraform** and **Ansible** on your system. Refer to *Chapter 8, Infrastructure as Code (IaC) with Terraform*, and *Chapter 9, Configuration Management with Ansible*, for more details on installing and setting up Terraform and Ansible.

Immutable infrastructure with HashiCorp's Packer

Imagine you are the author of a book and you need to make changes to an existing edition. When you want to make changes, such as improving the content or fixing the issues and ensuring the book is up to date, you don't edit the existing book. Instead, you create a new edition with the desired updates while keeping the existing editions intact, like the new edition of this book. This concept aligns with **immutable infrastructure**.

In IT and systems management, immutable infrastructure is a strategy where, instead of making changes to existing servers or **Virtual Machines (VMs)**, you generate entirely new instances with the desired configuration. These new instances replace the old ones instead of modifying them, like creating a new book edition when you want to incorporate changes.

Here's how it works:

- **Building from scratch:** When you need to update a part of your infrastructure, you avoid making direct changes to the existing servers or machines. Instead, you create new ones from a pre-established template (an image) that includes the updated configuration.
- **No in-place modifications:** Like not editing an existing book, you avoid making in-place modifications to current servers. This practice reduces the risk of unforeseen changes or configuration inconsistencies.
- **Consistency:** Immutable infrastructure ensures that every server or instance is identical because they all originate from the same template. This uniformity is valuable for ensuring reliability and predictability.
- **Rolling updates:** When it's time to implement an update, you systematically replace the old instances with the new ones in a controlled manner. This minimizes downtime and potential risks.
- **Scalability:** Scaling your infrastructure becomes effortless by generating new instances as needed. This is akin to publishing new book editions when there's a surge in demand, or things become outdated.
- **Rollback and recovery:** If issues arise from an update, you can swiftly revert to the previous version by re-creating instances from a known good template.

So, consider immutable infrastructure as a means of maintaining your infrastructure by creating new, improved instances rather than attempting to revise or modify existing ones. This approach elevates consistency, reliability, and predictability within your IT environment.

To understand this further, let's consider the traditional method of setting up applications via Terraform and Ansible. We would use Terraform to spin up the infrastructure and then use Ansible on top to apply the relevant configuration to the infrastructure. That is what we did in the last chapter. While that is a viable approach, and many enterprises use it, there is a better way to do it with modern DevOps approaches and immutable infrastructure.

Immutable infrastructure is a ground-breaking concept that emerged due to the problems with **mutable infrastructure**. In a mutable infrastructure approach, we generally update servers in place. So, we follow a mutable process when we install Apache in a VM using Ansible and customize it further. We may want to update the servers, patch them, update our Apache to a newer version, and update our application code from time to time.

The issue with this approach is that while we can manage it well with Ansible (or related tools, such as **Puppet**, **Chef**, and **SaltStack**), the problem always remains that we are making live changes in a production environment that might go wrong for various reasons. Worse, it might update something we did not anticipate or test in the first place. We also might end up in a partial upgrade state that might be difficult to roll back.

With the scalable infrastructure that the cloud provides, you can have a dynamic horizontal scaling model where VMs scale with traffic. Therefore, you can have the best possible utilization of your infrastructure – the best bang for your buck! The problem with the traditional approach is that even if we use Ansible to apply the configuration to new machines, it is slower to get ready. Therefore, the scaling is not optimal, especially for bursty traffic.

Immutable infrastructure helps you manage these problems by taking the same approach we took for containers – *baking configuration directly into the OS image using modern DevOps tools and practices*. Immutable infrastructure helps you deploy the tested configuration to production by replacing the existing VM without doing any updates in place. It is faster to start and easy to roll back. You can also version infrastructure changes with this approach.

HashiCorp has an excellent suite of DevOps products related to infrastructure and configuration management. HashiCorp provides **Packer** to help you create immutable infrastructure by baking configurations directly in your VM image, rather than the slow process of creating a VM with a generic OS image and then customizing it later. It works on a similar principle as Docker uses to bake container images; that is, you define a template (configuration file) that specifies the source image, the desired configuration, and any provisioning steps needed to set up the software on the image. Packer then builds the image by creating a temporary instance with the base image, applying the defined configuration, and capturing the machine image for reuse.

Packer provides some of the following key features:

- **Multi-platform support:** Packer works on the plugin architecture and, therefore, can be used to create VM images for a lot of different cloud and on-premises platforms, such as VMware, Oracle VirtualBox, Amazon EC2, Azure's ARM, Google Cloud Compute, and container images for Docker or other container runtimes.
- **Automation:** Packer automates image creation and eliminates manual effort to build images. It also helps you with your multi-cloud strategy, as you can use a single configuration to build images for various platforms.
- **Fosters GitOps:** Packer configurations are machine-readable and written in HCL or JSON, so they can easily sit with your code. This, therefore, fosters GitOps.
- **Integration with other tools:** Packer integrates well with other HashiCorp tools, such as Terraform and Vagrant.

Packer uses a staging VM to customize the image. The following is the process that Packer follows while building the custom image:

1. You start with Packer configuration HCL files to define the base image you want to start from and where to build the image. You also define the provisioner for building the custom image, such as Ansible, and specify what playbooks to use.
2. When you run a Packer build, Packer uses the details in the configuration files to create a build VM from the base image, run the provisioner to customize it, turn off the build VM, take a snapshot, and save that as a disk image. It finally saves the image in an image repository.
3. You can then build the VM from the custom image using Terraform or other tools.

The following figure explains the process in detail:

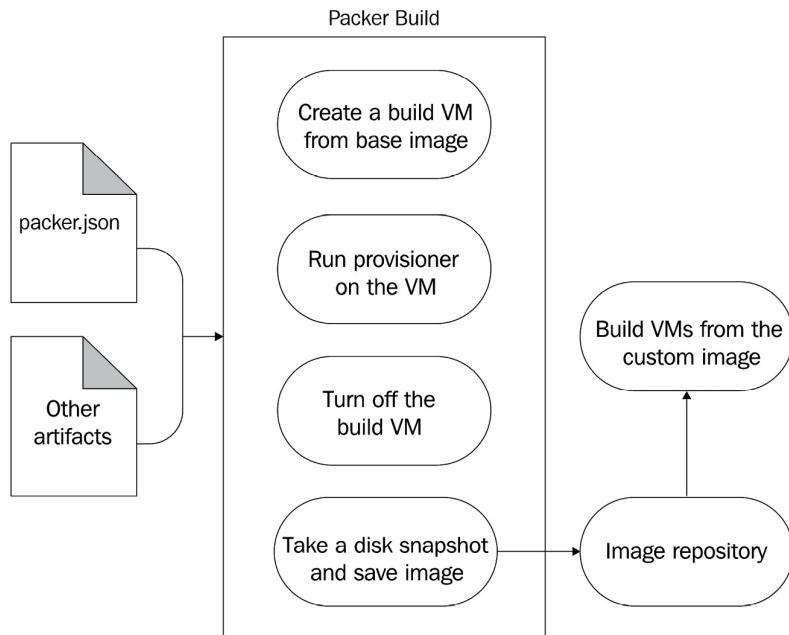


Figure 10.1 – Packer build process

The result is that your application is quick to start up and scales very well. For any changes within your configuration, create a new disk image with Packer and Ansible and then use Terraform to apply the changes to your resources. Terraform will then spin down the old VMs and spin up new ones with the new configuration. If you can relate it to the container deployment workflow, you can make real sense of it. It's akin to using the container workflow within the VM world! But is immutable infrastructure for everyone? Let's understand where it fits best.

When to use immutable infrastructure

Deciding to switch to immutable infrastructure is difficult, especially when your Ops team treats servers as pets. Most people get paranoid about the idea of deleting an existing server and creating a new one for every update. Well, you need to do a lot of convincing when you first come up with the idea. However, it does not mean that you must use immutable infrastructure to do proper DevOps. It all depends on your use case.

Let's look at each approach's pros and cons to understand them better.

Pros of mutable infrastructure

Let's begin with the pros of mutable infrastructure:

- If adequately managed, mutable infrastructure is faster to upgrade and change. It makes security patches quicker.
- It is simpler to manage, as we don't have to worry about building the entire VM image and redeploying it for every update.

Cons of mutable infrastructure

Next, let's see the cons of mutable infrastructure:

- It eventually results in configuration drift. When people start making changes manually in the server and do not use a config management tool, it becomes difficult to know what's in the server after a particular point. Then, you will have to start relying on snapshots.
- Versioning is impossible with mutable infrastructure, and rolling back changes is troublesome.
- There is a possibility of partial updates because of technical issues such as a patchy network, unresponsive **apt** repositories, and so on.
- There is a risk because changes are applied directly to the production environment. There is also a chance that you will end up in an unanticipated state that is difficult to troubleshoot.
- Because of configuration drift, it is impossible to say that the current configuration is the same as being tracked in version control. Therefore, building a new server from scratch may require manual intervention and comprehensive testing.

Similarly, let's look at the pros and cons of immutable infrastructure.

Pros of immutable infrastructure

The pros of immutable infrastructure are as follows:

- It eliminates configuration drift as the infrastructure cannot change once deployed, and any changes should come via the CI/CD process.
- It is DevOps-friendly as every build and deployment process inherently follows modern DevOps practices.
- It makes discrete versioning possible as every image generated from an image build can be versioned and kept within an image repository. That makes rollouts and rollbacks much more straightforward and promotes modern DevOps practices such as **canary** and **blue-green** deployments with A/B testing.
- The image is pre-built and tested, so we always get a predictable state from immutable infrastructure. We, therefore, reduce a lot of risk from production implementations.

- It helps with horizontal scaling on the cloud because you can now create servers from pre-built images, making new VMs faster to start up and get ready.

Cons of immutable infrastructure

The cons of immutable infrastructure are as follows:

- Building and deploying immutable infrastructure is a bit complex, and it is slow to add updates and manage urgent hotfixes
- There are storage and network overheads in generating and managing VM images

So, as we've looked at the pros and cons of both approaches, it ultimately depends on how you currently do infrastructure management and your end goal. Immutable infrastructure has a huge benefit, and therefore, it is something that every modern DevOps engineer should understand and implement if possible. However, technical and process constraints prevent people from doing it – while some constraints are related to the technology stack, most are simply related to processes and red tape. Immutable infrastructure is best when you need consistently reproducible and exceptionally reliable deployments. This approach minimizes the risk of configuration drift and streamlines updates by reconstructing entire environments instead of tweaking existing elements. It proves especially advantageous in scenarios such as microservices architectures, container orchestration, and situations where rapid scaling and the ability to roll back changes are paramount.

We all know that DevOps is not all about tools but it is a cultural change that should originate from the very top. If it is not possible to use immutable infrastructure, you can always use a **config management** tool such as Ansible on top of live servers. That makes things manageable to a certain extent.

Now, moving on to Packer, let's look at how to install it.

Installing Packer

You can install Packer on a variety of platforms in a variety of ways. Please refer to <https://developer.hashicorp.com/packer/downloads>. As Packer is available as an **apt** package, use the following commands to install Packer on Ubuntu Linux:

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | sudo \
gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
sudo tee /etc/apt/sources.list.d/hashicorp.list
$ sudo apt update && sudo apt install -y packer
```

To verify the installation, run the following command:

```
$ packer --version
1.9.2
```

As we see, Packer is installed successfully. We can proceed with the next activity in our goal – *creating playbooks*.

Creating the Apache and MySQL playbooks

As our goal is to spin up a scalable **LAMP stack** in this chapter, we must start by defining Ansible playbooks that would run on the build VM. We've already created some roles for Apache and MySQL in *Chapter 9, Configuration Management with Ansible*. We will use the same roles within this setup as well.

Therefore, we will have the following directory structure within the ch10 directory:

```
└── ansible
    ├── dbserver-playbook.yaml
    ├── roles
    │   ├── apache
    │   ├── common
    │   └── mysql
    └── webserver-playbook.yaml
└── packer
    ├── dbserver.pkr.hcl
    ├── plugins.pkr.hcl
    ├── variables.pkr.hcl
    ├── variables.pkrvars.hcl
    └── webserver.pkr.hcl
└── terraform
    ├── main.tf
    ├── outputs.tf
    ├── terraform.tfvars
    └── vars.tf
```

We have two playbooks within the `ansible` directory – `webserver-playbook.yaml` and `dbserver-playbook.yaml`. Let's look at each to understand how we write our playbooks for Ansible.

`webserver-playbook.yaml` looks like the following:

```
---
- hosts: default
  become: true
  roles:
    - common
    - apache
```

`dbserver-playbook.yaml` looks like the following:

```
---
- hosts: default
  become: true
  roles:
```

```
- common  
- mysql
```

As we can see, both playbooks have `hosts` set to `default`. That is because we will not define the inventory for this playbook. Instead, Packer will use the build VM to build the image and dynamically generate the inventory.

Note

Packer will also ignore any `remote_user` attributes within the task and use the user present in the Ansible provisioner's config.

As we've already tested this configuration in the previous chapter, all we need to do now is define the Packer configuration, so let's go ahead and do that in the next section.

Building the Apache and MySQL images using Packer and Ansible provisioners

We will now use Packer to create the Apache and MySQL images. Before defining the Packer configuration, we have a few prerequisites to allow Packer to build custom images.

Prerequisites

We must create an **Azure service principal** for Packer to interact with Azure and build the image.

First, log in to your Azure account using the Azure CLI with the following command:

```
$ az login
```

Now, set the subscription to the subscription ID we got in response to the `az login` command to an environment variable using the following:

```
$ export SUBSCRIPTION_ID=<SUBSCRIPTION_ID>
```

Next, let's set the subscription ID using the following command:

```
$ az account set --subscription="${SUBSCRIPTION_ID}"
```

Then, create the service principal with contributor access using the following command:

```
$ az ad sp create-for-rbac --role="Contributor" \  
--scopes="/subscriptions/${SUBSCRIPTION_ID}"  
{ "appId": "00000000-0000-0000-0000-000000000000", "name": "http://azure-  
cli-2021-01-07-05-59-24", "password": "xxxxxxxxxxxxxxxxxxxxxxxxx", "tenant": "00000000-  
0000-0000-000000000000" }
```

We've successfully created the service principal. The response JSON consists of `appId`, `password`, and `tenant` values that we will use in the subsequent sections.

Note

You can also reuse the service principal we created in *Chapter 8, Infrastructure as Code (IaC) with Terraform*, instead.

Now, let's go ahead and set the values of these variables in the `packer/variables.pkrvars.hcl` file with the details:

```
client_id = "<VALUE_OF_APP_ID>"  
client_secret = "<VALUE_OF_PASSWORD>"  
tenant_id = "<VALUE_OF_TENANT>"  
subscription_id = "<SUBSCRIPTION_ID>"
```

We will use the variable file in our Packer build. We also need a resource group for storing the built images.

To create the resource group, run the following command:

```
$ az group create -n packer-rg -l eastus
```

Now, let's go ahead and define the Packer configuration.

Defining the Packer configuration

Packer allows us to define configuration in JSON as well as HCL files. As JSON is now deprecated and HCL is preferred, let's define the Packer configuration using HCL.

To access resources for this section, switch to the following directory:

```
$ cd ~/modern-devops/ch10/packer
```

We will create the following files in the `packer` directory:

- `variables.pkrvars.hcl`: Contains a list of variables we would use while applying the configuration
- `plugins.pkrvars.hcl`: Contains the Packer plugin configuration
- `webserver.pkrvars.hcl`: Contains the Packer configuration for building the web server image
- `dbserver.pkrvars.hcl`: Contains the Packer configuration for building the dbserver image
- `variables.pkrvars.hcl`: Contains the values of the Packer variables defined in the `variables.pkrvars.hcl` file

The `variables.pkr.hcl` file contains the following:

```
variable "client_id" {
    type    = string
}
variable "client_secret" {
    type    = string
}
variable "subscription_id" {
    type    = string
}
variable "tenant_id" {
    type    = string
}
```

The `variables.pkr.hcl` file defines a list of user variables that we can use within the `source` and `build` blocks of the Packer configuration. We've defined four string variables – `client_id`, `client_secret`, `tenant_id`, and `subscription_id`. We can pass the values of these variables by using the `variables.pkrvars.hcl` variable file we defined in the last section.

Tip

Always provide sensitive data from external variables, such as a variable file, environment variables, or a secret manager, such as HashiCorp's Vault. You should never commit sensitive information with code.

The `plugins.pkr.hcl` file contains the following block:

`packer`: This section defines the common configuration for Packer. In this case, we've defined the plugins required to build the image. There are two plugins defined here – `ansible` and `azure`. Plugins contain a `source` and `version` attribute. They contain everything you would need to interact with the technology component:

```
packer {
    required_plugins {
        ansible = {
            source  = "github.com/hashicorp/ansible"
            version = "=1.1.0"
        }
        azure = {
            source  = "github.com/hashicorp/azure"
            version = "=1.4.5"
        }
    }
}
```

The `webserver.pkr.hcl` file contains the following sections:

- `source`: The `source` block contains the configuration we would use to build the VM. As we build an `azure-arm` image, we define the source as follows:

```
source "azure-arm" "webserver" {
  client_id           = var.client_id
  client_secret        = var.client_secret
  image_offer          = "UbuntuServer"
  image_publisher      = "Canonical"
  image_sku            = "18.04-LTS"
  location             = "East US"
  managed_image_name   = "apache-webserver"
  managed_image_resource_group_name = "packer-rg"
  os_type              = "Linux"
  subscription_id      = var.subscription_id
  tenant_id            = var.tenant_id
  vm_size              = "Standard_DS2_v2"
}
```

Different types of sources have different attributes that help us connect and authenticate with the cloud provider that the source is associated with. Other attributes define the build VM's specification and the base image that the build VM will use. It also describes the properties of the custom image we're trying to create. Since we're using Azure in this case, its source type is `azure-arm` and consists of `client_id`, `client_secret`, `tenant_id`, and `subscription_id`, which helps Packer authenticate with the Azure API server. These attributes' values are sourced from the `variables.pkr.hcl` file.

Tip

The managed image name can also contain a version. That will help you build a new image for every new version you want to deploy.

- `build`: The `build` block consists of `sources` and `provisioner` attributes. It contains all the sources we want to use, and the `provisioner` attribute allows us to configure the build VM to achieve the desired configuration. We've defined the following `build` block:

```
build {
  sources = ["source.azure-arm.webserver"]
  provisioner "ansible" {
    playbook_file = ".../ansible/webserver-playbook.yaml"
  }
}
```

We've defined an **Ansible provisioner** to customize our VM. There are a lot of provisioners that Packer provides. Luckily, Packer provides the Ansible provisioner out of the box. The Ansible provisioner requires the path to the playbook file; therefore, in this case, we've provided `.../ansible/webserver-playbook.yaml`.

Tip

You can specify multiple sources in the `build` block, each with the same or different types. Similarly, we can have numerous provisioners, each executed in parallel. So, if you want to build the same configuration for multiple cloud providers, you can specify multiple sources for each cloud provider.

Similarly, we've defined the following `dbserver.pkr.hcl` file:

```
source "azure-arm" "dbserver" {
    ...
    managed_image_name          = "mysql-dbserver"
    ...
}
build {
    sources = ["source.azure-arm.dbserver"]
    provisioner "ansible" {
        playbook_file = "../ansible/dbserver-playbook.yaml"
    }
}
```

The `source` block has the same configuration as the web server apart from `managed_image_name`. The `build` block is also like the web server, but instead, it uses the `../ansible/dbserver-playbook.yaml` playbook.

Now, let's look at the Packer workflow and how to use it to build the image.

The Packer workflow for building images

The Packer workflow comprises two steps – `init` and `build`.

As we already know, Packer uses plugins to interact with the cloud providers; therefore, we need to install them. To do so, Packer provides the `init` command.

Let's initialize and install the required plugins using the following command:

```
$ packer init .
Installed plugin github.com/hashicorp/ansible v1.1.0 in "~/.config/packer/plugins/github.com/hashicorp/ansible/packer-plugin-ansible_v1.1.0_x5.0_linux_amd64"
Installed plugin github.com/hashicorp/azure v1.4.5 in "~/.config/packer/plugins/github.com/hashicorp/azure/packer-plugin-azure_v1.4.5_x5.0_linux_amd64"
```

As we can see, the plugin is now installed. Let's now go ahead and build the image.

We use the `build` command to create an image using Packer. As we would need to pass values to variables, we will specify the variable values using a command-line argument, as in the following command:

```
$ packer build -var-file="variables.pkrvars.hcl" .
```

Packer would build parallel stacks using both the `webserver` and `dbserver` configs.

Packer first creates temporary resource groups to spin up staging VMs:

```
=> azure-arm.webserver: Creating resource group ...
=> azure-arm.webserver: -> ResourceGroupName : 'pkr-Resource-Group-7dfj1c2iej'
=> azure-arm.webserver: -> Location : 'East US'
=> azure-arm.dbserver: Creating resource group ...
=> azure-arm.dbserver: -> ResourceGroupName : 'pkr-Resource-Group-11xqpxsm3'
=> azure-arm.dbserver: -> Location : 'East US'
```

Packer then validates and deploys the deployment templates and gets the IP addresses of the staging VMs:

```
=> azure-arm.webserver: Validating deployment template ...
=> azure-arm.webserver: Deploying deployment template ...
=> azure-arm.webserver: -> DeploymentName : 'pkrdp7dfj1c2iej'
=> azure-arm.webserver: Getting the VM's IP address ...
=> azure-arm.webserver: -> IP Address : '104.41.158.85'
=> azure-arm.dbserver: Validating deployment template ...
=> azure-arm.dbserver: Deploying deployment template ...
=> azure-arm.dbserver: -> DeploymentName : 'pkrdp11xqpxsm3'
=> azure-arm.dbserver: Getting the VM's IP address ...
=> azure-arm.dbserver: -> IP Address : '40.114.7.11'
```

Then, Packer uses SSH to connect with the staging VMs and provisions them with Ansible:

```
=> azure-arm.webserver: Waiting for SSH to become available...
=> azure-arm.dbserver: Waiting for SSH to become available...
=> azure-arm.webserver: Connected to SSH!
=> azure-arm.dbserver: Connected to SSH!
=> azure-arm.webserver: Provisioning with Ansible...
=> azure-arm.dbserver: Provisioning with Ansible...
=> azure-arm.webserver: Executing Ansible: ansible-playbook -e packer_build_
name="webserver" -e packer_builder_type=azure-arm --ssh-extra-args '-o IdentitiesOnly=yes'
-e ansible_ssh_private_key_file=/tmp/ansible-key328774773 -i /tmp/packer-provisioner-
ansible747322992 ~/ansible/webserver-playbook.yaml
=> azure-arm.dbserver: Executing Ansible: ansible-playbook -e packer_build_
name="dbserver" -e packer_builder_type=azure-arm --ssh-extra-args '-o IdentitiesOnly=yes'
-e ansible_ssh_private_key_file=/tmp/ansible-key906086565 -i /tmp/packer-provisioner-
ansible3847259155 ~/ansible/dbserver-playbook.yaml
azure-arm.webserver: PLAY RECAP ****
** 
azure-arm.webserver: default: ok=7 changed=5 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
azure-arm.dbserver: PLAY RECAP ****
** 
azure-arm.dbserver: default: ok=11 changed=7 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Once the Ansible run is complete, Packer gets the disk details, captures the images, and creates the machine images in the resource groups we specified in the Packer configuration:

```
=> azure-arm.webserver: Querying the machine's properties
=> azure-arm.dbserver: Querying the machine's properties
=> azure-arm.webserver: Querying the machine's additional disks properties ...
=> azure-arm.dbserver: Querying the machine's additional disks properties ...
=> azure-arm.webserver: Powering off machine ...
```

```

==> azure-arm.dbserver: Powering off machine ...
==> azure-arm.webserver: Generalizing machine ...
==> azure-arm.dbserver: Generalizing machine ...
==> azure-arm.webserver: Capturing image ...
==> azure-arm.dbserver: Capturing image ...
==> azure-arm.webserver: -> Image ResourceGroupName: 'packer-rg'
==> azure-arm.dbserver: -> Image ResourceGroupName: 'packer-rg'
==> azure-arm.webserver: -> Image Name: 'apache-webserver'
==> azure-arm.webserver: -> Image Location: 'East US'
==> azure-arm.dbserver: -> Image Name: 'mysql-dbserver'
==> azure-arm.dbserver: -> Image Location: 'East US'

```

Finally, it removes the deployment object and the temporary resource group it created:

```

==> azure-arm.webserver: Deleting Virtual Machine deployment and its attached resources...
==> azure-arm.dbserver: Deleting Virtual Machine deployment and its attached resources...
==> azure-arm.webserver: Cleanup requested, deleting resource group ...
==> azure-arm.dbserver: Cleanup requested, deleting resource group ...
==> azure-arm.webserver: Resource group has been deleted.
==> azure-arm.dbserver: Resource group has been deleted.

```

It then provides the list of artifacts it has generated:

```

==> Builds finished. The artifacts of successful builds are:
--> azure-arm: Azure.ResourceManagement.VMImage:
OSType: Linux
ManagedImageResourceGroupName: packer-rg
ManagedImageName: apache-webserver
ManagedImageId: /subscriptions/Id/resourceGroups/packer-rg/providers/Microsoft.Compute/
images/apache-webserver
ManagedImageLocation: West Europe
OSType: Linux
ManagedImageResourceGroupName: packer-rg
ManagedImageName: mysql-dbserver
ManagedImageId: /subscriptions/Id/resourceGroups/packer-rg/providers/Microsoft.Compute/
images/mysql-dbserver

```

If we look at the `packer-rg` resource group, we will find that there are two VM images within it:

Name	Type	Location
apache-webserver	Image	East US
mysql-dbserver	Image	East US

Figure 10.2 – Packer custom images

We've successfully built custom images with Packer!

Tip

It isn't possible to rerun Packer with the same managed image name once the image is created in the resource group. That is because we don't want to override an existing image accidentally. While you can override it by using the `-force` flag with `packer build`, you should include a version within the image name to allow multiple versions of the image to exist in the resource group. For example, instead of using `apache-webserver`, you can use `apache-webserver-0.0.1`.

It's time to use these images and create our infrastructure with them now.

Creating the required infrastructure with Terraform

Our goal was to build a scalable LAMP stack, so we will define a **VM scale set** using the `apache-webserver` image we created and a single VM with the `mysql-dbserver` image. A VM scale set is an autoscaling group of VMs that will scale out and scale back horizontally based on traffic, similar to how we did with containers on Kubernetes.

We will create the following resources:

- A new resource group called `lamp-rg`
- A virtual network within the resource group called `lampvnet`
- A subnet within `lampvnet` called `lampsusb`
- Within the subnet, we create a **Network Interface Card (NIC)** for the database called `db-nic` that contains the following:
 - A network security group called `db-nsg`
 - A VM called `db` that uses the custom `mysql-dbserver` image
- We then create a VM scale set that includes the following:
 - A network profile called `webnnp`
 - A backend address pool
 - A load balancer called `web-lb`
 - A public IP address attached to `web-lb`
 - An HTTP probe that checks the health of port 80

The following figure explains the topology graphically:

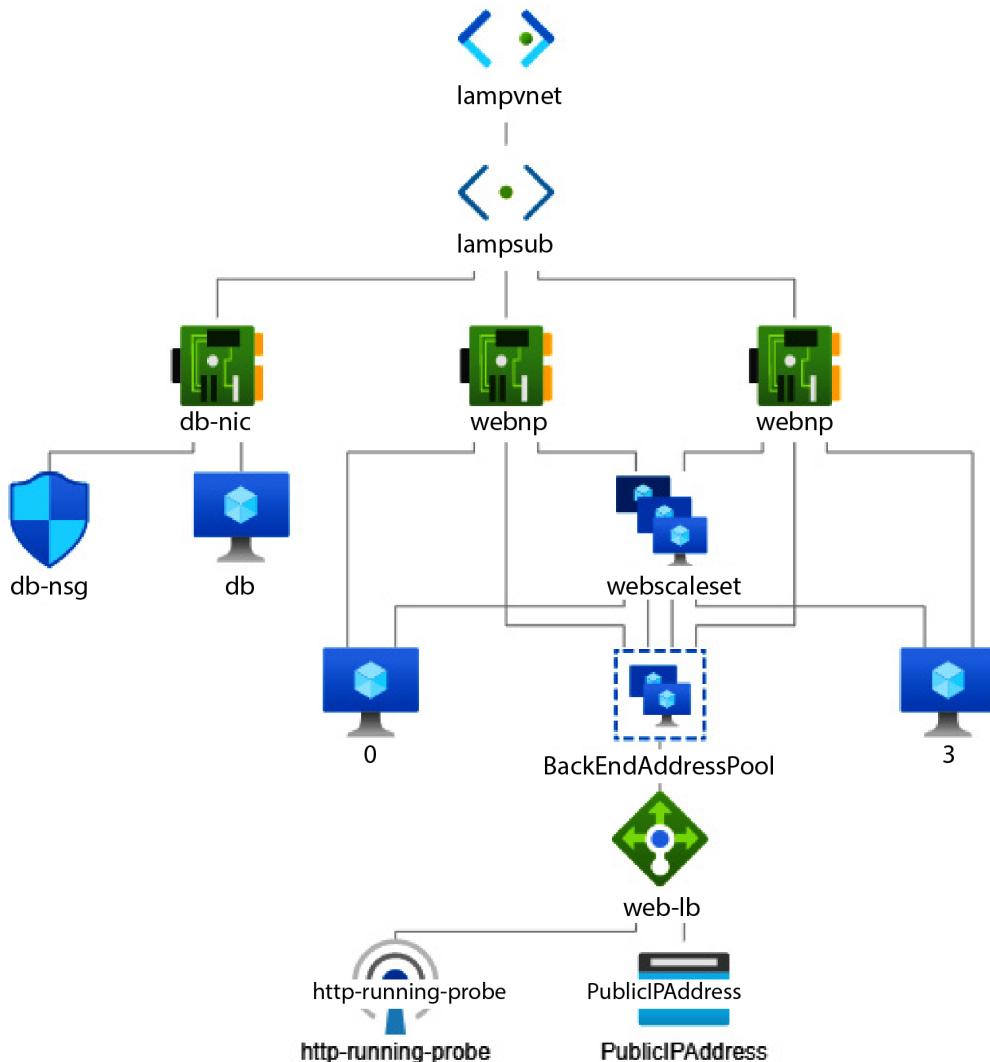


Figure 10.3 – Scalable LAMP stack topology diagram

To access resources for this section, switch to the following directory:

```
$ cd ~/modern-devops/ch10/terraform
```

We use the following Terraform template, `main.tf`, to define the configuration.

We first define the Terraform providers:

```
terraform {
  required_providers {
    azurerm = {
      source  = "azurerm"
    }
  }
  provider "azurerm" {
    subscription_id = var.subscription_id
    client_id       = var.client_id
    client_secret   = var.client_secret
    tenant_id        = var.tenant_id
  }
}
```

We then define the custom image data sources so that we can use them within our configuration:

```
data "azurerm_image" "websig" {
  name          = "apache-webserver"
  resource_group_name = "packer-rg"
}
data "azurerm_image" "dbsig" {
  name          = "mysql-dbserver"
  resource_group_name = "packer-rg"
}
```

We then define the resource group, virtual network, and subnet:

```
resource "azurerm_resource_group" "main" {
  name      = var.rg_name
  location  = var.location
}
resource "azurerm_virtual_network" "main" {
  name          = "lampvnet"
  address_space = ["10.0.0.0/16"]
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
}
resource "azurerm_subnet" "main" {
  name          = "lampsusb"
  resource_group_name = azurerm_resource_group.main.name
  virtual_network_name = azurerm_virtual_network.main.name
  address_prefixes = ["10.0.2.0/24"]
}
```

As the Apache web servers will remain behind a network load balancer, we will define the load balancer and the public IP address that we will attach to it:

```
resource "azurerm_public_ip" "main" {
  name          = "webip"
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
  allocation_method  = "Static"
  domain_name_label = azurerm_resource_group.main.name
}

resource "azurerm_lb" "main" {
  name          = "web-lb"
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
  frontend_ip_configuration {
    name          = "PublicIPAddress"
    public_ip_address_id = azurerm_public_ip.main.id
  }
  tags = {}
}
```

We will then define a backend address pool to the load balancer so that we can use this within the Apache VM scale set:

```
resource "azurerm_lb_backend_address_pool" "bpepool" {
  loadbalancer_id  = azurerm_lb.main.id
  name            = "BackEndAddressPool"
}
```

We will define an HTTP probe on port 80 for a health check and attach it to the load balancer:

```
resource "azurerm_lb_probe" "main" {
  loadbalancer_id  = azurerm_lb.main.id
  name            = "http-running-probe"
  port            = 80
}
```

We need a **NAT rule** to map the load balancer ports to the backend pool port, and therefore, we will define a load balancer rule that will map port 80 on the load balancer with port 80 of the backend pool VMs. We will also attach the HTTP health check probe in this config:

```
resource "azurerm_lb_rule" "lbnatrule" {
  resource_group_name      = azurerm_resource_group.main.name
  loadbalancer_id          = azurerm_lb.main.id
  name                     = "http"
  protocol                 = "Tcp"
  frontend_port            = 80
  backend_port              = 80
  backend_address_pool_ids = [ azurerm_lb_backend_address_pool.bpepool.id ]
  frontend_ip_configuration_name = "PublicIPAddress"
```

```

    probe_id          = azurerm_lb_probe.main.id
}

```

Now, we will define the VM scale set within the resource group using the custom image and the load balancer we defined before:

```

resource "azurerm_virtual_machine_scale_set" "main" {
  name          = "webscaleset"
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
  upgrade_policy_mode = "Manual"
  sku {
    name      = "Standard_DS1_v2"
    tier      = "Standard"
    capacity  = 2
  }
  storage_profile_image_reference {
    id=data.azurerm_image.websig.id
  }
}

```

We then go ahead and define the OS disk and the data disk:

```

storage_profile_os_disk {
  name          = ""
  caching       = "ReadWrite"
  create_option = "FromImage"
  managed_disk_type = "Standard_LRS"
}
storage_profile_data_disk {
  lun          = 0
  caching       = "ReadWrite"
  create_option = "Empty"
  disk_size_gb = 10
}

```

The OS profile defines how we log in to the VM:

```

os_profile {
  computer_name_prefix = "web"
  admin_username       = var.admin_username
  admin_password       = var.admin_password
}
os_profile_linux_config {
  disable_password_authentication = false
}

```

We then define a network profile that will associate the scale set with the load balancer we defined before:

```

network_profile {
  name      = "webnp"
  primary   = true
}

```

```

ip_configuration {
  name          = "IPConfiguration"
  subnet_id    = azurerm_subnet.main.id
  load_balancer_backend_address_pool_ids = [azurerm_lb_backend_address_pool.bpepool.
id]
  primary      = true
}
tags = {}
}

```

Now, moving on to the database configuration, we will start by defining a network security group for the database servers to allow ports 22 and 3306 from internal servers within the virtual network:

```

resource "azurerm_network_security_group" "db_nsg" {
  name          = "db-nsg"
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
  security_rule {
    name          = "SSH"
    priority      = 1001
    direction     = "Inbound"
    access        = "Allow"
    protocol      = "Tcp"
    source_port_range = "*"
    destination_port_range = "22"
    source_address_prefix   = "*"
    destination_address_prefix = "*"
  }
  security_rule {
    name          = "SQL"
    priority      = 1002
    direction     = "Inbound"
    access        = "Allow"
    protocol      = "Tcp"
    source_port_range = "*"
    destination_port_range = "3306"
    source_address_prefix   = "*"
    destination_address_prefix = "*"
  }
  tags = {}
}

```

We then define a NIC to provide an internal IP to the VM:

```

resource "azurerm_network_interface" "db" {
  name          = "db-nic"
  location      = var.location
  resource_group_name = azurerm_resource_group.main.name
  ip_configuration {
    name          = "db-ipconfiguration"
    subnet_id    = azurerm_subnet.main.id
  }
}

```

```

    private_ip_address_allocation = "Dynamic"
}
}
}
```

We will then associate the network security group to the network interface:

```

resource "azurerm_network_interface_security_group_association" "db" {
    network_interface_id      = azurerm_network_interface.db.id
    network_security_group_id = azurerm_network_security_group.db_nsg.id
}
```

Finally, we'll define the database VM using the custom image:

```

resource "azurerm_virtual_machine" "db" {
    name          = "db"
    location      = var.location
    resource_group_name = azurerm_resource_group.main.name
    network_interface_ids = [azurerm_network_interface.db.id]
    vm_size       = var.vm_size
    delete_os_disk_on_termination = true
    storage_image_reference {
        id = data.azurerm_image.dbsig.id
    }
    storage_os_disk {
        name          = "db-osdisk"
        caching       = "ReadWrite"
        create_option = "FromImage"
        managed_disk_type = "Standard_LRS"
    }
    os_profile {
        computer_name = "db"
        admin_username = var.admin_username
        admin_password = var.admin_password
    }
    os_profile_linux_config {
        disable_password_authentication = false
    }
    tags = {}
}
```

Now, as we've defined everything we needed, fill the `terraform.tfvars` file with the required information, and go ahead and initialize our Terraform workspace by using the following command:

```
$ terraform init
```

As Terraform has initialized successfully, use the following command to apply the Terraform configuration:

```

$ terraform apply
Apply complete! Resources: 13 added, 0 changed, 0 destroyed.
Outputs:
web_ip_addr = "40.115.61.69"
```

As Terraform has applied the configuration and provided the load balancer IP address as an output, let's use that to navigate to the web server:



Figure 10.4 – LAMP stack working correctly

As we get the `Database Connected successfully` message, we see that the configuration is successful! We've successfully created a scalable LAMP stack using Packer, Ansible, and Terraform. It combines *IaC*, *configuration as code*, *immutable infrastructure*, and modern DevOps practices to create a seamless environment without manual intervention.

Summary

In this chapter, we have covered immutable infrastructure with Packer. We used Packer with the Ansible provisioner to build custom images for Apache and MySQL. We used the custom images to create a scalable LAMP stack using Terraform. The chapter introduced you to the era of modern DevOps, where everything is automated. We follow the same principles for building and deploying all kinds of infrastructure, be it containers or VMs. In the next chapter, we will discuss one of the most important topics of DevOps – **continuous integration**.

Questions

1. Immutable infrastructure helps avoid configuration drift. (True/False)
2. It is a best practice to source sensitive data from external variables such as environment variables or a secret management tool such as HashiCorp's Vault. (True/False)
3. What modifications must we make to our existing playbooks to allow Packer to use them?
 - A. Remove any existing `ansible.cfg` files from the current working directory.
 - B. Remove any host files from the current working directory.
 - C. Update the `hosts` attribute to default within the playbook.
 - D. None of the above.
4. Which of the following are the limitations of using the Ansible provisioner with Packer? (Choose two)
 - A. You cannot pass Jinja2 macros as is to your Ansible playbooks.
 - B. You cannot define `remote_user` within your Ansible playbooks.

- C. You cannot use Jinja2 templates within your Ansible playbooks.
 - D. You cannot use roles and variables within your Ansible playbooks.
5. While naming managed images, what should we consider? (Choose two)
- A. Name images as specifically as possible.
 - B. Use the version as part of the image.
 - C. Don't use the version as part of the image name. Instead, always use the `-force` flag within the Packer build.
6. When using multiple provisioners, how are configurations applied to the build VM?
- A. One after the other based on occurrence in the HCL file
 - B. Parallelly
7. We can use a single set of Packer files to build images with the same configuration in multiple cloud environments. (True/False)
8. What features does a VM scale set provide? (Choose two)
- A. It helps you horizontally scale VM instances with traffic.
 - B. It helps you auto-heal faulty VMs.
 - C. It helps you do canary deployments.
 - D. None of the above.

Answers

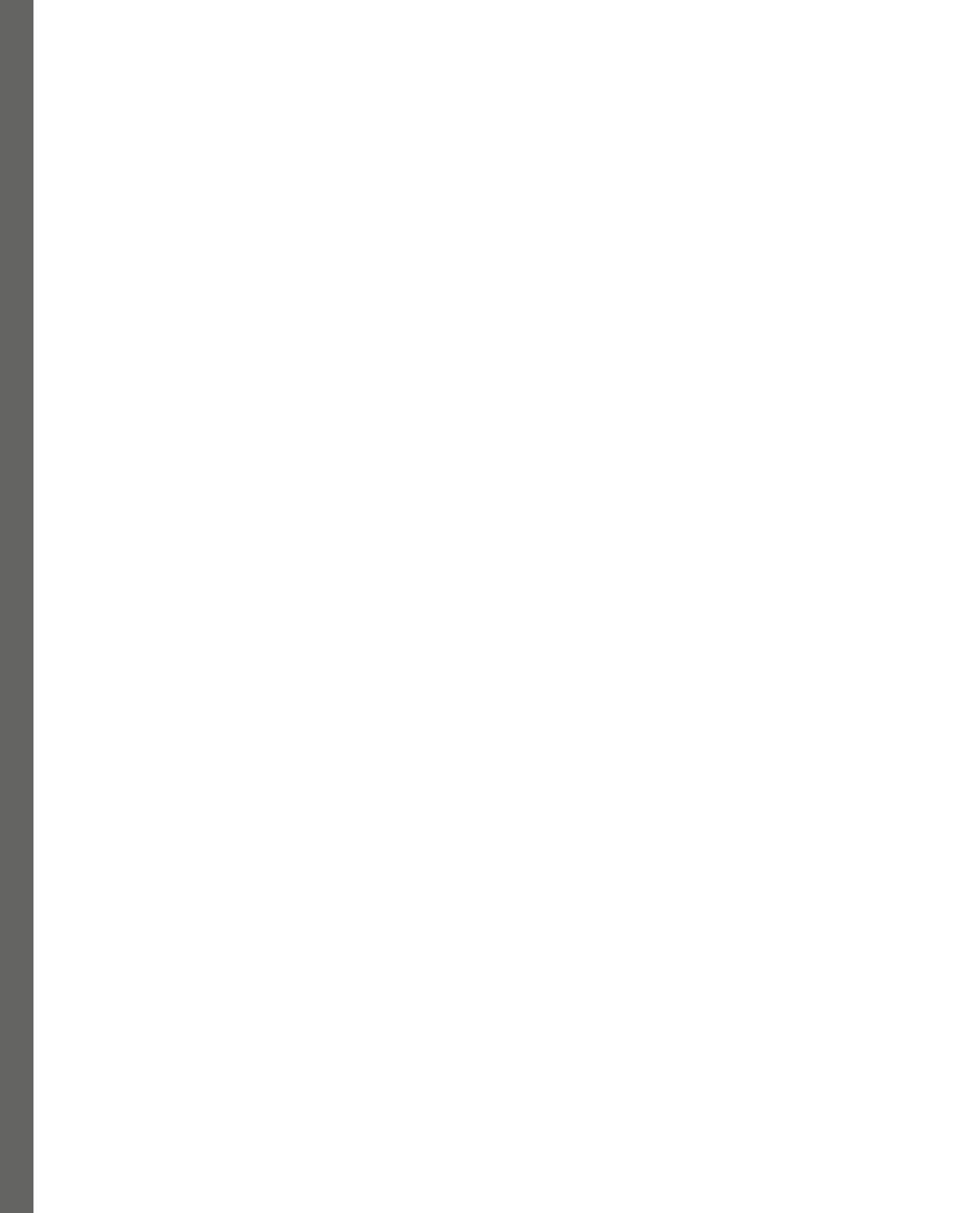
1. True
2. True
3. C
4. A, B
5. A, B
6. B
7. True
8. A, B, C

Part 4: Delivering Applications with GitOps

This section forms the core of the book and elucidates various tools and techniques to effectively implement modern DevOps in the cloud. With GitOps as the central guiding principle, we will explore various tools and techniques for continuously building, testing, securing, and deploying our applications into development, testing, and production environments.

This part has the following chapters:

- *Chapter 11, Continuous Integration with GitHub Actions and Jenkins*
- *Chapter 12, Continuous Deployment/Delivery with Argo CD*
- *Chapter 13, Securing and Testing Your CI/CD Pipeline*



11

Continuous Integration with GitHub Actions and Jenkins

In the previous chapters, we looked at individual tools that will help us implement several aspects of modern DevOps. Now, it's time to look at how we can combine all the tools and concepts we've learned about and use them to create a **continuous integration** (CI) pipeline. First, we will introduce a sample microservices-based blogging application, **Blog App**, and then look at some popular open source and SaaS-based tools that can get us started quickly with CI. We will begin with **GitHub Actions** and then move on to **Jenkins** with **Kaniko**. For every tool, we will implement CI for Blog App. We will try to keep the implementations cloud-agnostic. Since we've used the **GitOps** approach from the beginning, we will also use the same here. Finally, we will cover some best practices related to build performance.

In this chapter, we're going to cover the following main topics:

- The importance of automation
- Introduction to the sample microservices-based blogging application – Blog App
- Building a CI pipeline with GitHub Actions
- Scalable Jenkins on **Kubernetes** with Kaniko
- Automating a build with triggers
- Build performance best practices

Technical requirements

For this chapter, you will need to clone the following GitHub repository for some of the exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

Run the following command to clone the repository into your home directory, and `cd` into the `ch11` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch11
```

So, let's get started!

The importance of automation

Automation is akin to having an efficient team of robots at your disposal, tirelessly handling repetitive, time-consuming, and error-prone tasks. Let's simplify the significance of automation:

- **Efficiency:** Think of it as having a magical helper who completes tasks in a fraction of the time you would take. Automation accelerates repetitive tasks, executing actions, processing data, and running commands far more swiftly than humans.
- **Consistency:** Humans can tire or become distracted, leading to inconsistencies in task execution. Automation guarantees that tasks are consistently carried out according to predefined rules, every single time.
- **Accuracy:** Automation operates without the fatigue or lapses that humans may experience. It adheres to instructions with precision, minimizing the likelihood of errors that could result in costly repercussions.
- **Scale:** Whether managing one system or a thousand, automation effortlessly scales operations without additional human resources.
- **Cost savings:** By reducing the reliance on manual labor, automation yields significant cost savings in terms of time and human resources.
- **Risk reduction:** Certain tasks, such as making data backups and performing security checks, are crucial but can be overlooked or skipped by humans. Automation ensures these tasks are consistently performed, mitigating risks.
- **Faster response:** Automation detects and responds to issues in real time. For instance, it can automatically restart a crashed server or adjust resource allocation during high traffic, ensuring uninterrupted user experiences.
- **Resource allocation:** Automating routine tasks liberates human resources to concentrate on more strategic and creative endeavors that require critical thinking and decision-making.

- **Compliance:** Automation enforces and monitors compliance with policies and regulations, reducing the potential for legal and regulatory complications.
- **Data analysis:** Automation processes and analyzes vast data volumes rapidly, enabling data-driven decision-making and insights.
- **24/7 operations:** Automation operates tirelessly, 24/7, guaranteeing continuous operations and availability.
- **Adaptability:** Automation can be reprogrammed to adapt to evolving requirements and environments, making it versatile and future-proof.

In the tech realm, automation is the bedrock of modern IT operations, spanning from automating software deployments to managing cloud resources and configuring network devices. It empowers organizations to streamline processes, enhance reliability, and remain competitive in the fast-paced digital landscape.

In essence, automation resembles an exceedingly efficient, error-free, round-the-clock workforce that empowers individuals and organizations to accomplish more with less effort.

To benefit from automation, the project management function is quickly diluting, and software development teams are transitioning to Agile teams that deliver in Sprints iteratively. Therefore, if there is a new requirement, we don't wait for the entire thing to be signed off before we start doing design, development, QA, and so on. Instead, we break software into workable features and deliver them in smaller chunks to get value and customer feedback quickly. That means rapid software development with less risk of failure.

Well, the teams are agile, and they develop software faster. Still, many things in the **software development life cycle (SDLC)** process are conducted manually, such as the fact that some teams generate Code Builds only after completing the entire development for that cycle and later find numerous bugs. It becomes difficult to trace what caused that problem in the first place.

What if you could know the cause of a broken Build as soon as you check the code into source control? What if you understand that the software fails some tests as soon as the builds are executed? Well, that's CI for you in a nutshell.

CI is a process through which developers frequently check code into a source code repository, perhaps several times a day. Automated tooling behind the scenes can detect these commits and then build, run some tests, and tell you upfront whether the commit has caused any issues. This means that your developers, testers, product owners, operations team, and everyone comes to know what has caused the problem, and the developer can fix it quickly. This creates a feedback loop in software development. We always had a manual feedback loop within software development, which was slow. So, either you wait a long time before doing your next task or do the wrong thing until you realize it is too late to undo all of that. This adds to the rework effort of everything you have done hitherto.

As we all know, fixing a bug earlier in the SDLC cycle is cheaper than fixing it later. Therefore, CI aims to provide continuous feedback on the code quality early in the SDLC. This saves your developers and the organization a lot of time and money on fixing bugs they detect when most of your code is tested. Therefore, CI helps software development teams develop better software faster.

Since we've mentioned Agile, let's briefly discuss how it compares with DevOps. Agile is a way of working and is silent on the tools, techniques, and automation required to achieve it. DevOps is an extension of the Agile mindset and helps you implement it effectively. DevOps focuses heavily on automation and looks at avoiding manual work wherever possible. It also encourages software delivery automation and seeks to amplify or replace traditional tools and frameworks. With the advent of modern DevOps, specific tools, techniques, and best practices simplify the life of a developer, QA, and operator. Modern public cloud platforms and DevOps provide teams with ready-to-use dynamic infrastructure that helps businesses reduce the time to market and build scalable, elastic, high-performing infrastructure to keep enterprises live with minimal downtime.

When introducing modern DevOps in the first chapter, we discussed that it usually applies to modern cloud-native applications. I've built an example microservices-based Blog App to demonstrate this. We will use this application in this and future chapters of this book to ensure seamless development and delivery of this application using modern DevOps tools and practices. We'll look at the sample application in the next section.

Introduction to the sample microservices-based blogging application – Blog App

Blog App is a sample modern microservices-based blogging web application that allows users to create, manage, and interact with blog posts. It caters to both authors and readers. Users can sign up to this platform using their email addresses and start writing blog posts. Readers can publicly view all blog posts created by several authors, and logged-in users can also provide reviews and ratings.

The application is written in a popular Python-based web framework called **Flask** and uses **MongoDB** as the database. The application is split into several microservices for user, post, review, and rating management. There is a separate frontend microservice that allows for user interaction. Let's look at each microservice:

- **User Management:** The User Management microservice provides endpoints to create a user account, update the profile (name and password), and delete a user account.
- **Posts Management:** The Posts Management microservice provides endpoints to create, list, get, update, and delete posts.
- **Reviews Management:** The Reviews Management microservice allows users to add reviews on posts and update and delete them. Internally, it interacts with the Ratings Management microservice to manage the ratings provided, along with the reviews.

- **Ratings Management:** The Ratings Management microservice manages ratings for posts associated with a particular review. This microservice is called from the Reviews Management microservice internally and is not exposed to the Frontend microservice.
- **Frontend:** The Frontend microservice is a Python Flask user interface application built using **Bootstrap**, which provides users with a rich and interactive user interface. It allows users to sign up, log in, view, and navigate between posts, edit their posts, add and update reviews, and manage their profiles. The microservice interacts with the backend microservices seamlessly using HTTP requests.

The **users**, **posts**, **reviews**, and **ratings** microservices interact with **MongoDB** as the database.

The following service diagram shows the interactions graphically:

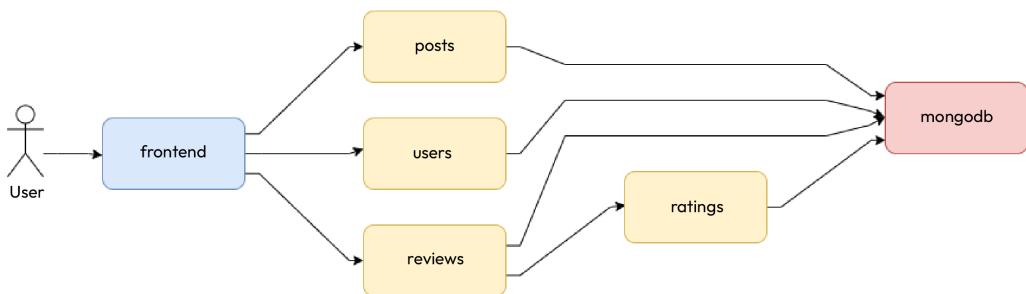


Figure 11.1 – Blog App services and interactions

As we can see, the individual microservices are fairly decoupled from each other and, therefore, can independently scale. It is also robust because the other parts of the application will work if a particular microservice is not working. The individual microservices can be independently developed and deployed as separate components, adding to the application's flexibility and maintainability. This application is an excellent example of leveraging microservices to build a modern, feature-rich web application.

Now, let's implement CI for this application. To implement CI, we will need a CI tool. We'll look at some of the popular tools and the options you have in the next section.

Building a CI pipeline with GitHub Actions

GitHub Actions is a SaaS-based tool that comes with **GitHub**. So, when you create your GitHub repository, you get access to this service out of the box. Therefore, GitHub Actions is one of the best tools for people new to CI/CD and who want to get started quickly. GitHub Actions helps you automate tasks, build, test, and deploy your code, and even streamline your workflow, making your life as a developer much easier.

Here's what GitHub Actions can do for you:

- **CI:** GitHub Actions can automatically build and test your code whenever you push changes to your repository. This ensures that your code remains error-free and ready for deployment.
- **CD:** You can use GitHub Actions to deploy your application to various hosting platforms, such as AWS, Azure, and GCP. This allows you to deliver updates to your users quickly and efficiently.
- **Workflow automation:** You can create custom workflows using GitHub Actions to automate repetitive tasks in your development process. For example, you can automatically label and assign issues, trigger builds on specific events, or send notifications to your team.
- **Custom scripts:** GitHub Actions allows you to run custom scripts and commands, giving you full control over your automation tasks. Whether you need to compile code, run tests, or execute deployment scripts, GitHub Actions can handle it.
- **Community actions:** GitHub Actions has a marketplace where you can find pre-built actions created by the community. These actions cover a wide range of tasks, from publishing to npm to deploying to popular cloud providers. You can easily incorporate these actions into your workflow.
- **Scheduled jobs:** You can schedule actions to run at specific times or intervals. This is handy for tasks such as generating reports, sending reminders, or performing maintenance during non-peak hours.
- **Multi-platform support:** GitHub Actions supports various programming languages, operating systems, and cloud environments, which means you can build and deploy applications for different platforms with ease.
- **Integration:** GitHub Actions seamlessly integrates with your GitHub repositories, making it a natural extension of your development environment. You can define workflows by using YAML files directly in your repository.

GitHub Actions revolutionizes the way developers work by automating routine tasks, ensuring code quality, and streamlining the SDLC. It's a valuable tool for teams and individual developers looking to enhance productivity and maintain high-quality code.

Now, let's create a CI pipeline for our sample Blog App. Blog App consists of multiple microservices, and each microservice runs on an individual **Docker** container. We also have unit tests written for each microservice, which we can run to verify the code changes. If the tests pass, the build will pass; otherwise, it will fail.

To access the resources for this section, cd into the following directory:

```
$ cd ~/modern-devops/blog-app
```

This directory contains multiple microservices and is structured as follows:

```
.  
├── frontend  
│   ├── Dockerfile  
│   ├── app.py  
│   ├── app.test.py  
│   ├── requirements.txt  
│   ├── static  
│   └── templates  
├── posts  
│   ├── Dockerfile  
│   ├── app.py  
│   ├── app.test.py  
│   └── requirements.txt  
├── ratings ...  
└── reviews ...  
└── users ...
```

The **frontend** directory contains files for the **frontend** microservice, and notably, it includes `app.py` (the Flask application code), `app.test.py` (the unit tests for the Flask application), `requirements.txt` (which contains all Python modules required by the app), and `Dockerfile`. It also includes a few other directories catering to the user interface elements of this app.

The **posts**, **reviews**, **ratings**, and **users** microservices have the same structure and contain `app.py`, `app.test.py`, `requirements.txt`, and `Dockerfile` files.

So, let's start by switching to the `posts` directory:

```
$ cd posts
```

As we know that Docker is inherently CI-compliant, we can run the tests using `Dockerfile` itself. Let's investigate the `Dockerfile` of the `posts` service:

```
FROM python:3.7-alpine  
ENV FLASK_APP=app.py  
ENV FLASK_RUN_HOST=0.0.0.0  
RUN apk add --no-cache gcc musl-dev linux-headers  
COPY requirements.txt requirements.txt  
RUN pip install -r requirements.txt  
EXPOSE 5000  
COPY . .  
RUN python app.test.py  
CMD ["flask", "run"]
```

This Dockerfile starts with the `python:3.7-alpine` base image, installs the requirements, and copies the code into the working directory. It runs the `app.test.py` unit test to check whether the code would work if we deploy it. Finally, the `CMD` command defines a `flask run` command to run when we launch the container.

Let's build our Dockerfile and see what we get:

```
$ docker build --progress=plain -t posts .
#4 [1/6] FROM docker.io/library/python:3.7-alpine
#5 [internal] load build context
#6 [2/6] RUN apk add --no-cache gcc musl-dev linux-headers
#7 [3/6] COPY requirements.txt requirements.txt
#8 [4/6] RUN pip install -r requirements.txt
#9 [5/6] COPY ..
#10 [6/6] RUN python app.test.py
#10 0.676 -----
#10 0.676 Ran 8 tests in 0.026s
#11 exporting to image
#11 naming to docker.io/library/posts done
```

As we can see, it built the container, executed a test on it, and responded with `Ran 8 tests in 0.026s` and an OK message. Therefore, we could use Dockerfile to build and test this app. We used the `--progress=plain` argument with the `docker build` command. This is because we wanted to see the stepwise output of the logs rather than Docker merging progress into a single message (this is now a default behavior).

Now, let's look at GitHub Actions and how we can automate this step.

Creating a GitHub repository

Before we can use GitHub Actions, we need to create a GitHub repository. As we know that each microservice can be independently developed, we will place all of them in separate Git repositories. For this exercise, we will focus only on the `posts` microservice and leave the rest to you as an exercise.

To do so, go to <https://github.com/new> and create a new repository. Give it an appropriate name. For this exercise, I am going to use `mdo-posts`.

Once you've created it, clone the repository by using the following command:

```
$ git clone https://github.com/<GitHub_Username>/mdo-posts.git
```

Then, change the directory into the repository directory and copy the `app.py`, `app.test.py`, `requirements.txt`, and `Dockerfile` files into the repository's directory using the following commands:

```
$ cd mdo-posts  
$ cp ~/modern-devops/blog-app/posts/* .
```

Now, we need to create a GitHub Actions workflow file. We'll do this in the next section.

Creating a GitHub Actions workflow

A GitHub Actions workflow is a simple YAML file that contains the build steps. We must create this workflow in the `.github/workflows` directory within the repository. We can do this using the following command:

```
$ mkdir -p .github/workflows
```

We will use the following GitHub Actions workflow file, `build.yaml`, for this exercise:

```
name: Build and Test App  
on:  
  push:  
    branches: [ main ]  
  pull_request:  
    branches: [ main ]  
jobs:  
  build:  
    runs-on: ubuntu-latest  
    steps:  
      - uses: actions/checkout@v2  
      - name: Login to Docker Hub  
        id: login  
        run: docker login -u ${{ secrets.DOCKER_USER }} -p ${{ secrets.DOCKER_PASSWORD }}  
      - name: Build the Docker image  
        id: build  
        run: docker build . --file Dockerfile --tag ${{ secrets.DOCKER_USER }}/  
mdo-posts:$({git rev-parse --short "$GITHUB_SHA"})  
      - name: Push the Docker image  
        id: push  
        run: docker push ${{ secrets.DOCKER_USER }}/mdo-posts:$({git rev-parse --short  
"$GITHUB_SHA"})
```

This file comprises the following:

- `name`: The workflow's name – `Build and Test App` in this case.
- `on`: This describes when this workflow will run. In this case, it will run if a push or pull request is sent on the `main` branch.
- `jobs`: A GitHub Actions workflow contains one or more jobs that run in parallel by default. This attribute includes all jobs.
- `jobs.build`: This is a job that does the container build.
- `jobs.build.runs-on`: This describes where the build job will run. We've specified `ubuntu-latest` here. This means that this job will run on an Ubuntu VM.
- `jobs.build.steps`: This consists of the steps that run sequentially within the job. The build job consists of four build steps: `checkout`, which will check out the code from your repository; `login`, which will log in to Docker Hub; `build`, which will run a Docker build on your code; and `push`, which will push your Docker image to **Docker Hub**. Note that we tag the image with the Git commit SHA. This relates the build with the commit, making Git the single source of truth.
- `jobs.build.steps.uses`: This is the first step and describes an action you will run as a part of your job. Actions are reusable pieces of code that you can execute in your pipeline. In this case, it runs the `checkout` action. It checks out the code from the current branch where the action is triggered.

Tip

Always use a version with your actions. This will prevent your build from breaking if a later version is incompatible with your pipeline.

- `jobs.build.steps.name`: This is the name of your build step.
- `jobs.build.steps.id`: This is the unique identifier of your build step.
- `jobs.build.steps.run`: This is the command it executes as part of the build step.

The workflow also contains variables within `${{ }}`. We can define multiple variables within the workflow and use them in the subsequent steps. In this case, we've used two variables – `${{ secrets.DOCKER_USER }}` and `${{ secrets.DOCKER_PASSWORD }}`. These variables are sourced from **GitHub secrets**.

Tip

It is best practice to use GitHub secrets to store sensitive information. Never store these details directly in the repository with code.

You must define two secrets within your repository using the following URL: https://github.com/<your_user>/mdo-posts/settings/secrets/actions.

Define two secrets within the repository:

```
DOCKER_USER=<Your Docker Hub username>
DOCKER_PASSWORD=<Your Docker Hub password>
```

Now, let's move this `build.yml` file to the `workflows` directory by using the following command:

```
$ mv build.yml .github/workflows/
```

Now, we're ready to push this code to GitHub. Run the following commands to commit and push the changes to your GitHub repository:

```
$ git add --all
$ git commit -m 'Initial commit'
$ git push
```

Now, go to the **Workflows** tab of your GitHub repository by visiting https://github.com/<your_user>/mdo-posts/actions. You should see something similar to the following:

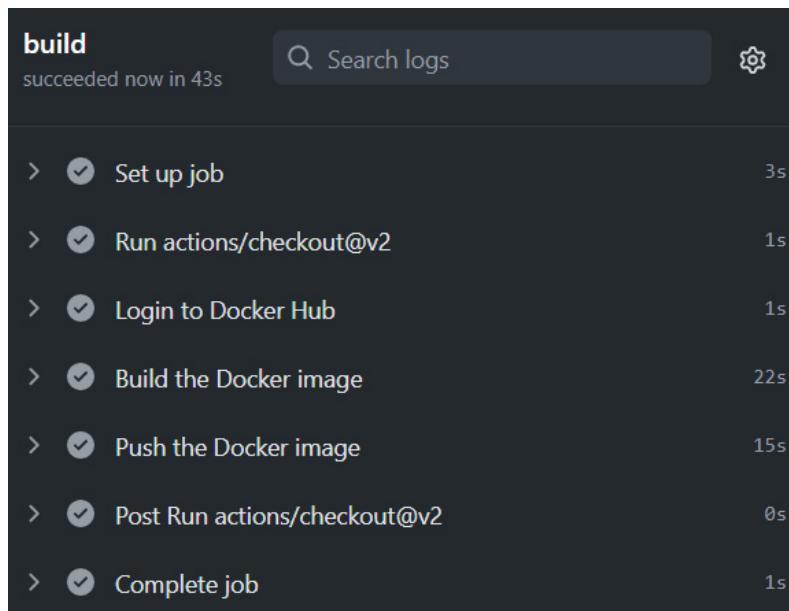


Figure 11.2 – GitHub Actions

As we can see, GitHub has run a build using our workflow file, and it has built the code and pushed the image to **Docker Hub**. Upon visiting your Docker Hub account, you should see your image present in your account:

bharamicrosystems / mdo-posts

Description

This repository does not have a description 

 Last pushed: 2 minutes ago

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
 latest		Image	---	2 minutes ago

[See all](#) [Go to Advanced Image Management](#)

Figure 11.3 – Docker Hub image

Now, let's try to break our code somehow. Let's suppose that someone from your team changed the `app.py` code, and instead of returning `post` in the `create_post` response, it started returning `pos`. Let's see what would happen in that scenario.

Make the following changes to the `create_post` function in the `app.py` file:

```
@app.route('/posts', methods=['POST'])
def create_post():
    ...
    return jsonify({'pos': str(inserted_post.inserted_id)}), 201
```

Now, commit and push the code to GitHub using the following commands:

```
$ git add --all
$ git commit -m 'Updated create_post'
$ git push
```

Now, go to GitHub Actions and find the latest build. You will see that the build will error out and give the following output:

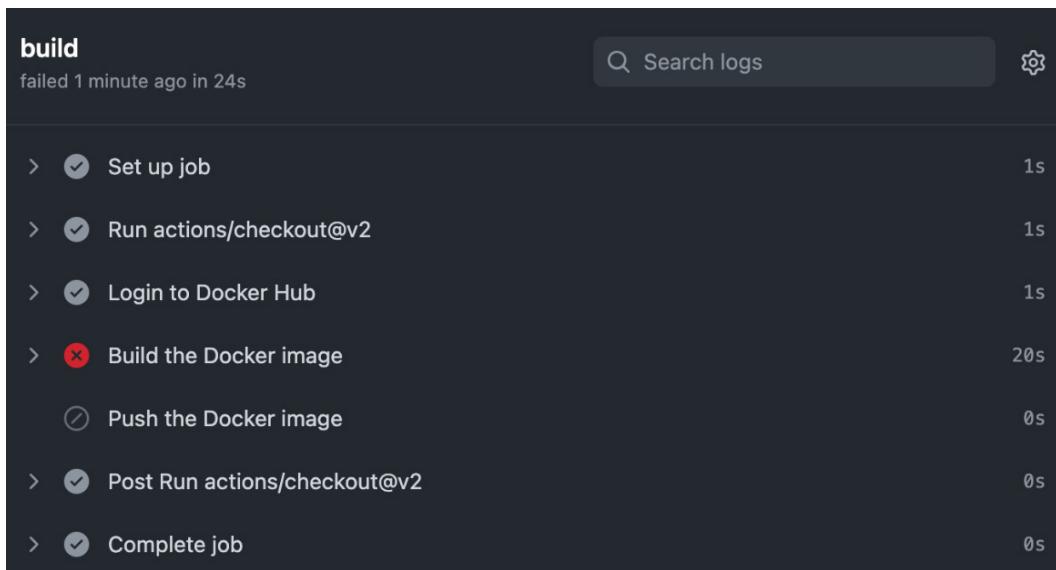


Figure 11.4 – GitHub Actions – build failure

As we can see, the **Build the Docker image** step has failed. If you click on the step and scroll down to see what happened with it, you will find that the `app.test.py` execution failed. This is because of a test case failure with `AssertionError: 'post' not found in {'pos': '60458fb603c395f9a81c9f4a'}`. As the expected `post` key was not found in the output, `{ 'pos': '60458fb603c395f9a81c9f4a' }`, the test case failed, as shown in the following screenshot:

```

build
failed 4 minutes ago in 24s
Search logs
20s

Build the Docker image
138 #11 0.681 FAILED (failures=1)
139 #11 ERROR: process "/bin/sh -c python app.test.py" did not complete successfully: exit code: 1
140 -----
141   > [6/6] RUN python app.test.py:
142 0.681   File "/usr/local/lib/python3.7/unittest/mock.py", line 1256, in patched
143 0.681     return func(*args, **keywargs)
144 0.681   File "app.test.py", line 40, in test_create_post
145 0.681     self.assertIn('post', data)
146 0.681 AssertionError: 'post' not found in {'pos': '60458fb603c395f9a81c9f4a'}
147 0.681
148 0.681 -----
149 0.681 Ran 8 tests in 0.022s
150 0.681
151 0.681 FAILED (failures=1)
152 -----
153 Dockerfile:9
154 -----
155 7 |   EXPOSE 5000
156 8 |   COPY . .
157 9 | >>> RUN python app.test.py
158 10 |   CMD ["flask", "run"]
159 11 |
160 -----
161 ERROR: failed to solve: process "/bin/sh -c python app.test.py" did not complete successfully: exit code: 1
162 Error: Process completed with exit code 1.

Push the Docker image
0s

Post Run actions/checkout@v2
0s

Complete job
0s

```

Figure 11.5 – GitHub Actions – test failure

We uncovered the error when someone pushed the buggy code to the Git repository. Are you able to see the benefits of CI already?

Now, let's fix the code and commit the code again.

Modify the `create_post` function of `app.py` so that it looks as follows:

```

@app.route('/posts', methods=['POST'])
def create_post():
    ...
    return jsonify({'post': str(inserted_post.inserted_id)}), 201

```

Then, commit and push the code to GitHub using the following commands:

```

$ git add --all
$ git commit -m 'Updated create_post'
$ git push

```

This time, the build will be successful:

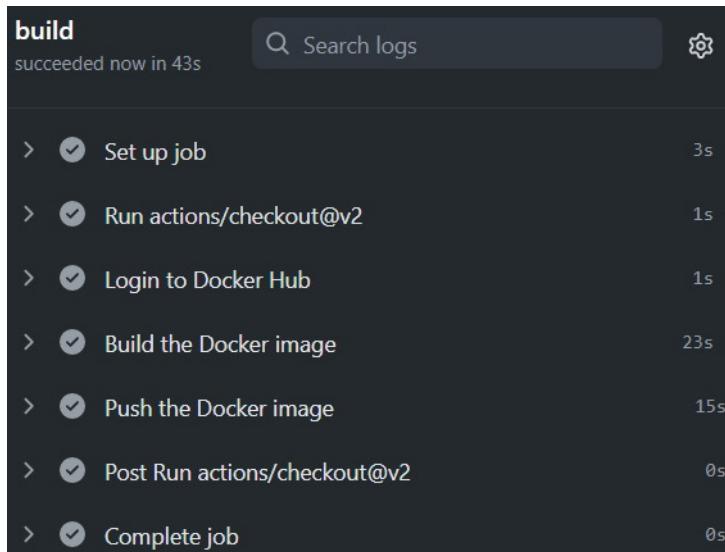


Figure 11.6 – GitHub Actions – build success

Did you see how simple this was? We got started with CI quickly and implemented GitOps behind the scenes since the config file required to build and test the code also resided with the application code.

As an exercise, repeat the same process for the **reviews**, **users**, **ratings**, and **frontend** microservices. You can play around with them to understand how it works.

Not everyone uses GitHub, so the SaaS offering might not be an option for them. Therefore, in the next section, we'll look at the most popular open source CI tool: Jenkins.

Scalable Jenkins on Kubernetes with Kaniko

Imagine you're running a workshop where you build all sorts of machines. In this workshop, you have a magical conveyor belt called Jenkins for assembling these machines. But to make your workshop even more efficient and adaptable, you've got a team of tiny robot workers called Kaniko that assist in constructing the individual parts of each machine. Let's draw parallels between this workshop analogy and the technology world:

- **Scalable Jenkins:** Jenkins is a widely used automation server that helps automate various tasks, particularly those related to building, testing, and deploying software. “Scalable Jenkins” means configuring Jenkins in a way that allows it to efficiently handle a growing workload, much like having a spacious workshop capable of producing numerous machines.

- **Kubernetes:** Think of Kubernetes as the workshop manager. It's an orchestration platform that automates the process of deploying, scaling, and managing containerized applications. Kubernetes ensures that Jenkins and the team of tiny robots (Kaniko) work seamlessly together and can adapt to changing demands.
- **Kaniko:** Kaniko is equivalent to your team of miniature robot workers. In the context of containerization, Kaniko is a tool that aids in building container images, which are akin to the individual parts of your machines. What makes Kaniko special is that it can do this without needing elevated access to the Docker daemon. Unlike traditional container builders, Kaniko doesn't require special privileges, making it a more secure choice for constructing containers, especially within a Kubernetes environment.

Now, let's combine the three tools and see what we can achieve:

- **Building containers at scale:** Your workshop can manufacture multiple machines simultaneously, thanks to Jenkins and the tiny robots. Similarly, with Jenkins on Kubernetes using Kaniko, you can efficiently and concurrently create container images. This ability to scale is crucial in modern application development, where containerization plays a pivotal role.
- **Isolation and security:** Just as Kaniko's tiny robots operate within a controlled environment, Kaniko ensures that container image building takes place in an isolated and secure manner within a Kubernetes cluster. This means that different teams or projects can use Jenkins and Kaniko without interfering with each other's container-building processes.
- **Consistency and automation:** Similar to how the conveyor belt (Jenkins) guarantees consistent machine assembly, Jenkins on Kubernetes with Kaniko ensures uniform container image construction. Automation is at the heart of this setup, simplifying the process of building and managing container images for applications.

To summarize, scalable Jenkins on Kubernetes with Kaniko refers to the practice of setting up Jenkins to efficiently build and manage container images using Kaniko within a Kubernetes environment. It enables consistent, parallel, and secure construction of container images, aligning perfectly with modern software development workflows.

So, the analogy of a workshop with Jenkins, Kubernetes, and Kaniko vividly illustrates how this setup streamlines container image building, making it scalable, efficient, and secure for contemporary software development practices. Now, let's dive deeper into Jenkins.

Jenkins is the most popular CI tool available in the market. It is open source, simple to install, and runs with ease. It is a Java-based tool with a plugin-based architecture designed to support several integrations, such as with a source code management tool such as *Git*, *SVN*, and *Mercurial*, or with popular artifact repositories such as *Nexus* and *Artifactory*. It also integrates well with well-known build tools such as *Ant*, *Maven*, and *Gradle*, aside from the standard shell scripting and Windows batch file executions.

Jenkins follows a *controller-agent* model. Though technically, you can run all your builds on the controller machine itself, it makes sense to offload your CI builds to other servers in your network to have a distributed architecture. This does not overload your controller machine. You can use it to store the build configurations and other management data and manage the entire CI build cluster, something along the lines of what's shown in the following diagram:

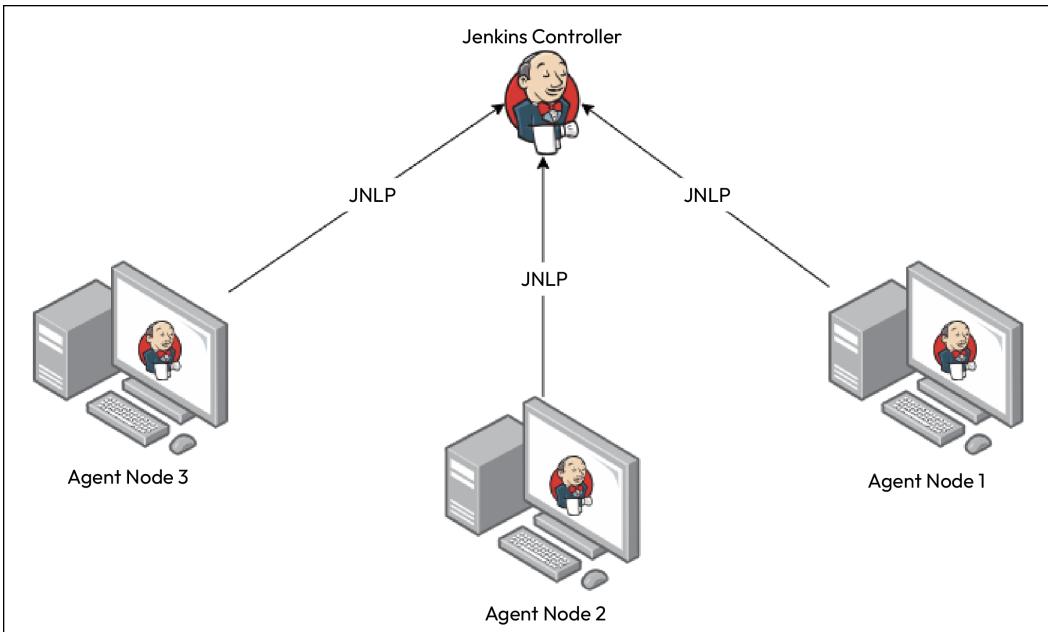


Figure 11.7 – Scalable Jenkins

In the preceding diagram, multiple static Jenkins agents connect to a Jenkins controller. Now, this architecture works well, but it needs to be more scalable. Modern DevOps emphasizes resource utilization, so we only want to roll out an agent machine when we want to build. Therefore, automating your builds to roll out an agent machine when required is a better way to do it. This might be overkill when rolling out new virtual machines, as it takes some minutes to provision a new VM, even when using a prebuilt image with Packer. A better alternative is to use a container.

Jenkins integrates quite well with Kubernetes, allowing you to run your build on a Kubernetes cluster. That way, whenever you trigger a build on Jenkins, Jenkins instructs Kubernetes to create a new agent container that will then connect with the controller machine and run the build within itself. This is *build on-demand* at its best. The following diagram shows this process in detail:

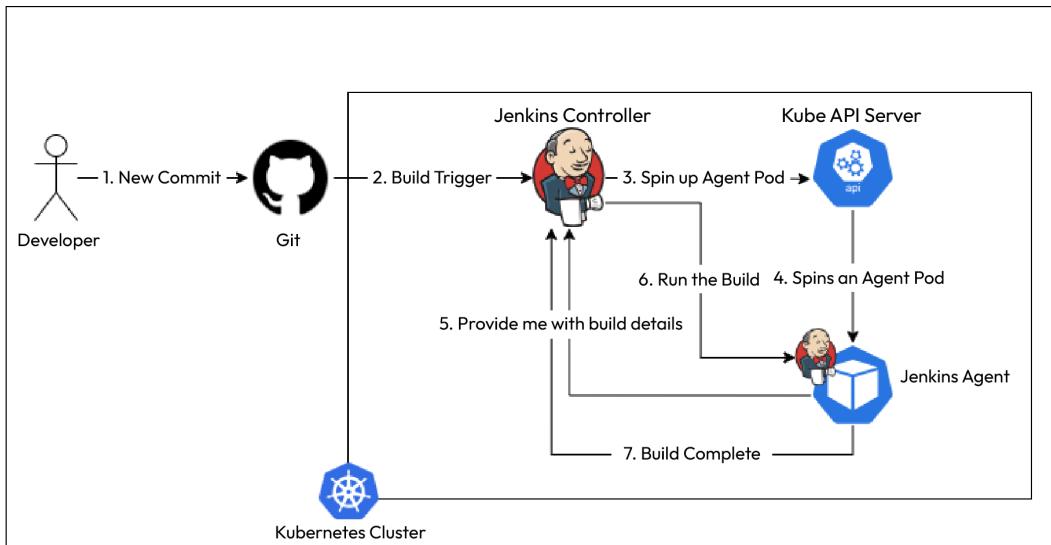


Figure 11.8 – Scalable Jenkins CI workflow

This sounds great, and we can go ahead and run this build, but there are issues with this approach. We must understand that the Jenkins controller and agents run as containers and aren't full-fledged virtual machines. Therefore, if we want to run a Docker build within the container, we must run the container in privileged mode. This isn't a security best practice, and your admin should already have turned that off. This is because running a container in privileged mode exposes your host filesystem to the container. A hacker who can access your container will have full access so that they can do whatever they want in your system.

To solve that problem, you can use a container build tool such as **Kaniko**. Kaniko is a build tool provided by Google that helps you build your containers without access to the Docker daemon, and you do not even need Docker installed in your container. It is a great way to run your builds within a **Kubernetes cluster** and create a scalable CI environment. It is effortless, not hacky, and provides a secure method of building your containers, as we will see in the subsequent sections.

This section will use **Google Kubernetes Engine (GKE)**. As mentioned previously, Google Cloud provides a free trial worth \$300 for 90 days. You can sign up at <https://cloud.google.com/free> if you have not already done so.

Spinning up Google Kubernetes Engine

Once you've signed up and are in your console, open the **Google Cloud Shell** CLI to run the following commands.

You need to enable the Kubernetes Engine API first using the following command:

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

To create a two-node autoscaling GKE cluster that scales from *one* to *five* nodes, run the following command:

```
$ gcloud container clusters create cluster-1 --num-nodes 2 \
--enable-autoscaling --min-nodes 1 --max-nodes 5 --zone us-central1-a
```

And that's it! The cluster will be up and running.

You must also clone the following GitHub repository for some of the exercises provided: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

Run the following command to clone the repository into your home directory and cd into the following directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch11/jenkins/jenkins-controller
```

We will use the **Jenkins Configuration as Code** feature to configure Jenkins as it is a declarative way of managing your configuration and is also GitOps-friendly. You need to create a simple YAML file with all the required configurations and then copy the file to the Jenkins controller after setting an environment variable that points to the file. Jenkins will then automatically configure all aspects defined in the YAML file on bootup.

Let's start by creating the `casc.yaml` file to define our configuration.

Creating the Jenkins CaC (JCasC) file

The **Jenkins CaC (JCasC)** file is a simple YAML file that helps us define Jenkins configuration declaratively. We will create a single `casc.yaml` file for that purpose, and I will explain parts of it. Let's start by defining **Jenkins Global Security**.

Configuring Jenkins Global Security

By default Jenkins is insecure – that is, if you fire up a vanilla Jenkins from the official Docker image and expose it, anyone can do anything with that Jenkins instance. To ensure that we protect it, we need the following configuration:

```
jenkins:
  remotingSecurity:
    enabled: true
  securityRealm:
```

```

local:
  allowsSignup: false
  users:
    - id: ${JENKINS_ADMIN_ID}
      password: ${JENKINS_ADMIN_PASSWORD}
  authorizationStrategy:
    globalMatrix:
      permissions:
        - "Overall/Administer:admin"
        - "Overall/Read:authenticated"

```

In the preceding configuration, we've defined the following:

- **remotingSecurity**: We've enabled this feature, which will secure the communication between the Jenkins controller and agents that we will create dynamically using Kubernetes.
- **securityRealm**: We've set the security realm to `local`, which means that the Jenkins controller itself will do all authentication and user management. We could have also offloaded this to an external entity such as LDAP:
 - `allowSignup`: This is set to `false`. This means you don't see a sign-up link on the Jenkins home page, and the Jenkins admin should manually create users.
 - `users`: We'll create a single user with `id` and `password` sourced from two environment variables called `JENKINS_ADMIN_ID` and `JENKINS_ADMIN_PASSWORD`, respectively.
- **authorizationStrategy**: We've defined a matrix-based authorization strategy where we provide administrator privileges to `admin` and read privileges to `authenticated` non-admin users.

Also, as we want Jenkins to execute all their builds in the agents and not the controller machine, we need to specify the following settings:

```

jenkins:
  systemMessage: "Welcome to Jenkins!"
  numExecutors: 0

```

We've set `numExecutors` to 0 to allow no builds on the controller and also set `systemMessage` on the Jenkins welcome screen.

Now that we've set up the security aspects of the Jenkins controller, we will configure Jenkins to connect with the Kubernetes cluster.

Connecting Jenkins with the cluster

We will install the Kubernetes plugin to connect the Jenkins controller with the cluster. We're doing this because we want Jenkins to dynamically spin up agents for builds as Kubernetes **pods**.

We will start by creating a `kubernetes` configuration under `jenkins.clouds`, as follows:

```
jenkins
  clouds:
    - kubernetes:
        serverUrl: "https://<kubernetes_control_plane_ip>"  
        jenkinsUrl: "http://jenkins-service:8080"  
        jenkinsTunnel: "jenkins-service:50000"  
        skipTlsVerify: false  
        useJenkinsProxy: false  
        maxRequestsPerHost: 32  
        name: "kubernetes"  
        readTimeout: 15  
        podLabels:  
          - key: jenkins  
            value: agent  
...  
...
```

As we have a placeholder called `<kubernetes_control_plane_ip>` within the configuration, we must replace this with the Kubernetes control plane's IP address. Run the following command to fetch the control plane's IP address:

```
$ kubectl cluster-info | grep "control plane"
Kubernetes control plane is running at https://35.224.6.58
```

Now, replace the `<kubernetes_control_plane_ip>` placeholder with the actual IP address you obtained from the preceding command by using the following command:

```
$ sed -i 's/<kubernetes_control_plane_ip>/actual_ip/g' casc.yaml
```

Let's look at each attribute in the config file:

- `serverUrl`: This denotes the Kubernetes control plane server URL, allowing the Jenkins controller to communicate with the Kubernetes API server.
- `jenkinsUrl`: This denotes the Jenkins controller URL. We've set it to `http://jenkins-service:8080`.
- `jenkinsTunnel`: This describes how the agent pods will connect with the Jenkins controller. As the JNLP port is 50000, we've set it to `jenkins-service:50000`.
- `podLabels`: We've also set up some pod labels, `key=jenkins` and `value=agent`. These will be set on the agent pods.

Other attributes are also set to their default values.

Every Kubernetes cloud configuration consists of multiple pod templates describing how the agent pods will be configured. The configuration looks like this:

```
- kubernetes:  
  ...  
    templates:  
      - name: "jenkins-agent"  
        label: "jenkins-agent"  
        hostNetwork: false  
        nodeUsageMode: "NORMAL"  
        serviceAccount: "jenkins"  
        imagePullSecrets:  
          - name: regcred  
        yamlMergeStrategy: "override"  
        containers:  
          ...
```

Here, we've defined the following:

- The template's name and label. We set both to jenkins-agent.
- hostNetwork: This is set to false as we don't want the container to interact with the host network.
- serviceAccount: We've set this to jenkins as we want to use this service account to interact with Kubernetes.
- imagePullSecrets: We have also provided an image pull secret called regcred to authenticate with the container registry to pull the jnlp image.

Every pod template also contains a **container template**. We can define that using the following configuration:

```
...  
  containers:  
    - name: jnlp  
      image: "<your_dockerhub_user>/jenkins-jnlp-kaniko"  
      workingDir: "/home/jenkins/agent"  
      command: ""  
      args: ""  
      livenessProbe:  
        failureThreshold: 1  
        initialDelaySeconds: 2  
        periodSeconds: 3  
        successThreshold: 4  
        timeoutSeconds: 5  
      volumes:  
        - secretVolume:  
            mountPath: /kaniko/.docker  
            secretName: regcred
```

Here, we have specified the following:

- `name`: Set to `jnlp`.
- `image`: Here, we've specified the *Docker agent image* we will build in the next section. Ensure that you replace the `<your_dockerhub_user>` placeholder with your Docker Hub user by using the following command:

```
$ sed -i 's/<your_dockerhub_user>/actual_dockerhub_user/g' casc.yaml
```

- `workingDir`: Set to `/home/jenkins/agent`.
- We've set the `command` and `args` fields to blank as we don't need to pass them.
- `livenessProbe`: We've defined a liveness probe for the agent pod.
- `volumes`: We've mounted the `regcred` secret to the `kaniko/.docker` file as a volume. As `regcred` contains the Docker registry credentials, Kaniko will use this to connect with your container registry.

Now that our configuration file is ready, we'll go ahead and install Jenkins in the next section.

Installing Jenkins

As we're running on a Kubernetes cluster, we only need the latest official Jenkins image from Docker Hub. We will customize the image according to our requirements.

The following `Dockerfile` file will help us create the image with the required plugins and the initial configuration:

```
FROM jenkins/jenkins
ENV CASC_JENKINS_CONFIG /usr/local/casc.yaml
ENV JAVA_OPTS -Djenkins.install.runSetupWizard=false
COPY casc.yaml /usr/local/casc.yaml
COPY plugins.txt /usr/share/jenkins/ref/plugins.txt
RUN jenkins-plugin-cli --plugin-file /usr/share/jenkins/ref/plugins.txt
```

The `Dockerfile` starts from the Jenkins base image. Then, we declare two environment variables – `CASC_JENKINS_CONFIG`, which points to the `casc.yaml` file we defined in the previous section, and `JAVA_OPTS`, which tells Jenkins not to run the setup wizard. Then, we copy the `casc.yaml` and `plugins.txt` files to their respective directories within the Jenkins container. Finally, we run `jenkins-plugin-cli` on the `plugins.txt` file, which installs the required plugins.

The `plugins.txt` file contains a list of all Jenkins plugins that we will need in this setup.

Tip

You can customize and install more plugins for the controller image based on your requirements by updating the `plugins.txt` file.

Let's build the image from the `Dockerfile` file using the following command:

```
$ docker build -t <your_dockerhub_user>/jenkins-controller-kaniko .
```

Now that we've built the image, use the following command to log in and push the image to Docker Hub:

```
$ docker login  
$ docker push <your_dockerhub_user>/jenkins-controller-kaniko
```

We must also build the Jenkins agent image to run our builds. Remember that Jenkins agents need all the supporting tools you need to run your builds. You can find the resources for the agents in the following directory:

```
$ cd ~/modern-devops/ch11/jenkins/jenkins-agent
```

We will use the following `Dockerfile` to do that:

```
FROM gcr.io/kaniko-project/executor:v1.13.0 as kaniko  
FROM jenkins/inbound-agent  
COPY --from=kaniko /kaniko /kaniko  
WORKDIR /kaniko  
USER root
```

This `Dockerfile` uses a multi-stage build to take the kaniko base image and copy the kaniko binary from the kaniko base image to the inbound-agent base image. Let's go ahead and build and push the container using the following commands:

```
$ docker build -t <your_dockerhub_user>/jenkins-jnlp-kaniko .  
$ docker push <your_dockerhub_user>/jenkins-jnlp-kaniko
```

To deploy Jenkins on our Kubernetes cluster, we will first create a `jenkins` service account. A Kubernetes **service account** resource helps pods authenticate with the **Kubernetes API server**. We will give the service account permission to interact with the Kubernetes API server as `cluster-admin` using a cluster role binding. A Kubernetes **ClusterRoleBinding** resource helps provide permissions to a service account to perform certain actions in the Kubernetes cluster. The `jenkins-sa-crb.yaml` manifest describes this. To access these resources, run the following command:

```
$ cd ~/modern-devops/ch11/jenkins/jenkins-controller
```

To apply the manifest, run the following command:

```
$ kubectl apply -f jenkins-sa-crb.yaml
```

The next step involves creating a **PersistentVolumeClaim** resource to store Jenkins data to ensure that the Jenkins data persists beyond the pod's life cycle and will exist even when we delete the pod.

To apply the manifest, run the following command:

```
$ kubectl apply -f jenkins-pvc.yaml
```

Then, we will create a Kubernetes **Secret** called `regcred` to help the Jenkins pod authenticate with the Docker registry. Use the following command to do so:

```
$ kubectl create secret docker-registry regcred --docker-username=<username> \
--docker-password=<password> --docker-server=https://index.docker.io/v1/
```

Now, we'll define a **Deployment** resource, `jenkins-deployment.yaml`, that will run the Jenkins container. The pod uses the `jenkins` service account and defines a **PersistentVolume** resource called `jenkins-pv-storage` using the **PersistentVolumeClaim** resource called `jenkins-pv-claim` that we defined. We define the Jenkins container that uses the Jenkins controller image we created. It exposes HTTP port 8080 for the *Web UI*, and port 50000 for *JNLP*, which the agents would use to interact with the Jenkins controller. We will also mount the `jenkins-pv-storage` volume to `/var/jenkins_home` to persist the Jenkins data beyond the pod's life cycle. We specify `regcred` as the `imagePullSecret` attribute in the pod image. We also use `initContainer` to assign ownership to `jenkins` for `/var/jenkins_home`.

As the file contains placeholders, replace `<your_dockerhub_user>` with your Docker Hub user and `<jenkins_admin_pass>` with a Jenkins admin password of your choice using the following commands:

```
$ sed -i 's/<your_dockerhub_user>/actual_dockerhub_user/g' jenkins-deployment.yaml
```

Apply the manifest using the following command:

```
$ kubectl apply -f jenkins-deployment.yaml
```

As we've created the deployment, we can expose the deployment on a **LoadBalancer** Service using the `jenkins-svc.yaml` manifest. This service exposes ports 8080 and 50000 on a load balancer. Use the following command to apply the manifest:

```
$ kubectl apply -f jenkins-svc.yaml
```

Let's get the service to find the external IP to use that to access Jenkins:

```
$ kubectl get svc jenkins-service
NAME           EXTERNAL-IP          PORT(S)
jenkins-service  LOAD_BALANCER_EXTERNAL_IP  8080,50000
```

Now, to access the service, go to `http://<LOAD_BALANCER_EXTERNAL_IP>:8080` in your browser window:

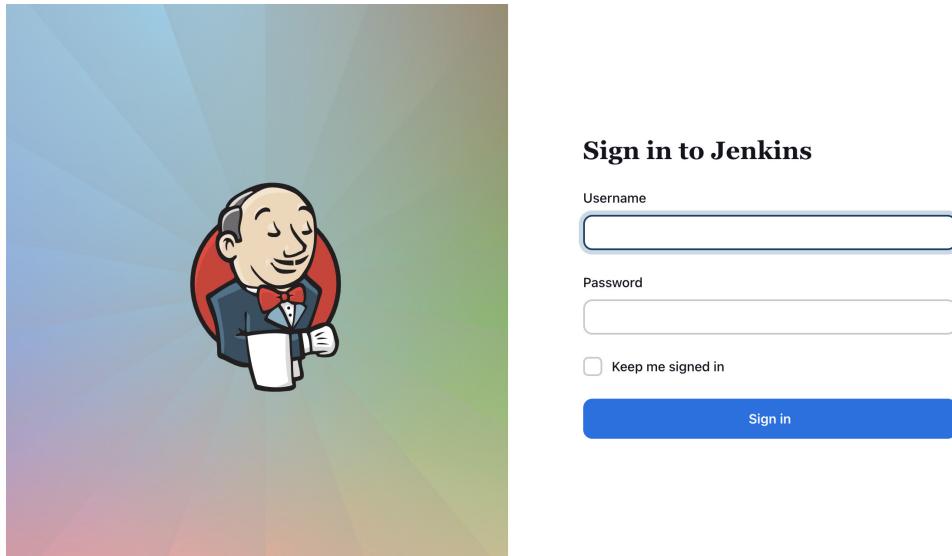


Figure 11.9 – Jenkins login page

As we can see, we're greeted with a login page. This means Global Security is working correctly. Let's log in using the admin username and password we set:

The image shows the Jenkins home page after logging in. The top navigation bar includes the Jenkins logo, a search bar, and user information for "admin". Below the header, the dashboard shows links for "New Item", "People", "Build History", "Manage Jenkins", and "My Views". A main section titled "Welcome to Jenkins!" displays a message about starting a software project. It includes a "Create a job" button. At the bottom, there are sections for "Build Queue" (empty) and "Build Executor Status". The footer indicates "REST API" and "Jenkins 2.418".

Figure 11.10 – Jenkins home page

As we can see, we've successfully logged in to Jenkins. Now, let's go ahead and create our first Jenkins job.

Running our first Jenkins job

Before we create our first job, we'll have to prepare our repository to run the job. We will reuse the `mdo-posts` repository for this. We will copy a `build.sh` file to the repository, which will build the container image for the `posts` microservice and push it to Docker Hub.

The `build.sh` script takes `IMAGE_ID` and `IMAGE_TAG` as arguments. It passes them to the **Kaniko** executor script, which builds the container image using the `Dockerfile` and pushes it to Docker Hub using the following code:

```
IMAGE_ID=$1 && \
IMAGE_TAG=$2 && \
export DOCKER_CONFIG=/kaniko/.dockerconfig && \
/kaniko/executor \
--context $(pwd) \
--dockerfile $(pwd)/Dockerfile \
--destination $IMAGE_ID:$IMAGE_TAG \
--force
```

We will need to copy this file to our local repository using the following commands:

```
$ cp ~/modern-devops/ch11/jenkins/jenkins-agent/build.sh ~/mdo-posts/
```

Once you've done this, `cd` into your local repository – that is, `~/mdo-posts` – and commit and push your changes to GitHub. Once you've done this, you'll be ready to create a job in Jenkins.

To create a new job in Jenkins, go to the Jenkins home page and select **New Item | Freestyle Job**. Provide a job name (preferably the same as the Git repository name), then click **Next**.

Click on **Source Code Management**, select **Git**, and add your Git repository URL, as shown in the following example. Specify the branch from where you want to build:

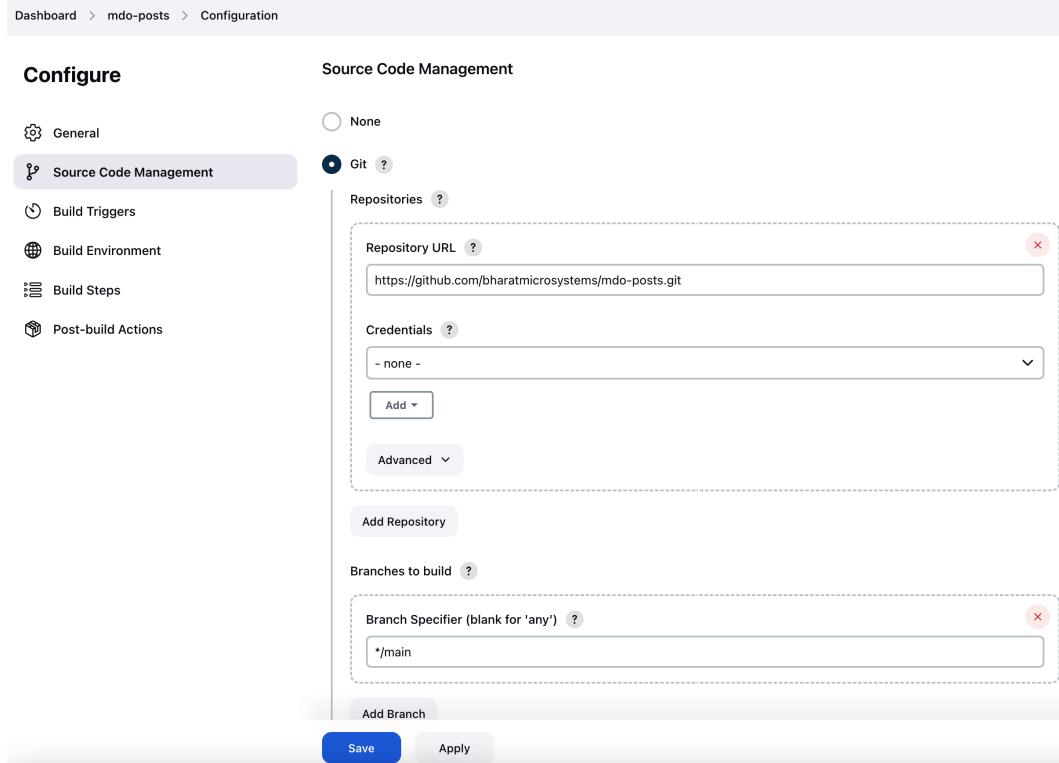


Figure 11.11 – Jenkins Source Code Management configuration

Go to **Build Triggers**, select **Poll SCM**, and add the following details:

The screenshot shows the Jenkins 'Build Triggers' configuration page. Under the 'Poll SCM' section, the 'Schedule' dropdown is set to 'H * * * *'. A warning message at the bottom states: '⚠ Do you really mean "every minute" when you say "* * * *"? Perhaps you meant "H * * * *" to poll once per hour'. Below the schedule, there is an unchecked checkbox for 'Ignore post-commit hooks'.

Figure 11.12 – Jenkins – Build Triggers configuration

Then, click on **Build | Add Build Step | Execute shell**. The **Execute shell** build step executes a sequence of shell commands on the Linux CLI. In this example, we're running the `build.sh` script with the `<your_dockerhub_user>/<image>` argument and the `image tag`. Change the details according to your requirements. Once you've finished, click **Save**:

Build Steps

The screenshot shows the Jenkins 'Execute shell' build step configuration. The 'Command' field contains the command: `chmod +x build.sh && ./build.sh bharamicrosystems/mdo-posts latest`. There is an 'Advanced' button with a dropdown arrow, and a 'Add build step ▾' button at the bottom left.

Figure 11.13 – Jenkins – Execute shell configuration

Now, we're ready to build this job. To do so, you can either go to your job configuration and click **Build Now** or push a change to GitHub. You should see something like the following:

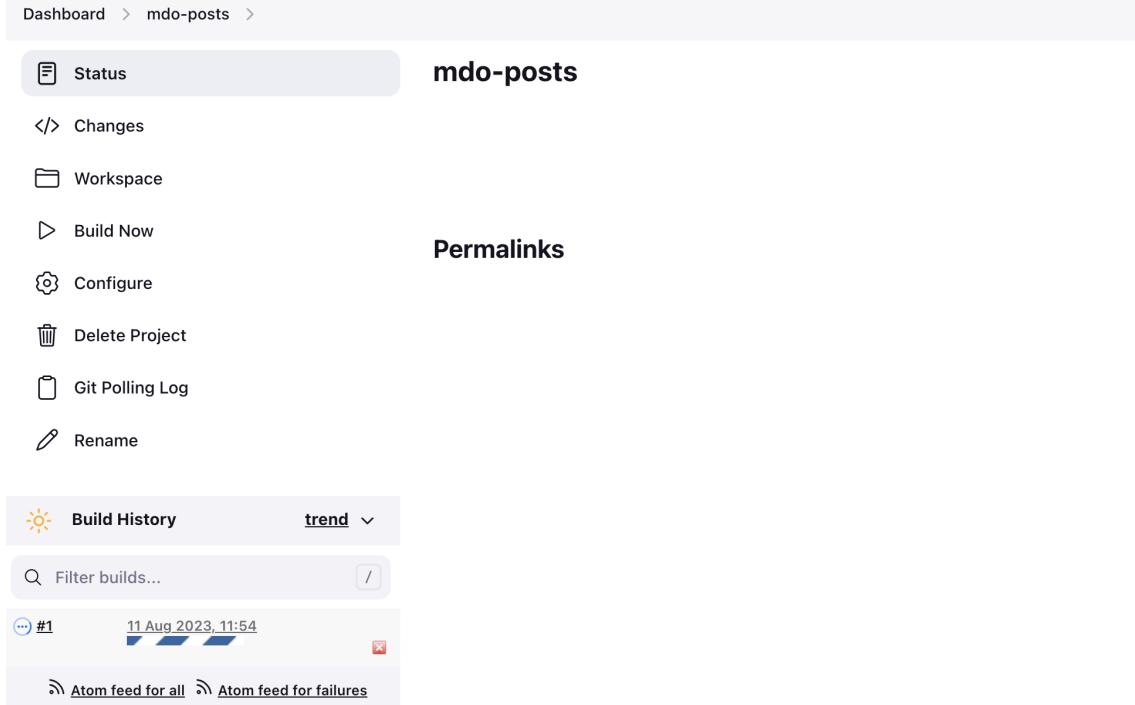


Figure 11.14 – Jenkins job page

Jenkins will successfully create an agent pod in Kubernetes, where it will run this job, and soon, the job will start building. Click **Build | Console Output**. If everything is OK, you'll see that the build was successful and that Jenkins has built the **posts** service and executed a unit test before pushing the Docker image to the registry:

```
[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run: pip install --upgrade pip
[36mINFO [0m[0030] Taking snapshot of full filesystem...
[36mINFO [0m[0035] EXPOSE 5000
[36mINFO [0m[0035] Cmd: EXPOSE
[36mINFO [0m[0035] Adding exposed port: 5000/tcp
[36mINFO [0m[0035] COPY .
[36mINFO [0m[0036] Taking snapshot of files...
[36mINFO [0m[0036] RUN python app.test.py
[36mINFO [0m[0036] Cmd: /bin/sh
[36mINFO [0m[0036] Args: [-c python app.test.py]
[36mINFO [0m[0036] Running: [/bin/sh -c python app.test.py]
.....
-----
Ran 8 tests in 0.059s

OK
[36mINFO [0m[0037] Taking snapshot of full filesystem...
[36mINFO [0m[0038] CMD ["flask", "run"]
[36mINFO [0m[0038] Pushing image to bharamicrosystems/mdo-posts:latest
[36mINFO [0m[0047] Pushed index.docker.io/bharamicrosystems/mdo-
posts@sha256:588d8a408b365580bec4864690e7a42401ff2a3374d500eefc0e93333c3bfb05
Finished: SUCCESS
```

Figure 11.15 – Jenkins console output

With that, we're able to run a Docker build using a scalable Jenkins server. As we can see, we've set up polling on the SCM settings to look for changes every minute and build the job if we detect any. However, this is resource-intensive and does not help in the long run. Just imagine that you have hundreds of jobs interacting with multiple GitHub repositories, and the Jenkins controller is polling them every minute. A better approach would be if GitHub could trigger a **post-commit webhook** on Jenkins. Here, Jenkins can build the job whenever there are changes in the repository. We'll look at that scenario in the next section.

Automating a build with triggers

The best way to allow your CI build to trigger when you make changes to your code is to use a post-commit webhook. We looked at such an example in the GitHub Actions workflow. Let's try to automate the build with triggers in the case of Jenkins. We'll have to make some changes on both the Jenkins and the GitHub sides to do so. We'll deal with Jenkins first; then, we'll configure GitHub.

Go to **Job configuration | Build Triggers** and make the following changes:

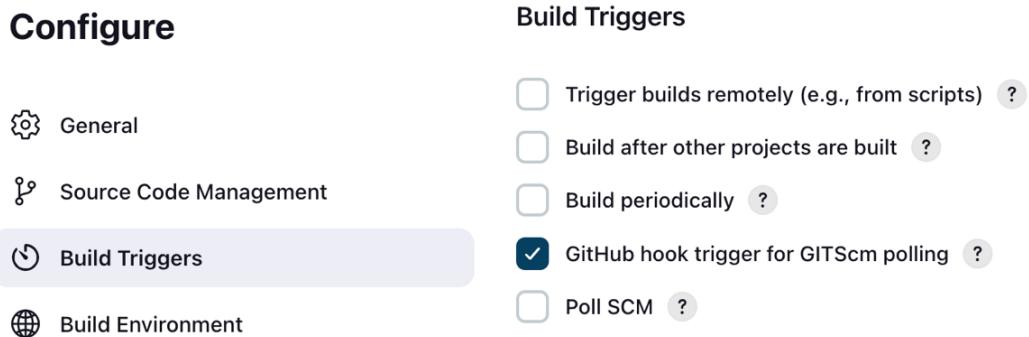


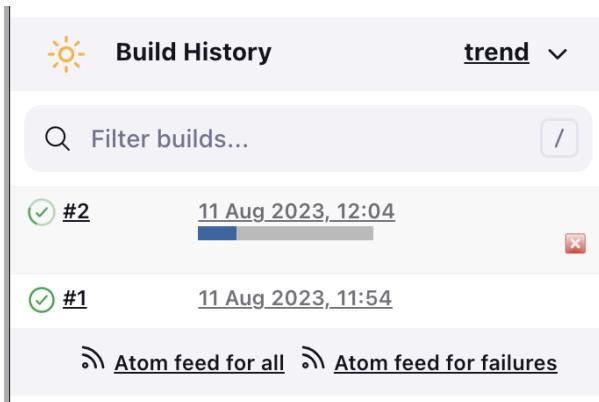
Figure 11.16 – Jenkins GitHub hook trigger

Save the configuration by clicking **Save**. Now, go to your GitHub repository, click **Settings | Webhooks | Add Webhook**, and add the following details. Then, click **Add Webhook**:

The screenshot shows the GitHub 'Webhooks / Manage webhook' page. At the top, it says 'We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in our developer documentation.' Below this, there are fields for 'Payload URL *' containing 'http://[YOUR_JENKINS_IP]:8080/github-webhook/' and 'Content type' set to 'application/json'. There is also a 'Secret' field which is empty.

Figure 11.17 – GitHub webhook

Now, push a change to the repository. The job on Jenkins will start building:



The screenshot shows the Jenkins Build History interface. At the top, there is a search bar labeled "Filter builds..." with a magnifying glass icon and a dropdown menu with a slash icon. Below the search bar, there are two build entries. Each entry has a green checkmark icon and a build number. The first entry is "#2" with the timestamp "11 Aug 2023, 12:04" and a progress bar indicating it is still running. The second entry is "#1" with the timestamp "11 Aug 2023, 11:54". At the bottom of the screen, there are two RSS feed icons labeled "Atom feed for all" and "Atom feed for failures".

Figure 11.18 – Jenkins GitHub webhook trigger

This is automated build triggers in action. Jenkins is one of the most popular open source CI tools on the market. The most significant advantage of it is that you can pretty much run it anywhere. However, it does come with some management overhead. You may have noticed how simple it was to start with GitHub Actions, but Jenkins is slightly more complicated.

Several other SaaS platforms offer CI and CD as a service. For instance, if you are running on AWS, you'd get their inbuilt CI with **AWS Code Commit** and **Code Build**; Azure provides an entire suite of services for CI and CD in their **Azure DevOps** offering; and GCP provides **Cloud Build** for that job.

CI follows the same principle, regardless of the tooling you choose to implement. It is more of a process and a cultural change within your organization. Now, let's look at some of the best practices regarding CI.

Building performance best practices

CI is an ongoing process, so you will have a lot of parallel builds running within your environment at a given time. In such situations, we can optimize them using several best practices.

Aim for faster builds

The faster you can complete your build, the quicker you will get feedback and run your next iteration. A slow build slows down your development team. Take steps to ensure that builds are faster. For example, in Docker's case, it makes sense to use smaller base images as it will download the code from the image registry every time it does a build. Using a single base image for most builds will also speed up your build time. Using tests will help, but make sure that they aren't long-running. We want to avoid a CI build that runs for hours. Therefore, it would be good to offload long-running tests into another job or use a pipeline. Run activities in parallel if possible.

Always use post-commit triggers

Post-commit triggers help your team significantly. They will not have to log in to the CI server and trigger the build manually. That completely decouples your development team from CI management.

Configure build reporting

You don't want your development team to log in to the CI tool and check how the build runs. Instead, all they want to know is the result of the build and the build logs. Therefore, you can configure build reporting to send your build status via email or, even better, using a **Slack** channel.

Customize the build server size

Not all builds work the same in similar kinds of build machines. You may want to choose machines based on what suits your build environment best. If your builds tend to consume more CPU than memory, it will make sense to choose such machines to run your builds instead of the standard ones.

Ensure that your builds only contain what you need

Builds move across networks. You download base images, build your application image, and push that to the container registry. Bloated images not only take a lot of network bandwidth and time to transmit but also make your build vulnerable to security issues. Therefore, it is always best practice to only include what you require in the build and avoid bloat. You can use Docker's **multi-stage builds** for these kinds of situations.

Parallelize your builds

Run tests and build processes concurrently to reduce overall execution time. Leverage distributed systems or cloud-based CI/CD platforms for scalable parallelization, allowing you to handle larger workloads efficiently.

Make use of caching

Cache dependencies and build artifacts to prevent redundant downloads and builds, saving valuable time. Implement caching mechanisms such as Docker layer caching or use your package manager's built-in caches to minimize data transfer and build steps.

Use incremental building

Configure your CI/CD pipeline to perform incremental builds, rebuilding only what has changed since the last build. Maintain robust version control practices to accurately track and identify changes.

Optimize testing

Prioritize and optimize tests by running quicker unit tests before slower integration or end-to-end tests. Use testing frameworks such as TestNG, JUnit, or PyTest to categorize and parallelize tests effectively.

Use artifact management

Efficiently store and manage build artifacts, preferably in a dedicated artifact repository such as Artifactory or Nexus. Implement artifact versioning and retention policies to maintain a clean artifact repository.

Manage application dependencies

Keep a clean and minimal set of dependencies to reduce build and test times. Regularly update dependencies to benefit from performance improvements and security updates.

Utilize Infrastructure as Code

Utilize **Infrastructure as Code (IaC)** to provision and configure build and test environments consistently. Optimize IaC templates to minimize resource utilization, ensuring efficient resource allocation.

Use containerization to manage build and test environments

Containerize applications and utilize container orchestration tools such as Kubernetes to manage test environments efficiently. Leverage container caching to accelerate image builds and enhance resource utilization.

Utilize cloud-based CI/CD

Consider adopting cloud-based CI/CD services such as AWS CodePipeline, Google Cloud Build, Azure DevOps, or Travis CI for enhanced scalability and performance. Harness on-demand cloud resources to expand parallelization capabilities and adapt to varying workloads.

Monitor and profile your CI/CD pipelines

Implement performance monitoring and profiling tools to identify bottlenecks and areas for improvement within your CI/CD pipeline. Regularly analyze build and test logs to gather insights for optimizing performance.

Pipeline optimization

Continuously review and optimize your CI/CD pipeline configuration for efficiency and relevance. Remove unnecessary steps or stages that do not contribute significantly to the process.

Implement automated cleanup

Implement automated cleanup routines to remove stale artifacts, containers, and virtual machines, preventing resource clutter. Regularly purge old build artifacts and unused resources to maintain a tidy environment.

Documentation and training

Document best practices and performance guidelines for your CI/CD processes, ensuring that the entire team follows these standards consistently. Provide training and guidance to team members to empower them to implement and maintain these optimization strategies effectively.

By implementing these strategies, you can significantly enhance the speed, efficiency, and reliability of your CI/CD pipeline, ultimately leading to smoother software development and delivery processes. These are some of the best practices at a high level, and they are not exhaustive, but they are good enough so that you can start optimizing your CI environment.

Summary

This chapter covered CI, and you understood the need for CI and the basic CI workflow for a container application. We then looked at GitHub Actions, which we can use to build an effective CI pipeline. Next, we looked at the Jenkins open source offering and deployed a scalable Jenkins on Kubernetes with Kaniko, setting up a Jenkins controller-agent model. We then understood how to use hooks for automating builds, both in the GitHub Actions-based workflow and the Jenkins-based workflow. Finally, we learned about build performance best practices and dos and don'ts.

By now, you should be familiar with CI and its nuances, along with the various tooling you can use to implement it.

In the next chapter, we will delve into continuous deployment/delivery in the container world.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following are CI tools? (Choose three)
 - A. Jenkins
 - B. GitHub Actions
 - C. Kubernetes
 - D. AWS Code Build

2. It is a best practice to configure post-commit triggers. (True/False)
3. Jenkins is a SaaS-based CI tool. (True/False)
4. Kaniko requires Docker to build your containers. (True/False)
5. Jenkins agents are required for which of the following reasons? (Choose three)
 - A. They make builds more scalable
 - B. They help offload the management function from the Jenkins controller
 - C. They allow for parallel builds
 - D. They keep the Jenkins controller less busy
6. Which of the following is required for a scalable Jenkins server, as described in the example in this chapter? (Choose three)
 - A. Kubernetes cluster
 - B. Jenkins controller node
 - C. Jenkins agent node
 - D. Credentials to interact with the container registry

Answers

The following are the answers to this chapter's questions:

1. A, B, D
2. True
3. False
4. False
5. A, C, D
6. A, B, D

12

Continuous Deployment/ Delivery with Argo CD

In the previous chapter, we looked at one of the key aspects of modern DevOps – **continuous integration (CI)**. CI is the first thing most organizations implement when they embrace DevOps, but things don't end with CI, which only delivers a tested build in an artifact repository. Instead, we would also want to deploy the artifact to our environments. In this chapter, we'll implement the next part of the DevOps toolchain – **continuous deployment/delivery (CD)**.

In this chapter, we're going to cover the following main topics:

- The importance of CD and automation
- CD models and tools
- The Blog App and its deployment configuration
- Continuous declarative IaC using an Environment repository
- Introduction to Argo CD
- Installing and setting up Argo CD
- Managing sensitive configurations and secrets
- Deploying the sample Blog App

Technical requirements

In this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine (GKE)**, for the exercises. At the time of writing, **Google Cloud Platform (GCP)** provides a free \$300 trial for 90 days, so you can go ahead and sign up for one at <https://console.cloud.google.com/>.

You will also need to clone the following GitHub repository for some exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices>.

Run the following command to clone the repository into your home directory, and `cd` into the `ch12` directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
$ cd modern-devops/ch12
```

So, let's get started!

The importance of CD and automation

CD forms the Ops part of your DevOps toolchain. So, while your developers are continuously building and pushing code and your CI pipeline is building, testing, and publishing the builds to your artifact repository, the Ops team will deploy the build to the test and staging environments. The QA team is the gatekeeper that will ensure that the code meets a certain quality, and only then will the Ops team deploy the code to production.

Now, for organizations implementing only the CI part, the rest of the activities are manual. For example, operators will pull the artifacts and run commands to do the deployments manually. Therefore, your deployment's velocity will depend on the availability of your Ops team to do it. As the deployments are manual, the process is error-prone, and human beings tend to make mistakes in repeatable jobs.

One of the essential principles of modern DevOps is to avoid **toil**. Toil is nothing but repeatable jobs that developers and operators do day in and day out, and all of that toil can be removed by automation. This will help your team focus on the more important things at hand.

With **continuous delivery**, standard tooling can deploy code to higher environments based on certain gate conditions. CD pipelines will trigger when a tested build arrives at the artifact repository or, in the case of GitOps, if any changes are detected in the Environment repository. The pipeline then decides, based on a set configuration, where and how to deploy the code. It also establishes whether manual checks are required, such as raising a change ticket and checking whether it's approved.

While **continuous deployment** and delivery are often confused with being the same thing, there is a slight difference between them. Continuous delivery enables your team to deliver tested code in your environment based on a human trigger. So, while you don't have to do anything more than click a button to do a deployment to production, it would still be initiated by someone at a convenient time (a maintenance window). Continuous deployments go a step further when they integrate with the CI process and will start the deployment process as soon as a new tested build is available for them to consume. There is no need for manual intervention, and continuous deployment will only stop in case of a failed test.

The monitoring tool forms the next part of the DevOps toolchain. The Ops team can learn from managing their production environment and provide developers with feedback regarding what they need to do better. That feedback ends up in the development backlog, and they can deliver it as features

in future releases. That completes the cycle, and now you have your team churning out a technology product continuously.

CD offers several advantages. Some of them are as follows:

- **Faster time to market:** CD and CI reduce the time it takes to deliver new features, enhancements, and bug fixes to end users. This agility can give your organization a competitive edge by allowing you to respond quickly to market demands.
- **Reduced risk:** By automating the deployment process and frequently pushing small code changes, you minimize the risk of large, error-prone deployments. Bugs and issues are more likely to be caught early, and rollbacks can be less complex.
- **Improved code quality:** Frequent automated testing and quality checks are an integral part of CD and CI. This results in higher code quality as developers are encouraged to write cleaner, more maintainable code. Any issues are caught and addressed sooner in the development process.
- **Enhanced collaboration:** CD and CI encourage collaboration between development and operations teams. It breaks down traditional silos and encourages cross-functional teamwork, leading to better communication and understanding.
- **Efficiency and productivity:** Automation of repetitive tasks, such as testing, building, and deployment, frees up developers' time to focus on more valuable tasks, such as creating new features and improvements.
- **Customer feedback:** CD allows you to gather feedback from real users more quickly. By deploying small changes frequently, you can gather user feedback and adjust your development efforts accordingly, ensuring that your product better meets user needs.
- **Continuous improvement:** CD promotes a culture of continuous improvement. By analyzing data on deployments and monitoring, teams can identify areas for enhancement and iterate on their processes.
- **Better security:** Frequent updates mean that security vulnerabilities can be addressed promptly, reducing the window of opportunity for attackers. Security checks can be automated and integrated into the CI/CD pipeline.
- **Reduced manual intervention:** CD minimizes the need for manual intervention in the deployment process. This reduces the potential for human error and streamlines the release process.
- **Scalability:** As your product grows and the number of developers and your code base complexity increases, CD can help maintain a manageable development process. It scales effectively by automating many of the release and testing processes.
- **Cost savings:** Although implementing CI/CD requires an initial investment in tools and processes, it can lead to cost savings in the long run by reducing the need for extensive manual testing, lowering deployment-related errors, and improving resource utilization.

- **Compliance and auditing:** For organizations with regulatory requirements, CD can improve compliance by providing a detailed history of changes and deployments, making it easier to track and audit code changes.

It's important to note that while CD and CI offer many advantages, they also require careful planning, infrastructure, and cultural changes to be effective.

There are several models and tools available to implement CD. We'll have a look at some of them in the next section.

CD models and tools

A typical CI/CD workflow looks as described in the following figure and the subsequent steps:

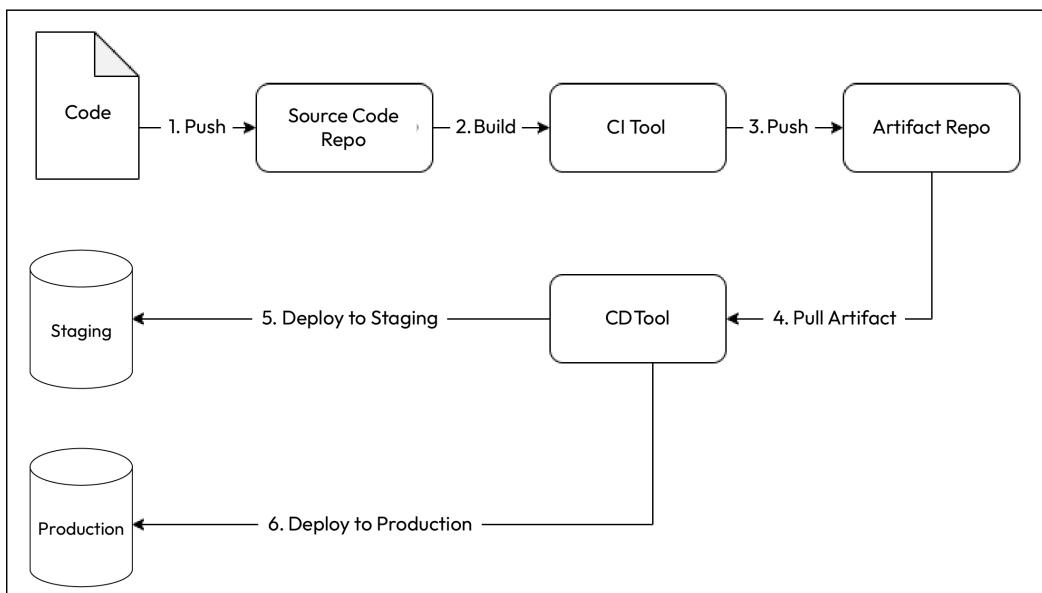


Figure 12.1 – CI/CD workflow

1. Your developers write code and push it to a code repository (typically a Git repository).
2. Your CI tool builds the code, runs a series of tests, and pushes the tested build to an artifact repository. Your CD tool then picks up the artifact and deploys it to your test and staging environments. Based on whether you want to do continuous deployment or delivery, it automatically deploys the artifact to the production environment.

Well, what do you choose for a delivery tool? Let's look at the example we covered in *Chapter 11, Continuous Integration*. We picked up the **posts** microservice app and used a CI tool such as GitHub Actions/Jenkins that uses **Docker** to create a container out of it and push it to our **Docker Hub** container registry. Well, we could have used the same tool for deploying to our environment.

For example, if we wanted to deploy to **Kubernetes**, it would have been a simple YAML update and `kubectl apply`. We could easily do this with any of those tools, but we chose not to do it. Why? The answer is simple – CI tools are meant for CI, and if you want to use them for anything else, you'll get stuck at a certain point. That does not mean that you cannot use these tools for CD. It will only suit a few use cases based on the deployment model you follow.

Several deployment models exist based on your application, technology stack, customers, risk appetite, and cost consciousness. Let's look at some of the popular deployment models that are used within the industry.

Simple deployment model

The **simple deployment model** is one of the most straightforward of all: you deploy the required version of your application after removing the old one. It completely replaces the previous version, and rolling back involves redeploying the older version after removing the deployed one:

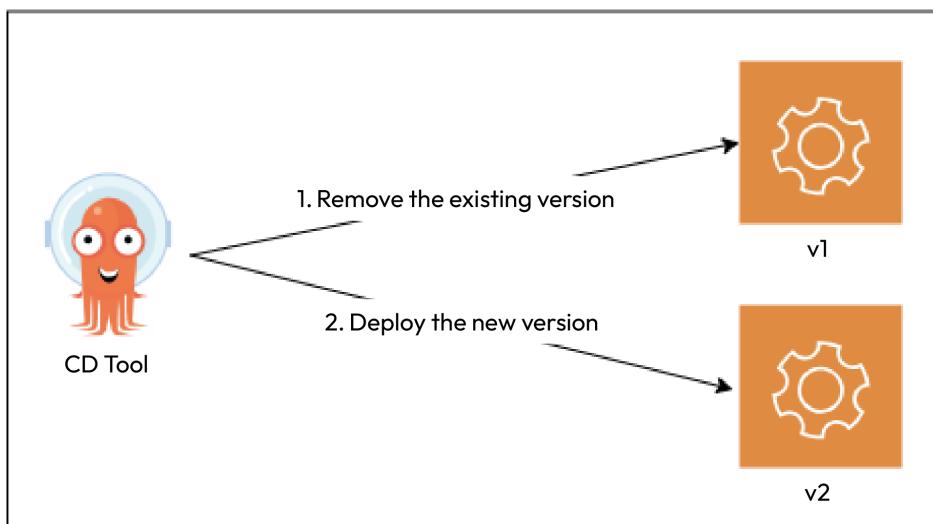


Figure 12.2 – Simple deployment model

As it is a simple way of deploying things, you can manage this using a CI tool such as **Jenkins** or **GitHub Actions**. However, the simple deployment model is not the most desired deployment method because of some inherent risks. This kind of change is disruptive and typically needs downtime. This means your service would remain unavailable to your customers for the upgrade period. It might be OK for organizations that do not have users 24/7, but disruptions eat into the **service-level objectives (SLOs)** and **service-level agreements (SLAs)** of global organizations. Even if there isn't one, they hamper customer experience and the organization's reputation.

Therefore, to manage such kinds of situations, we have some complex deployment models.

Complex deployment models

Complex deployment models, unlike simple deployment models, try to minimize disruptions and downtimes within the application and make rolling out releases more seamless to the extent that most users don't even notice when the upgrade is being conducted. Two main kinds of complex deployments are prevalent in the industry; let's take a look.

Blue/Green deployments

Blue/Green deployments (also known as **Red/Black deployments**) roll out the new version (*Green*) in addition to the existing version (*Blue*). You can then do sanity checks and other activities with the latest version to ensure that everything is good to go. Then, you can switch traffic from the old to the new version and monitor for any issues. If you encounter problems, you switch back traffic to the old version. Otherwise, you keep the latest version running and remove the old version:

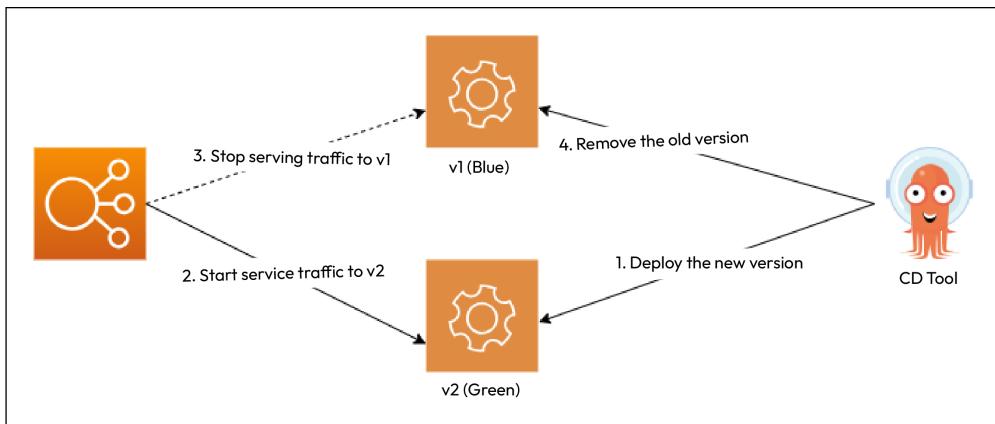


Figure 12.3 – Blue/Green deployments

You can take Blue/Green deployments to the next level using canary deployments.

Canary deployments and A/B testing

Canary deployments are similar to Blue/Green deployments but are generally utilized for risky upgrades. So, like Blue/Green deployments, we deploy the new version alongside the existing one. Instead of switching all traffic to the latest version at once, we only switch traffic to a small subset of users. As we do that, we can understand from our logs and user behaviors whether the switchover is causing any issues. This is called **A/B testing**. When we do A/B testing, we can target a specific group of users based on location, language, age group, or users who have opted to test Beta versions of a product. That will help organizations gather feedback without disrupting general users and make changes to the product once they're satisfied with what they are rolling out. You can make the release generally available by switching over the total traffic to the new version and getting rid of the old version:

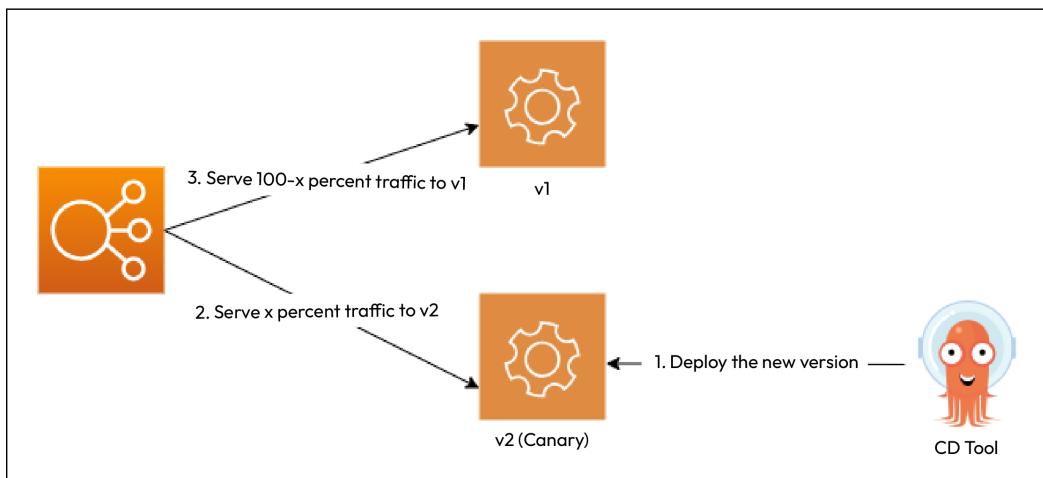


Figure 12.4 – Canary deployments

While complex deployments cause the least disruption to users, they are generally complex to manage using traditional CI tools such as Jenkins. Therefore, we need to get the tooling right on it. Several CD tools are available in the market, including **Argo CD**, **Spinnaker**, **Circle CI**, and **AWS Code Deploy**. As this entire book is focused on GitOps, and Argo CD is a GitOps native tool, for this chapter, we will focus on Argo CD. Before we delve into deploying the application, let's revisit what we want to deploy.

The Blog App and its deployment configuration

Since we discussed the Blog App in the last chapter, let's look at the services and their interactions again:

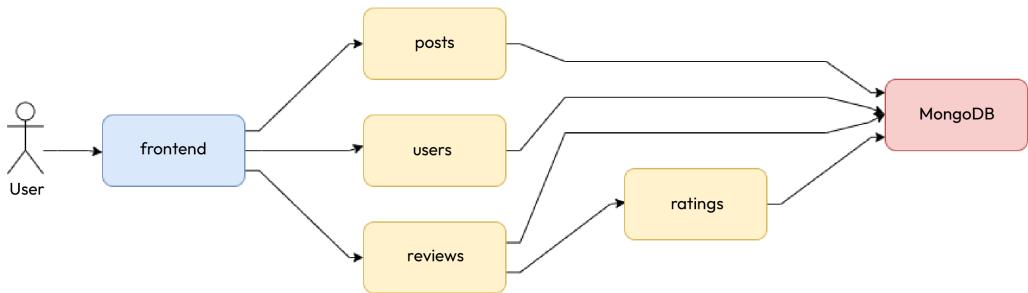


Figure 12.5 – The Blog App and its services and interactions

So far, we've created CI pipelines for building, testing, and pushing our Blog App microservice containers. These microservices need to run somewhere. So, we need an environment for this. We will deploy the application in a **GKE** cluster; for that, we will need a Kubernetes YAML manifest. We built the container for the `posts` microservice as an example in the previous chapter, and I also left building the rest of the services as an exercise for you. Assuming you've built them, we will need the following resources for the application to run seamlessly:

- **MongoDB:** We will deploy an auth-enabled MongoDB database with root credentials. The credentials will be injected via environment variables sourced from a Kubernetes **Secret** resource. We also need to persist our database data, so for that, we need a **PersistentVolume** mounted to the container, which we will provision dynamically using a **PersistentVolumeClaim**. As the container is stateful, we will use a **StatefulSet** to manage it and, therefore, a headless **Service** to expose the database.
- **Posts, reviews, ratings, and users:** The posts, reviews, ratings, and users microservices will interact with MongoDB through the root credentials injected via environment variables sourced from the same **Secret** as MongoDB. We will deploy them using their respective **Deployment** resources and expose all of them via individual **ClusterIP Services**.
- **Frontend:** The *frontend* microservice does not need to interact with MongoDB, so there will be no interaction with the Secret resource. We will also deploy this service using a **Deployment** resource. As we want to expose the service on the internet, we will create a **LoadBalancer Service** for it.

We can summarize these aspects with the following diagram:

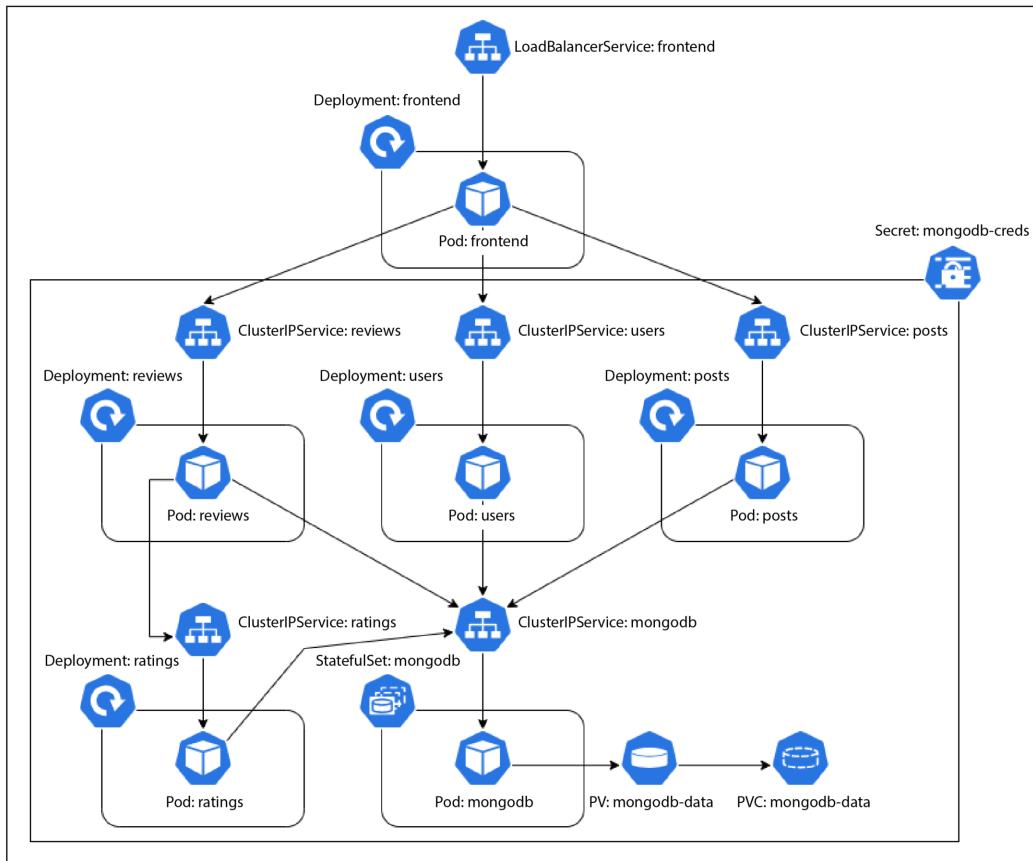


Figure 12.6 – The Blog App – Kubernetes resources and interactions

Now, as we're following the GitOps model, we need to store the manifests of all the resources on Git. However, since Kubernetes Secrets are not inherently secure, we cannot store their manifests directly on Git. Instead, we will use another resource called **SealedSecrets** to manage this securely.

In *Chapter 2, Source Code Management with Git and GitOps*, we discussed application and environment repositories forming the fundamental building blocks of GitOps-based CI and CD, respectively. In the previous chapter, we created an application repository on GitHub and used GitHub Actions (and Jenkins) to build, test, and push our application container to Docker Hub. As CD focuses on the Ops part of DevOps, we will need an **Environment repository** to implement it, so let's go ahead and create our Environment repository in the next section.

Continuous declarative IaC using an Environment repository

As we know by now, we must create a GKE cluster to host our microservices. So far, we've been using `gcloud` commands to do this; however, because `gcloud` commands are not declarative, using them is not ideal when implementing GitOps. Instead, we'll use **Terraform** to create the GKE cluster for us. This will ensure we can deploy and manage the cluster declaratively using a Git Environment repository. So, let's go ahead and create one.

Creating and setting up our Environment repository

Navigate to <https://github.com> and create a repository using a name of your choice. For this exercise, we will use `mdo-environments`. Once you have done that, navigate to Google Cloud Shell, generate a `ssh-key` pair using the `ssh-keygen` command, copy the public key to GitHub (refer to *Chapter 2, Source Code Management with Git and GitOps*, for step-by-step instructions), and clone the repository using the following commands:

```
$ cd ~  
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \  
modern-devops  
$ git clone git@github.com:<your_account>/mdo-environments.git  
$ cd mdo-environments
```

Let's copy a `.gitignore` file for Terraform to ensure that we do not unexpectedly check in Terraform state, backend, or `.tfvars` files by using the following command:

```
$ cp -r ~/modern-devops/ch12/.gitignore .
```

Now, let's push this code to GitHub using the following commands:

```
$ git add --all  
$ git commit -m 'Added gitignore'  
$ git push
```

Now that we've pushed our first file and initialized our repository, let's structure our repository according to our environments. We will have two branches within the Environment repository – **dev** and **prod**. All configurations in the **dev** branch will apply to the **development environment**, and those on **prod** will apply to the **production environment**. The following diagram illustrates this approach in detail:

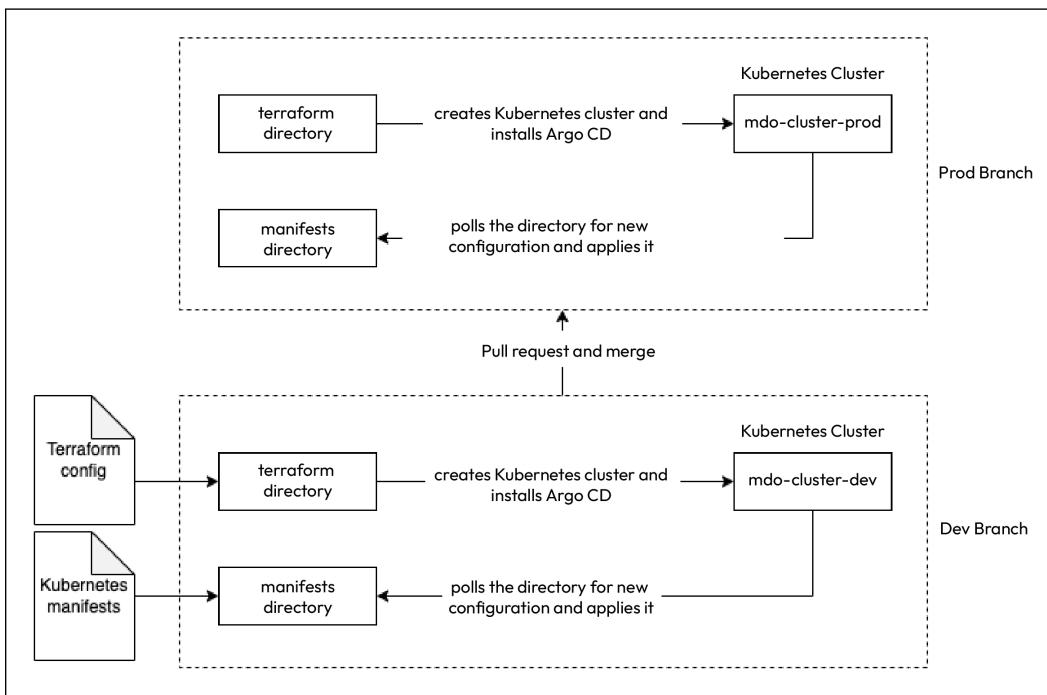


Figure 12.7 – CD process

The existing repository has a single branch called `master`. However, since we will be managing multiple environments in this repository, it would be good to rename the `master` branch to `prod`.

Go to https://github.com/<your_user>/mdo-environments/branches and click the pencil icon beside `master`. Type in `prod` and click on **Rename Branch**.

Now that we've renamed the branch, let's remove the existing local repository and clone the repository again using the following commands:

```
$ cd ~ && rm -rf mdo-environments
$ git clone git@github.com:<your_account>/mdo-environments.git
$ cd mdo-environments
```

We want to start with the dev environment, so it will be good to create a branch called `dev` from the `prod` branch. Run the following command to do so:

```
$ git branch dev && git checkout dev
```

Now, we can start writing the Terraform configuration within this directory. The configuration is available in `~/modern-devops/ch12/mando-environments/environments`. Copy everything from that directory to the current directory using the following commands:

```
$ cp -r ~/modern-devops/ch12/environments/terraform .
$ cp -r ~/modern-devops/ch12/environments/.github .
```

Within the `terraform` directory, there are several Terraform files.

The `cluster.tf` file contains the configuration to create the Kubernetes cluster. It looks like this:

```
resource "google_service_account" "main" {
  account_id    = "gke-${var.cluster_name}-${var.branch}-sa"
  display_name  = "GKE Cluster ${var.cluster_name}-${var.branch} Service Account"
}
resource "google_container_cluster" "main" {
  name          = "${var.cluster_name}-${var.branch}"
  location      = var.location
  initial_node_count = 3
  node_config {
    service_account = google_service_account.main.email
    oauth_scopes = [
      "https://www.googleapis.com/auth/cloud-platform"
    ]
  }
  timeouts {
    create = "30m"
    update = "40m"
  }
}
```

It creates two resources – a **service account** and a three-node **GKE instance** that uses the service account with the **cloud platform OAuth scope**.

We name the service account with a combination of the `cluster_name` and `branch` variables. This is necessary as we need to distinguish clusters between environments. So, if the cluster name is `mando-cluster` and the Git branch is `dev`, we will have a service account called `gke-mdo-cluster-dev-sa`. We will use the same naming convention on the GKE cluster. Therefore, the cluster's name would be `mando-cluster-dev`.

We have a `provider.tf` file that contains the provider and backend configuration. We're using a remote backend here as we want to persist the Terraform state remotely. In this scenario, we will use a **Google Cloud Storage (GCS) bucket**. The `provider.tf` file looks like this:

```
provider "google" {
  project      = var.project_id
  region       = "us-central1"
  zone         = "us-central1-c"
}
terraform {
  backend "gcs" {
```

```
    prefix  = "mdo-terraform"
}
}
```

Here, we've specified our default `region` and `zone` within the provider config. Additionally, we've declared the `gcs` backend, which only contains the `prefix` attribute with a value of `mdo-terraform`. We can separate configurations using the prefixes to store multiple Terraform states in a single bucket. We have purposefully not supplied the bucket name, which we will do at runtime using `-backend-config` during `terraform init`. The bucket name will be `tf-state-md-terraform-<PROJECT_ID>`.

Tip

As GCS buckets should have a globally unique name, it is good to use something such as `tf-state-md-terraform-<PROJECT_ID>` as the project ID is globally unique.

We also have the `variables.tf` file, which declares the `project_id`, `branch`, `cluster_name`, and `location` variables, as follows:

```
variable project_id {}
variable branch {...}
  default      = "dev"
}
variable cluster_name {...}
  default      = "mdo-cluster"
}
variable "location" {...}
  default      = "us-central1-a"
}
```

Now that we have the Terraform configuration ready, we need a workflow file that can be applied to our GCP project. For that, we've created the following GitHub Actions workflow file – that is, `.github/workflows/create-cluster.yml`:

```
name: Create Kubernetes Cluster
on: push
jobs:
  deploy-terraform:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./terraform
    steps:
      - uses: actions/checkout@v2
      - name: Install Terraform
        id: install-terraform
        run: wget -O terraform.zip https://releases.hashicorp.com/terraform/1.5.5/terraform_1.5.5_linux_amd64.zip && unzip terraform.zip && chmod +x terraform && sudo mv terraform /usr/local/bin
      - name: Apply Terraform
```

```

id: apply-terraform
  run: terraform init -backend-config="bucket=tf-state-mdo-terraform-${{ secrets.
PROJECT_ID }}"
  && terraform workspace select ${GITHUB_REF##*/} || terraform workspace new
${GITHUB_REF##*/} && terraform apply -auto-approve -var="project_id=${secrets.PROJECT_
ID }" -var=branch=${GITHUB_REF##*/}"
  env:
    GOOGLE_CREDENTIALS: ${secrets.GCP_CREDENTIALS }

```

This is a two-step build file. The first step installs Terraform, while the second step applies the Terraform configuration. Apart from that, we've specified `./terraform` as the working directory at the global level. Additionally, we're using a few secrets in this file, namely `GCP_CREDENTIALS`, which is the key file of the service account that Terraform uses to authenticate and authorize the GCP API, and the Google Cloud `PROJECT_ID`.

We've also supplied the bucket name as `tf-state-mdo-terraform-${{ secrets.
PROJECT_ID }}` to ensure that we have a unique bucket name.

As we've used Terraform workspaces to manage multiple environments, the preceding code selects an existing Terraform workspace with the branch name denoted by `${GITHUB_REF##*/}` or creates a new one. Workspaces are important here as we want to use the same configuration with different variable values for different environments. The Terraform workspaces correspond to environments, and environments correspond to the Git branch. So, as we have the `dev` and `prod` environments, we have the corresponding Terraform workspaces and Git branches.

From the Terraform and workflow configuration, we can deduce that we will need the following:

- A **service account** for Terraform to authenticate and authorize the GCP API and a JSON key file that we need to add as a GitHub secret
- The **project ID** that we'll configure as a GitHub secret
- A **GCS bucket** that we'll use as a backend for Terraform

So, let's go ahead and create a service account within GCP so that Terraform can use it to authenticate and authorize with the Google APIs. Use the following commands to create the service account, provide relevant **Identity and Access Management (IAM)** permissions, and download the credentials file:

```

$ PROJECT_ID=<project_id>
$ gcloud iam service-accounts create terraform \
--description="Service Account for terraform" \
--display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com

```

You will see a file called `key-file` within your working directory. Now, navigate to `https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new` and create a secret named `GCP_CREDENTIALS`. For the value, print the `key-file` file, copy its contents, and paste it into the **values** field of the GitHub secret.

Next, create another secret, `PROJECT_ID`, and specify your GCP project ID within the **values** field.

The next thing we need to do is create a GCS bucket for Terraform to use as a remote backend. To do this, run the following command:

```
$ gsutil mb gs://tf-state-mdt-terraform-${PROJECT_ID}
```

Additionally, we need to enable the GCP APIs that Terraform will use to create the resources. To do this, run the following command:

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

So, now that all the prerequisites have been met, we can push our code to the repository. Run the following commands to do this:

```
$ git add --all  
$ git commit -m 'Initial commit'  
$ git push --set-upstream origin dev
```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. Soon, the workflow will apply the configuration and create the Kubernetes cluster. This should appear as follows:

```
deploy-terraform
Started 1m 33s ago
Search logs
⚙️

> ✓ Set up job 2s
> ✓ Run actions/checkout@v2 1s
> ✓ Install Terraform 0s
▼ ⚡ Apply Terraform 1m 29s
  38 Created and switched to workspace "dev"!
  39 You're now on a new, empty workspace. Workspaces isolate their state,
  40 so if you run "terraform plan" Terraform will not see any existing state
  41 for this configuration.
  42 Terraform used the selected providers to generate the following execution
  43 plan. Resource actions are indicated with the following symbols:
  44   + create
  45 Terraform will perform the following actions:
  46     # google_container_cluster.main will be created
  47     + resource "google_container_cluster" "main" ***
  48       + cluster_ipv4_cidr      = (known after apply)
  49       + datapath_provider      = (known after apply)
  50       + default_max_pods_per_node = (known after apply)
```

Figure 12.8 – GitOps with GitHub Actions and Terraform

To verify whether the cluster has been created successfully, run the following command:

```
$ gcloud container clusters list
NAME: mdo-cluster-dev
LOCATION: us-central1-a
MASTER_VERSION: 1.27.3-gke.100
MASTER_IP: x.x.x.x
MACHINE_TYPE: e2-medium
NODE_VERSION: 1.27.3-gke.100
NUM_NODES: 3
STATUS: RUNNING
```

As you can see, the `mdo-cluster-dev` cluster is running successfully in the environment. If we make any changes to the Terraform configuration, the changes will automatically be applied. We've successfully created our Environment using an Environment repository. That is the *push model GitOps* in action for you. Now, we need to run our application in the environment; to manage and deploy the application, we will need a dedicated CD tool. As stated previously, we will use Argo CD for this, so let's look at it.

Introduction to Argo CD

Argo CD is an open source, declarative, GitOps-based CD tool designed to automate deploying and managing applications and infrastructure on Kubernetes clusters. Argo CD serves as a robust application controller, efficiently managing and ensuring the smooth and secure operation of your applications. Argo CD works in the *pull-based GitOps model* and, therefore, polls the Environment repository regularly to detect any configuration drift. Suppose it finds any drift between the state in Git and the actual state of applications running in the environment. In that case, it will make corrective changes to reflect the desired configuration declared in the Git repository.

Argo CD is tailored explicitly to Kubernetes environments, making it a popular choice for managing applications on Kubernetes clusters.

In addition to the traditional Kubernetes manifest YAML files, Argo CD offers support for various alternative methods of defining Kubernetes configurations:

- Helm charts
- Kustomize
- Ksonnet
- Jsonnet files
- Plain YAML/JSON manifest files
- Integration with other customized configuration management tools through plugins

Within Argo CD, you can define applications encompassing both a *source* and a *target*. The source specifies details about the associated Git repository, the location of the manifests, helm charts, or kustomize files, and then applies these configurations to designated target environments. This empowers you to monitor changes within a specific branch, tag, or watch particular versions within your Git repository. Diverse tracking strategies are also at your disposal.

You can access a user-friendly web-based UI and a **command-line interface (CLI)** to interact with Argo CD. Moreover, Argo CD facilitates reporting on the application's status via sync hooks and app actions. If any modifications are made directly within the cluster that deviate from the GitOps approach, Argo CD can promptly notify your team, perhaps through a Slack channel.

The following diagram provides an overview of the Argo CD architecture:

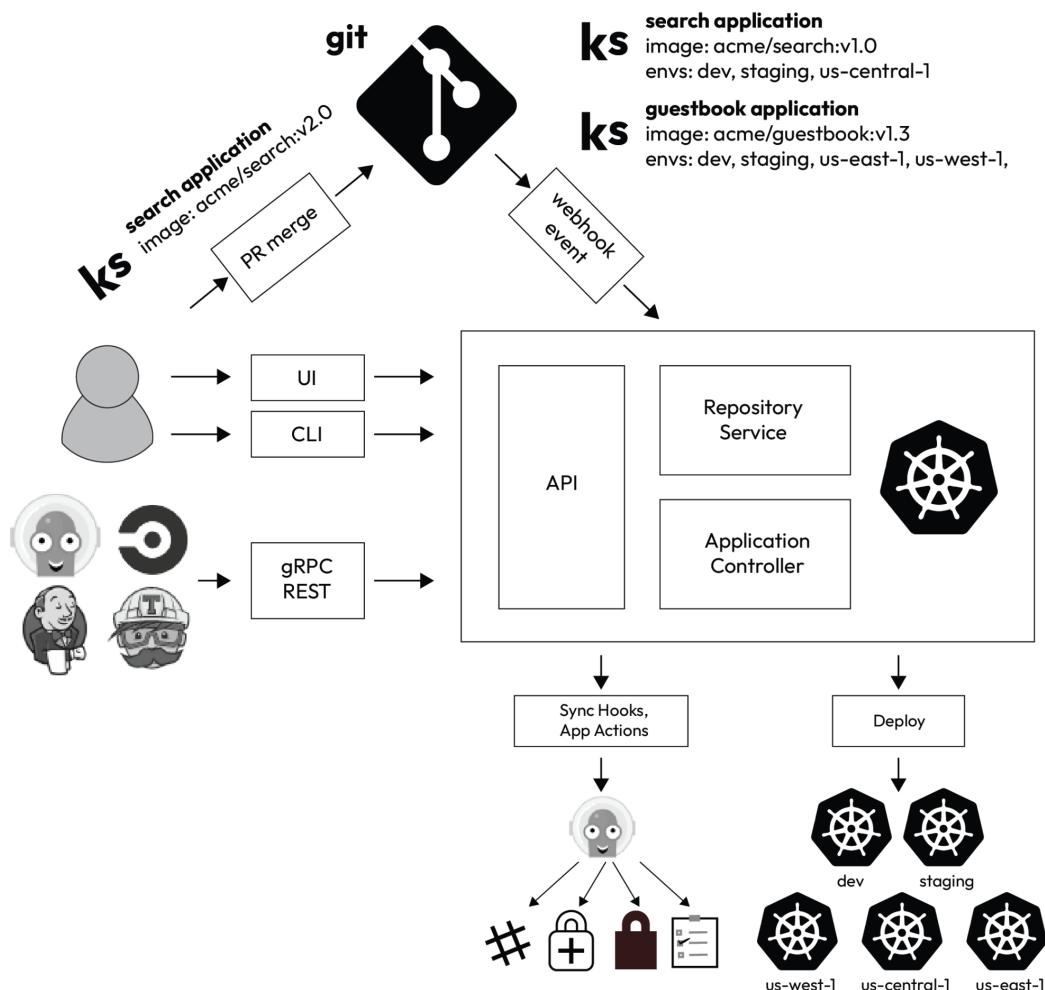


Figure 12.9 – Argo CD architecture

So, without further ado, let's spin up Argo CD.

Installing and setting up Argo CD

Installing Argo CD is simple – we need to apply the `install.yaml` manifest bundle that's available online at <https://github.com/argoproj/argo-cd/blob/master/manifests/install.yaml> on the Kubernetes cluster where we wish to install it. For a more customized installation, you can refer to <https://argo-cd.readthedocs.io/en/stable/operator-manual/installation/>.

As we're using GitOps for this chapter, we will not deploy Argo CD manually. Instead, we will use Terraform to set it up using the Environment repository.

The resources for this section are present in `~/modern-devops/ch12/environments-argocd-app`. We will use the same Environment repository as before for managing this environment.

Therefore, let's `cd` into the `mdo-environments` local repository and run the following commands:

```
$ cd ~/mdo-environments
$ cp -r ~/modern-devops/ch12/environments-argocd-app/terraform .
$ cp -r ~/modern-devops/ch12/environments-argocd-app/manifests .
$ cp -r ~/modern-devops/ch12/environments-argocd-app/.github .
```

Now, let's look at the directory structure to understand what we're doing:

```
.
├── .github
│   └── workflows
│       └── create-cluster.yml
└── manifests
    └── argocd
        ├── apps.yaml
        ├── install.yaml
        └── namespace.yaml
└── terraform
    ├── app.tf
    ├── argocd.tf
    ├── cluster.tf
    ├── provider.tf
    └── variables.tf
```

As we can see, the structure is similar to before, except for a few changes. Let's look at the Terraform configuration first.

Terraform changes

The `terraform` directory now contains two more files:

- `argocd.tf`: This contains the Terraform configuration for deploying Argo CD
- `app.tf`: This contains the Terraform configuration for configuring Argo CD apps

Let's explore both files in detail.

`argocd.tf`

This file starts with the `time_sleep` resource with an explicit dependency on the `google_container_cluster` resource. It will sleep for 30 seconds after the cluster is created so that it is ready to serve requests:

```
resource "time_sleep" "wait_30_seconds" {
  depends_on = [google_container_cluster.main]
  create_duration = "30s"
}
```

To connect with GKE, we will use the `gke_auth` module provided by `terraform-google-modules/kubernetes-engine/google//modules/auth`. We will add an explicit dependency to the `time_sleep` module so that authentication happens 30 seconds after the cluster is created:

```
module "gke_auth" {
  depends_on      = [time_sleep.wait_30_seconds]
  source          = "terraform-google-modules/kubernetes-engine/google//modules/auth"
  project_id     = var.project_id
  cluster_name    = google_container_cluster.main.name
  location        = var.location
  use_private_endpoint = false
}
```

Now that we've authenticated with the GKE cluster, we need to apply manifests to deploy Argo CD to the cluster. We will use the `gavinbunney/kubectl` plugin (<https://registry.terraform.io/providers/gavinbunney/kubectl/latest/docs>) for that.

We start by defining some data sources to help generate Kubernetes manifests that we will apply to install Argo CD. We will create two `kubectl_file_documents` data sources for the namespace and Argo CD app that point to the corresponding `namespace.yaml` and `install.yaml` files within the `manifests/argocd` directory:

```
data "kubectl_file_documents" "namespace" {
  content = file("../manifests/argocd/namespace.yaml")
}
data "kubectl_file_documents" "argocd" {
  content = file("../manifests/argocd/install.yaml")
}
```

Using these data sources, we can create two `kubectl_manifest` resources for the namespace and Argo CD app. These resources will apply the manifests within the GKE cluster:

```
resource "kubectl_manifest" "namespace" {
  for_each = data.kubectl_file_documents.namespace.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
resource "kubectl_manifest" "argocd" {
  depends_on = [
    kubectl_manifest.namespace,
  ]
  for_each = data.kubectl_file_documents.argocd.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

Now that we've added the configuration to install Argo CD, we also need to configure argo CD Applications. To do that, we have the `app.tf` file.

app.tf

Similar to the Argo CD configuration, we have a `kubectl_file_documents` data source reading from the `manifests/argocd/apps.yaml` file; the `kubectl_manifest` resource will apply the manifest to the Kubernetes cluster:

```
data "kubectl_file_documents" "apps" {
  content = file("../manifests/argocd/apps.yaml")
}
resource "kubectl_manifest" "apps" {
  depends_on = [
    kubectl_manifest.argocd,
  ]
  for_each = data.kubectl_file_documents.apps.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}
```

We've also modified the `provider.tf` file, so we'll explore that next.

provider.tf

Within this file, we have included the `kubectl` provider, as follows:

```
...
provider "kubectl" {
  host           = module.gke_auth.host
  cluster_ca_certificate = module.gke_auth.cluster_ca_certificate
  token          = module.gke_auth.token
  load_config_file = false
}
```

```
}
```

```
terraform {
```

```
    required_providers {
```

```
        kubectl = {
```

```
            source  = "gavinbunney/kubectl"
            version = ">= 1.7.0"
        }
    }...
}
```

Now, let's inspect the manifests directory.

The Kubernetes manifests

The manifests directory contains Kubernetes manifests that we will apply to the Kubernetes cluster. As we're setting up Argo CD first, it only contains the `argocd` directory at the moment; however, we will extend this to add further directories later in this chapter.

The `manifests/argocd` directory contains the following files:

- `namespace.yaml`: The manifest to create the `argocd` namespace where Argo CD will run.
- `install.yaml`: The manifest to create the Argo CD application. The manifest is downloaded from the official Argo CD release URL.
- `apps.yaml`: This contains an Argo CD **ApplicationSet** configuration.

While the `namespace.yaml` and `install.yaml` files are self-explanatory, let's discuss the `apps.yaml` file and the Argo CD ApplicationSet resource in more detail.

Argo CD Application and ApplicationSet

To manage applications declaratively, Argo CD uses the **Application** resource. An Application resource defines the configuration required for Argo CD to access Kubernetes deployment configuration stored in the Git repository using the `source` attribute and where it needs to apply them using the `target` attribute. An Application resource caters to one application. For example, to deploy our Blog App, we will need to create an Application resource like the following:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: blog-app
  namespace: argocd
spec:
  project: default
```

```

source:
  repoURL: https://github.com/<your_github_repo>/mdo-environments.git
  targetRevision: HEAD
  path: manifests/nginx
destination:
  server: https://kubernetes.default.svc
syncPolicy:
  automated:
    selfHeal: true

```

This manifest defines an Argo CD Application resource with the following sections:

- **project:** We can organize applications into distinct projects. In this case, we will stick to the default project.
- **source:** This section defines the configuration Argo CD requires to track and pull the application configuration from the Git repository. It typically contains `repoURL`, `targetRevision`, and the `path` value where the application manifests are located.
- **destination:** This section defines the `target` value to which we want to apply the manifest. It typically contains the `server` section and contains the Kubernetes cluster's URL.
- **syncPolicy:** This section defines any policies Argo CD should apply while syncing the Blog App from the Git repository and what to do when it detects a drift. In the preceding configuration, it would try to correct any drift from the Git repository automatically as `selfHeal` is set to `true`.

We can very well go ahead and define multiple application manifests for each application. However, for larger projects, it might turn out to be an overhead. To manage this, Argo CD provides a generic way of creating and managing applications via the `ApplicationSet` resource.

The `ApplicationSet` resource provides us with a way to dynamically generate Application resources by using a defined pattern. In our case, we have the following structure:

```

manifests
└── argocd
  └── apps.yaml
  └── install.yaml
  └── namespace.yaml
└── blog-app
  └── manifest.yaml
└── <other-app>
  └── manifest.yaml

```

So, logically, for every subdirectory of the `manifests` directory, we would need to create a new application with the directory name. The respective application configuration should source all manifests from the subdirectory.

We've defined the following `ApplicationSet` within the `apps.yaml` file:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: argo-apps
  namespace: argocd
spec:
  generators:
    - git:
        repoURL: https://github.com/<your_github_repo>/mdo-environments.git
        revision: HEAD
        directories:
          - path: manifests/*
            - path: manifests/argocd
              exclude: true
  template:
    metadata:
      name: '{{path.basename}}'
    spec:
      project: default
      source:
        repoURL: https://github.com/<your_github_repo>/mdo-environments.git
        targetRevision: HEAD
        path: '{{path}}'
      destination:
        server: https://kubernetes.default.svc
      syncPolicy:
        automated:
          selfHeal: true
```

`ApplicationSet` has the following sections:

- `generators`: This section defines how Argo CD should generate Application resources. We've used the `git` generator, which contains the `repoURL`, `revision`, and `directories` sections. The `directories` section defines the directory from where we would want to source our applications. We've set that to `manifests/*`. So, it will look for every subdirectory within the `manifests` directory. We have also defined an exclude directory called `manifests/argocd`. This is because we don't want Argo CD to manage the configuration to deploy itself.

- **templates:** This section defines the template for creating the application. As we can see, the contents are very similar to an Application resource definition. For `metadata.name`, we specified `{path.basename}`, which means it will create Application resources with the subdirectory name as we intended. The `template.spec.source.path` attribute contains the source path of the corresponding application manifests, so we've set that to `{path}` – that is, the subdirectory. So, we will have `blog-app` and `<other-app>` applications based on the preceding directory structure. The rest of the attributes are the same as those for the Application resource we discussed previously.

Now that we've configured everything we need to install and set up Argo CD, let's commit and push this configuration to the remote repository by using the following commands:

```
$ git add --all  
$ git commit -m "Added argocd configuration"  
$ git push
```

We will see that GitHub will run the Actions workflow on update and deploy Argo CD. Once the workflow is successful, we can go ahead and access the Argo CD Web UI.

Accessing the Argo CD Web UI

Before we can access the Argo CD Web UI, we must authenticate with the GKE cluster. To do so, run the following command:

```
$ gcloud container clusters get-credentials \  
  mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID
```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```
$ kubectl get svc argocd-server -n argocd  
NAME          TYPE      EXTERNAL-IP    PORTS          AGE  
argocd-server LoadBalancer 34.122.51.25  80/TCP,443/TCP 6m15s
```

We now know that Argo CD is accessible at `https://34.122.51.25/`. Upon visiting this link, you'll notice that username and password are required for authentication:

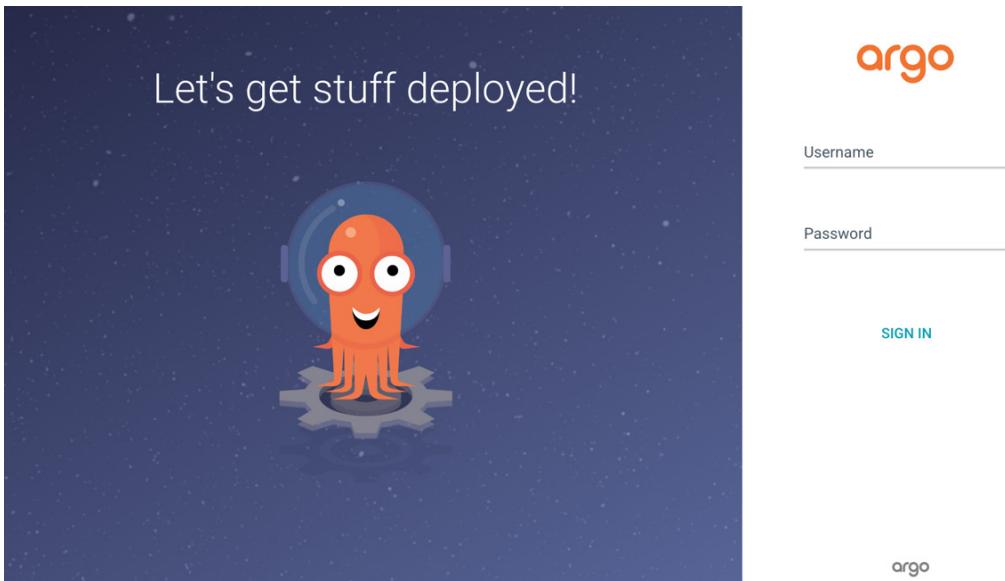


Figure 12.10 – Argo CD Web UI – login page

Argo CD provides an initial `admin` user by default, and the password for this user is stored in the `argocd-initial-admin-secret` **Secret** resource as plaintext. While you can use this default setup, it's worth noting that it is generated from the publicly available YAML manifest. Therefore, it's advisable to update it. To do so, execute the following command:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow two minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
-o jsonpath='{.data.password}' | base64 -d && echo
```

Now that you have the necessary credentials, log in and you will see the following page:

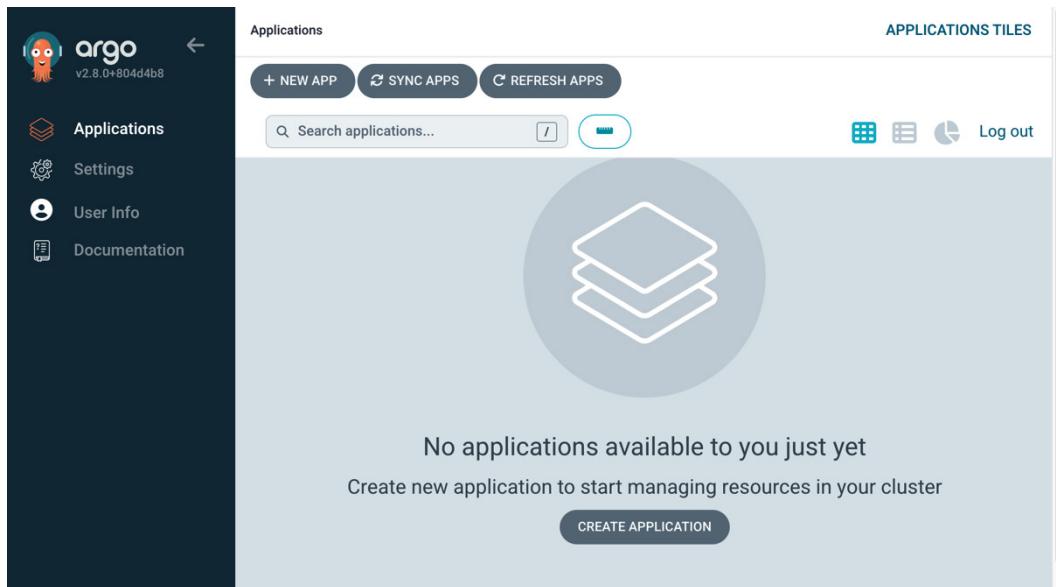


Figure 12.11 – Argo CD Web UI – home page

We've successfully set up Argo CD. The next step is to deploy our application; however, as we know that our application uses Kubernetes Secrets, which we cannot store on Git, we will have to find a mechanism to store it securely. To solve that problem, we have Bitnami's **SealedSecret** resource. We'll look at that in the next section.

Managing sensitive configurations and Secrets

Sealed Secrets solves the problem of *I can manage all my Kubernetes config in Git, except Secrets*. Sealed Secrets function as secure containers for your sensitive information. When you require a storage solution for secrets, such as passwords or keys, you place them in these specialized packages. Only the Sealed Secrets controller within Kubernetes can unlock and access the contents. This ensures the utmost security and protection for your valuable secrets. Created by *Bitnami Labs* and open sourced, they help you encrypt your Kubernetes Secrets into Sealed Secrets using asymmetric cryptography that only the Sealed Secrets controller running on the cluster can decrypt. This means you can store the Sealed Secrets in Git and use GitOps to set up everything, including Secrets.

Sealed Secrets comprises two components:

- A client-side utility called `kubeseal` helps us generate Sealed Secrets from standard Kubernetes Secret YAML

- A cluster-side Kubernetes controller/operator unseals your secrets and provides the key certificate to the client-side utility

The typical workflow when using Sealed Secrets is illustrated in the following diagram:

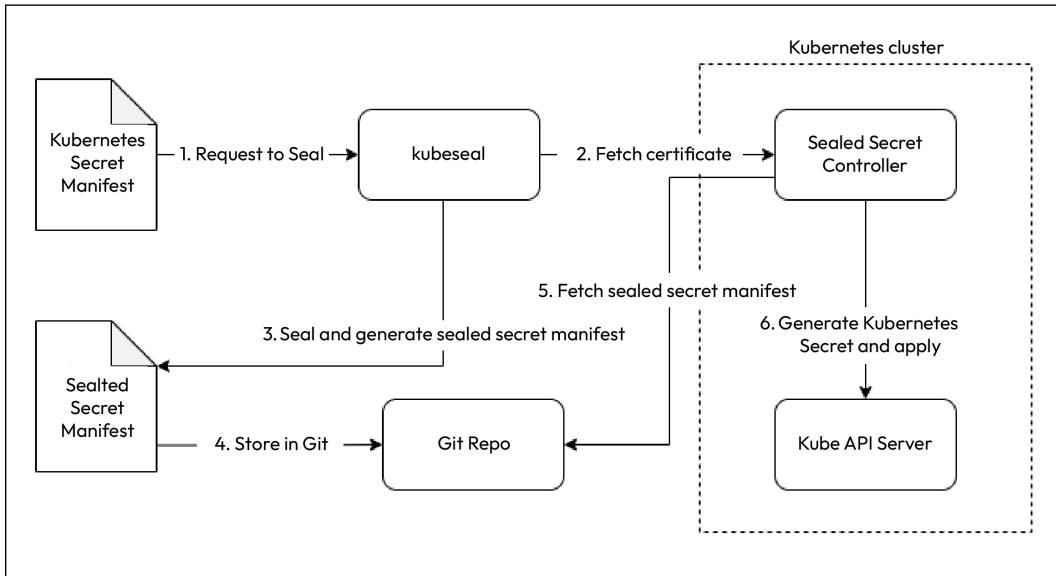


Figure 12.12 – Sealed Secrets workflow

Now, let's go ahead and install the Sealed Secrets operator.

Installing the Sealed Secrets operator

To install the **Sealed Secrets operator**, all you need to do is download the controller manifest from the latest release at <https://github.com/bitnami-labs/sealed-secrets/releases>. At the time of writing this book, <https://github.com/bitnami-labs/sealed-secrets/releases/download/v0.23.1/controller.yaml> is the latest controller manifest.

Create a new directory called `sealed-secrets` within the `manifest` directory and download `controller.yaml` using the following commands:

```
$ cd ~/mdo-environments/manifests & mkdir sealed-secrets  
$ cd sealed-secrets  
$ wget https://github.com/bitnami-labs/sealed-secrets\releases/download/v0.23.1/controller.yaml
```

Then, commit and push the changes to the remote repository. After about five minutes, Argo CD will create a new application called **sealed-secrets** and deploy it. You can visualize this in the Argo CD Web UI as follows:

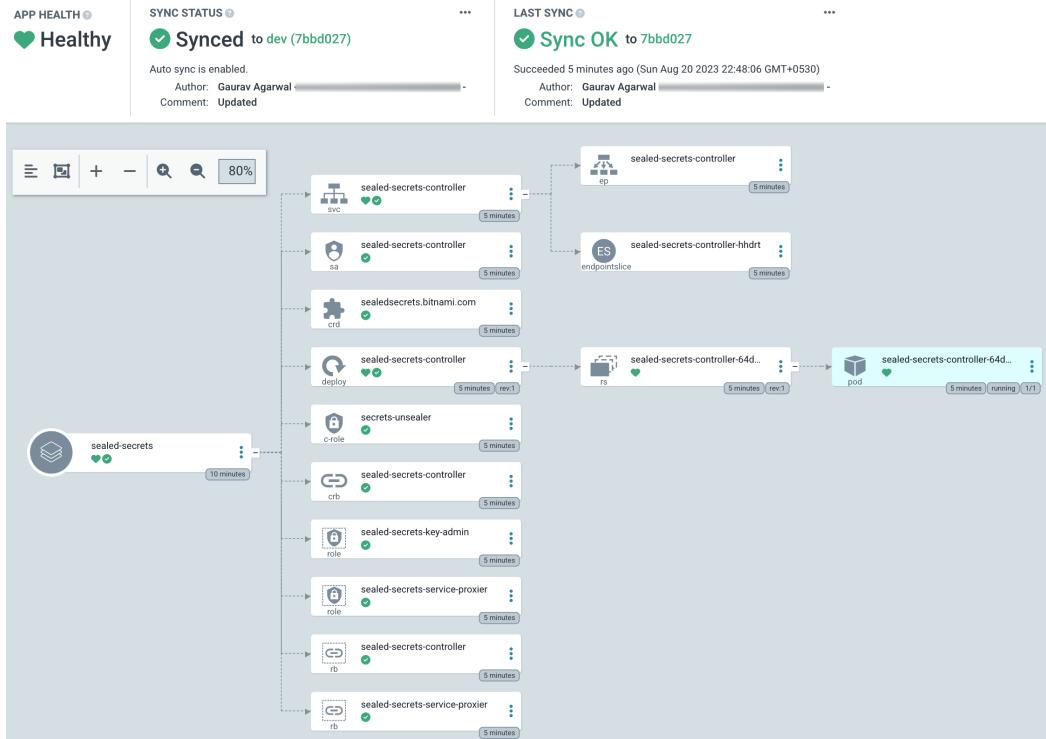


Figure 12.13 – Argo CD Web UI – Sealed Secrets

In the Kubernetes cluster, `sealed-secrets-controller` will be visible in the `kube-system` namespace. Run the following command to check this:

```
$ kubectl get deployment -n kube-system sealed-secrets-controller
NAME                      READY UP-TO-DATE AVAILABLE AGE
sealed-secrets-controller 1/1    1          1        6m4s
```

As we can see, the controller is running and ready. We can now install the client-side utility – `kubeseal`.

Installing kubeseal

To install the client-side utility, you can go to <https://github.com/bitnami-labs/sealed-secrets/releases> and get the kubeseal installation binary link from that page. The following commands will install kubeseal 0.23.1 on your system:

```
$ KUBESEAL_VERSION='0.23.1'  
$ wget "https://github.com/bitnami-labs/sealed-secrets/releases/download\\  
/v${KUBESEAL_VERSION:?}/kubeseal-${KUBESEAL_VERSION:?}-linux-amd64.tar.gz"  
$ tar -xvf kubeseal-${KUBESEAL_VERSION:?}-linux-amd64.tar.gz kubeseal  
$ sudo install -m 755 kubeseal /usr/local/bin/kubeseal  
$ rm -rf ./kubeseal*
```

To check whether kubeseal has been installed successfully, run the following command:

```
$ kubeseal --version  
kubeseal version: 0.23.1
```

Since kubeseal has been installed, let's go ahead and create a Sealed Secret for the blog-app.

Creating Sealed Secrets

To create a Sealed Secret, we have to define the Kubernetes Secret resource. The mongodb-creds Secret should contain some key-value pairs with the MONGO_INITDB_ROOT_USERNAME key with a value of root and the MONGO_INITDB_ROOT_PASSWORD key with any value you want as the password.

As we don't want to store the plaintext Secret as a file, we will first create the Kubernetes secret manifest called mongodb-creds using the --dry-run and -o yaml flags and then pipe the output directly to kubeseal to generate the SealedSecret resource using the following command:

```
$ kubectl create secret generic mongodb-creds \  
--dry-run=client -o yaml --namespace=blog-app \  
--from-literal=MONGO_INITDB_ROOT_USERNAME=root \  
--from-literal=MONGO_INITDB_ROOT_PASSWORD=<your_pwd> \  
| kubeseal -o yaml > mongodb-creds-sealed.yaml
```

This generates the mongodb-creds-sealed.yaml Sealed Secret, which looks like this:

```
apiVersion: bitnami.com/v1alpha1  
kind: SealedSecret  
metadata:  
  name: mongodb-creds
```

```

namespace: blog-app
spec:
  encryptedData:
    MONGO_INITDB_ROOT_PASSWORD: AgB+tySkf72M/...
    MONGO_INITDB_ROOT_USERNAME: AgA95xKJg8veOy8v/...
  template:
    metadata:
      name: mongodb-creds
      namespace: blog-app

```

As you can see, the Sealed Secret is very similar to the Secret manifest. Still, instead of containing a Base64-encoded secret value, it has encrypted it so that only the Sealed Secrets controller can decrypt it. You can easily check this file into source control. Let's go ahead and do that. Move the Sealed Secret YAML file to the `manifests/blog-app` directory using the following command:

```

$ mkdir -p ~/mdo-environments/manifests/blog-app/
$ mv mongodb-creds-sealed.yaml ~/mdo-environments/manifests/blog-app/

```

Now that we've successfully generated the Sealed Secret and moved it to the `manifests/blog-app` directory, we'll set up the rest of our application in the next section.

Deploying the sample Blog App

To deploy the sample Blog App, we need to define application resources. We've already discussed what our app is composed of. We have defined the application bundle as a Kubernetes manifest file called `blog-app.yaml`. We need to copy this YAML to the `manifests/blog-app` directory using the following command:

```

$ cp ~/modern-devops/ch12/blog-app/blog-app.yaml \
~/mdo-environments/manifests/blog-app/

```

I've prebuilt the microservices and used the required `git-sha` as the tag, as we did in the previous chapter. You can edit the YAML and replace it with your image for each application.

Once done, commit and push the changes to the `mdo-environments` repository.

As soon as you push the changes, you should notice that the `blog-app` application starts appearing in the Argo CD UI in less than five minutes:

The screenshot shows the Argo CD Web UI Applications page. At the top, there are buttons for '+ NEW APP', 'SYNC APPS', 'REFRESH APPS', a search bar ('Search applications.'), and icons for dashboard, applications, and metrics, along with a 'Log out' button. Below the header, a navigation bar allows sorting by name and setting items per page to 10. Two application cards are displayed:

- blog-app**
 - Project: default
 - Labels: argocd.argoproj.io/application-set...
 - Status: ⚡ Progressing 🟢 Synced
 - Reposi...: https://github.com/bharatmicrosys...
 - Target ...: dev
 - Path: manifests/blog-app
 - Destin...: in-cluster
 - Names...:
 - Create...: 08/21/2023 17:24:33 (a few secon...
 - Last Sy...: 08/21/2023 17:24:37 (a few secon...

Actions: SYNC, C, X
- sealed-secrets**
 - Project: default
 - Labels: argocd.argoproj.io/application-set...
 - Status: ❤️ Healthy 🟢 Synced
 - Reposi...: https://github.com/bharatmicrosys...
 - Target ...: dev
 - Path: manifests/sealed-secrets
 - Destin...: in-cluster
 - Names...:
 - Create...: 08/21/2023 17:03:32 (21 minutes ...)
 - Last Sy...: 08/21/2023 17:03:36 (21 minutes ...)

Actions: SYNC, C, X

Figure 12.14 – Argo CD Web UI – Applications

Wait for the application to progress. Once it turns green, you should see the following within the application:

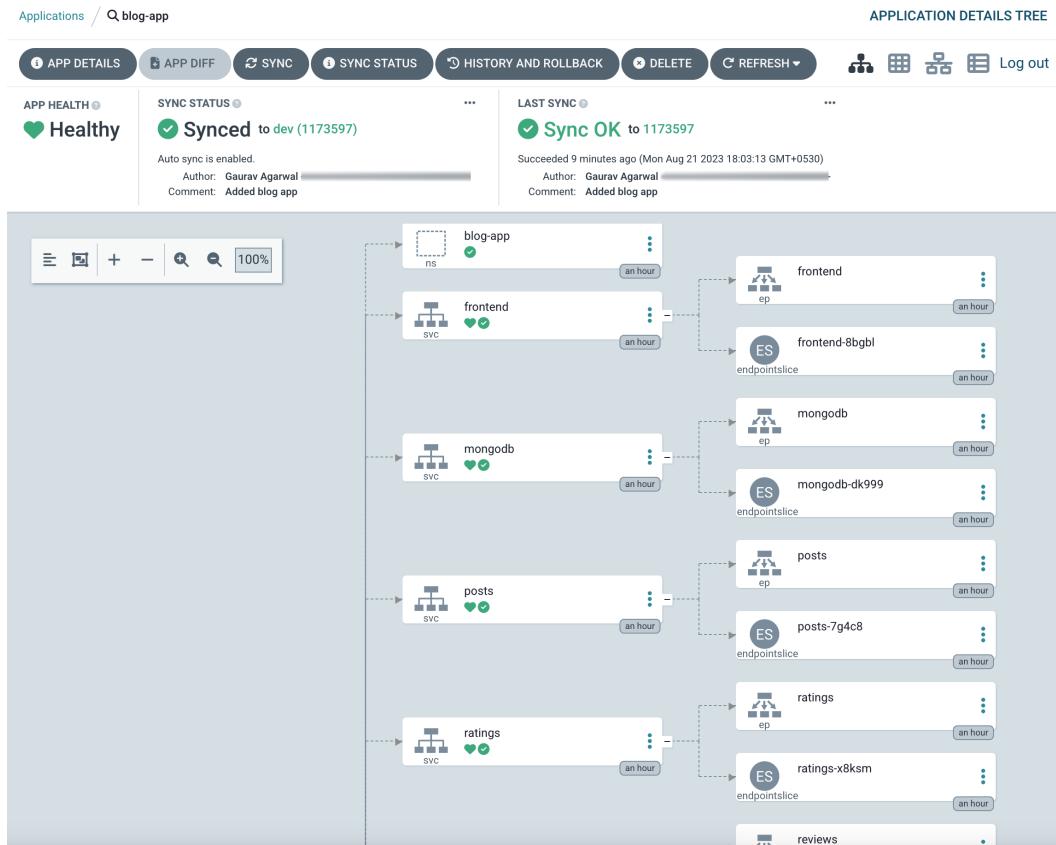


Figure 12.15 – Argo CD Web UI – blog-app

Now that the application is all synced up, we can check the resources that were created within the `blog-app` namespace. Let's list the Services first using the following command:

```
$ kubectl get svc -n blog-app
NAME      TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)
frontend  LoadBalancer  10.71.244.154  34.68.221.0  80:3203/TCP
mongodb   ClusterIP   None          <none>       27017/TCP
posts     ClusterIP   10.71.242.211  <none>       5000/TCP
ratings   ClusterIP   10.71.244.78   <none>       5000/TCP
reviews   ClusterIP   10.71.247.128  <none>       5000/TCP
users     ClusterIP   10.71.241.25   <none>       5000/TCP
```

As we can see, it lists all the Services that we've defined. Note that the `frontend` service is of the `LoadBalancer` type and has an **External IP**. Note down this External IP as we will use it to access our application.

Now, let's list the **pods** to see whether all the microservices are running fine:

```
$ kubectl get pod -n blog-app
NAME             READY   STATUS    RESTARTS
frontend-7cbdc4c6cd-4jzdw  1/1    Running   0
mongodb-0        1/1    Running   0
posts-588d8bcd99-sphpm  1/1    Running   0
ratings-7dc45697b-wwfqd  1/1    Running   0
reviews-68b7f9cb8f-2jgvv  1/1    Running   0
users-7cdd4cd94b-g67zw   1/1    Running   0
```

As we can see, all pods are running fine. Note that the `mongodb-0` pod contains a numeric prefix, but the rest of the pods have random UUIDs. You might recall that when we created a **StatefulSet**, the pods always maintained a unique ID and were created in order. At first glance, the application seems to be set up correctly. Let's list the Secrets as well to see whether the `mongodb-creds` secret has been created:

```
$ kubectl get secret -n blog-app
NAME          TYPE      DATA   AGE
mongodb-creds Opaque    2      80s
```

Here, we can see that the `mongodb-creds` Secret has been created. This shows us that SealedSecret is working fine.

Now, let's go ahead and access our application by opening `http://<frontend-svc-external-ip>`. If you see the following page, the application was deployed correctly:

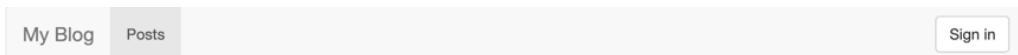


Figure 12.16 – Blog App home page

As an exercise, play around with the application by clicking on **Sign In > Not a user? Create an Account** and then fill in the details to register. You can *create* a new **Post**, add **Reviews**, and provide **Ratings**. You can also *update* your reviews, *delete* them, update ratings, and more. Try out the app to see whether all aspects are working correctly. You should be able to see something like the following:

The screenshot shows a web-based blog application. At the top, there's a navigation bar with tabs for "My Blog", "Posts", and "Actions". A user profile icon for "Gaurav" is also present. Below the navigation, the first post is titled "My first post" and is attributed to "Author: gaurav.agarwal@example.com". The post content is "This is my first post! I hope you like it!". Underneath the post, there's a section titled "Reviews:" with two entries. Each entry includes a "Edit" link and a trash can icon. The first review is "This is a great review!" with a five-star rating and the author "gaurav.agarwal@example.com". The second review is "This is awesome! I love it!" with a five-star rating and the same author. At the bottom, there's a section titled "Add a Review:" with a "Review" input field containing "Enter your review", a star rating section with a "Five Stars" button, and a "Submit" button.

Figure 12.17 – Blog App posts

As we're happy with the application, we can raise a pull request from the `dev` branch to the `prod` branch. Once you merge the pull request, you will see that similar services will emerge in the production environment. You can use pull request-based gating for CD as well. This ensures that your environments remain independent while being sourced from the same repository, albeit from different branches.

Summary

This chapter has covered continuous deployment and delivery, and we understood the need for CD and the basic CD workflow for a container application. We discussed several modern deployment strategies and how CI tools cannot fulfill those responsibilities. Using the GitOps principles, we created an Environment repository and deployed our GKE-based environment using GitHub Actions by employing the push-based model. Then, we looked at using Argo CD as our CD tool and installed it. To avoid committing sensitive information in Git, such as secrets, we discussed Bitnami's Sealed Secrets. We then deployed the sample Blog App using Argo CD, using GitOps all the while.

In the next chapter, we will explore another vital aspect of modern DevOps – securing the deployment pipeline.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following are CD tools? (Choose three)
 - A. Spinnaker
 - B. GitHub
 - C. Argo CD
 - D. AWS Code Deploy
2. CD requires human input for deployment to production. (True/False)
3. Argo CD supports Blue/Green deployments out of the box. (True/False)
4. What would you use to initiate deployment using Argo CD?
 - A. Trigger the pipeline manually
 - B. Check in changes to your Git repository
 - C. Use CI to trigger Argo CD pipelines
 - D. Argo CD pipelines don't react to external stimuli
5. An Argo CD ApplicationSet helps generate applications based on templates. (True/False)
6. What branch names should you prefer for your Environment repository?
 - A. dev, staging, and prod
 - B. feature, develop, and master
 - C. release and main
7. Which of the following deployment models does Argo CD use?
 - A. Push model
 - B. Pull model
 - C. Staggering model
8. You should use Terraform to install Argo CD as you can store all configurations in Git. (True/False)
9. Argo CD can sync resources from which of the following sources? (Choose two)
 - A. Git repository
 - B. Container Registry
 - C. JFrog Artifactory's raw repository

10. What would Argo CD do if you manually changed a resource outside Git?
 - A. Argo CD would change the resource so that it matches the Git configuration
 - B. Argo CD would notify you that a resource has changed outside Git
 - C. Argo CD would do nothing
11. You can check in Sealed Secrets to a Git repository. (True/False)

Answers

Here are the answers to this chapter's questions:

1. A, C, and D
2. True
3. True
4. B
5. True
6. A
7. B
8. True
9. A, B
10. A
11. True

13

Securing and Testing Your CI/CD Pipeline

In the previous chapters, we looked at **Continuous Integration (CI)** and **Continuous Deployment/Delivery (CD)** with GitOps as the central concept. Both concepts and the tooling surrounding them help us deliver better software faster. However, one of the most critical aspects of technology is security and quality assurance. Though security was not considered in DevOps' initial days, with the advent of **DevSecOps**, modern DevOps now places a great emphasis on it. In this chapter, we'll try to understand the concepts surrounding container applications' security and testing and how to apply them within CI and CD.

In this chapter, we're going to cover the following main topics:

- Securing and testing CI/CD pipelines
- Revisiting the Blog Application
- Container vulnerability scanning
- Managing secrets
- Binary authorization
- Release gating with pull requests and deploying our application in production
- Security and testing best practices for modern DevOps pipelines

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine (GKE)**, for the exercises. Currently, **Google Cloud Platform (GCP)** provides a free \$300 trial for 90 days, so you can go ahead and sign up for one at <https://console.cloud.google.com/>.

You will also need to clone the following GitHub repository for some of the exercises:

<https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

You can use the Cloud Shell offering available on GCP to follow this chapter. Go to Cloud Shell and start a new session. Run the following commands to clone the repository into your home directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
```

We also need to set the project ID and enable a few GCP APIs that we will use in this chapter. To do so, run the following commands:

```
$ PROJECT_ID=<YOUR_PROJECT_ID>
$ gcloud services enable iam.googleapis.com \
container.googleapis.com \
binaryauthorization.googleapis.com \
containeranalysis.googleapis.com \
secretmanager.googleapis.com \
cloudresourcemanager.googleapis.com \
cloudkms.googleapis.com
```

Now, in the next section, let's look at how to secure and test CI/CD pipelines.

Securing and testing CI/CD pipelines

With continuous cyber threats and the ongoing war between cybersecurity experts and cybercriminals, security has always been the top priority for most organizations, and it also forms a significant part of a mature organization's investment.

However, security comes with its costs. Most organizations have cybersecurity teams that audit their code regularly and give feedback. However, that process is generally slow and happens when most of the code is already developed and difficult to modify.

Similarly, while most organizations significantly emphasize automated testing, many still heavily depend on manual testing. Manual testing is not only labor-intensive but also lacks repeatability. DevOps places great importance on automating tests to ensure that they can be repeated with every release, enabling the detection of existing issues and thorough testing of new features. Additionally, automation is essential for efficiently conducting regression testing on bug fixes, as manual testing in such cases is inefficient.

Therefore, embedding security and testing at the early stages of development is an essential goal for modern DevOps. Embedding security with DevOps has led to the concept of DevSecOps, where developers, cybersecurity experts, and operations teams work together to create better and more secure software faster.

Securing and testing your software using CI/CD pipelines offers various significant business advantages. Firstly, it ensures security by protecting sensitive data, preventing vulnerabilities, and ensuring compliance. Secondly, it improves quality and reliability through early issue detection, consistency, and higher product quality. This leads to cost reduction by reducing rework, speeding up time to market, and optimizing resource usage. Additionally, it mitigates risks by increasing resilience and enabling stress testing. Moreover, it ensures business continuity through disaster recovery and efficient rollback procedures. Furthermore, it provides a competitive advantage by fostering faster innovation and market responsiveness. Finally, it enhances reputation and customer trust by building confidence in your products and services and safeguarding your brand's reputation. In essence, securing and testing CI/CD pipelines is both a technical necessity and a strategic business imperative that enhances security, quality, and reliability while reducing costs and risks, ultimately leading to improved customer satisfaction, business continuity, and a competitive edge in the market.

There are many ways of embedding security within the software supply chain. Some of these might include static code analysis, security testing, and applying organization-specific security policies within the process, but the idea of security is not to slow down development. Instead of human input, we can always use tools that can significantly improve the security posture of the software we develop. Similarly, testing need not be manual and slow and, instead, should use automation to plug in seamlessly with the CI/CD process.

CI/CD pipelines are one of the essential features of modern DevOps, and they orchestrate all processes and combine all tools to deliver better software faster, but how would you secure them? You may want to ask the following questions:

- How do I scan a container image for vulnerabilities?
- How do I store and manage sensitive information and secrets securely?
- How do I ensure that my application is tested before deployment to production?
- How do I ensure that only tested and approved container images are deployed in production?

Throughout this chapter, we will try to answer these using best practices and tooling. For reference, look at the following workflow diagram:

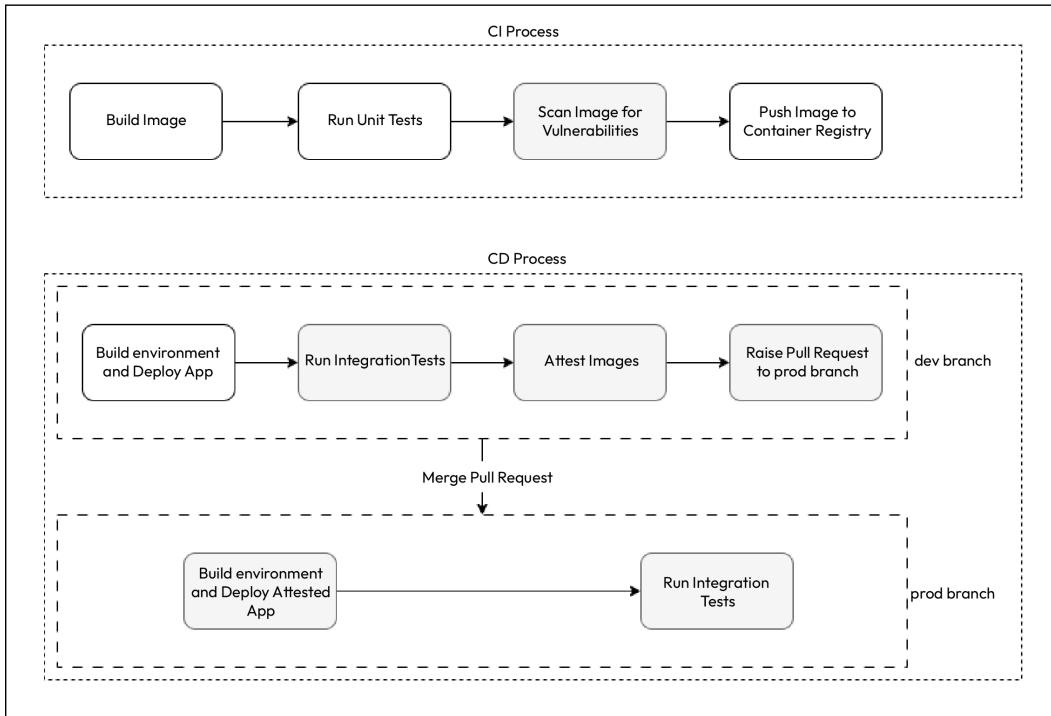


Figure 13.1 – Secure CI/CD workflow

As depicted in the previous figure, we need to modify the CI pipeline to include an additional step for vulnerability scanning. We also require two CD pipelines, one for the Dev environment and another for Prod. To enhance reusability, we'll restructure our GitHub Actions workflow. We'll divide the workflows into parent and child workflows. Let's begin by examining the CD workflow for the Dev environment to get an overview:

```

name: Dev Continuous Delivery Workflow
on:
  push:
    branches: [ dev ]
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./github/workflows/create-cluster.yml
    secrets: inherit
  run-tests:
    name: Run Integration Tests
    needs: [create-environment-and-deploy-app]
    uses: ./github/workflows/run-tests.yml
  
```

```
secrets: inherit
binary-auth:
  name: Attest Images
  needs: [run-tests]
  uses: ./github/workflows/attest-images.yml
  secrets: inherit
raise-pull-request:
  name: Raise Pull Request
  needs: [binary-auth]
  uses: ./github/workflows/raise-pr.yml
  secrets: inherit
```

The workflow begins with a name, followed by a declaration of `on push branches dev`. This configuration ensures that the workflow triggers with every push to the `dev` branch. We define multiple jobs in sequence, each depending on the previous one using the `needs` attribute. Each job invokes a child workflow specified by the `uses` attribute, and it provides GitHub secrets to these child workflows by setting `inherit` for the `secrets` attribute.

The workflow accomplishes the following tasks:

1. Sets up the Dev Kubernetes cluster, configures Argo CD and supporting tools to establish the environment, and deploys the sample Blog App.
2. Executes integration tests on the deployed Blog App.
3. If the tests pass, it utilizes binary authorization (more details to follow) to attest images, ensuring that only tested artifacts are allowed for deployment to production.
4. Initiates a pull request for deployment to the Prod environment.

In a similar manner, we have the following Prod CD Workflow file:

```
name: Prod Continuous Delivery Workflow
on:
  push:
    branches: [ prod ]
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./github/workflows/create-cluster.yml
    secrets: inherit
  run-tests:
    name: Run Integration Tests
    needs: [create-environment-and-deploy-app]
    uses: ./github/workflows/run-tests.yml
    secrets: inherit
```

This workflow is similar to the Dev workflow but does not include the `binary-auth` and `raise-pull-request` steps, as they are unnecessary at this stage. To understand it better, let's begin by examining the Dev workflow. The initial step of the Dev workflow involves creating the environment and deploying the application. However, before we proceed, let's revisit the Blog App in the next section.

Revisiting the Blog Application

As we already discussed the Blog App in the last chapter, let's look at the services and their interactions again in the following diagram:

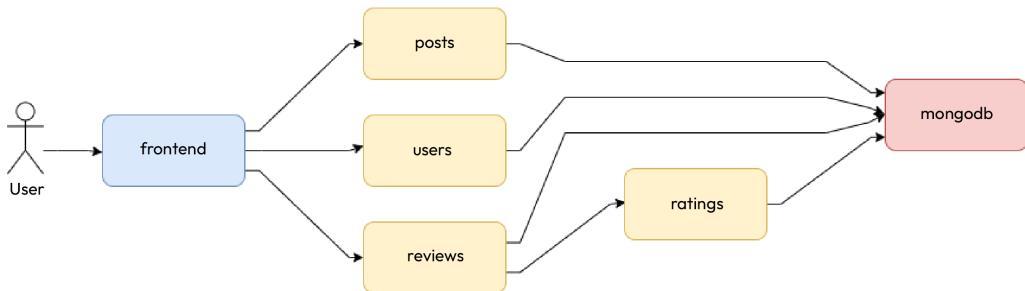


Figure 13.2 – The Blog App services and interactions

We've already created CI and CD pipelines for building, testing, and pushing our Blog Application microservices containers using GitHub Actions and deploying them using Argo CD in a GKE cluster.

If you remember, we created the following resources for the application to run seamlessly:

- **MongoDB** – We deployed an auth-enabled MongoDB database with root credentials. The credentials were injected via environment variables sourced from a Kubernetes **Secret** resource. To persist our database data, we created a **PersistentVolume** mounted to the container, which we provisioned dynamically using a **PersistentVolumeClaim**. As the container is stateful, we used a **StatefulSet** to manage it and, therefore, a headless Service to expose the database.
- **Posts, reviews, ratings, and users** – The posts, reviews, ratings, and users microservices interacted with MongoDB through the root credentials injected via environment variables sourced from the same **Secret** resource as MongoDB. We deployed them using their respective **Deployment** resources and exposed all of them via individual **ClusterIP** Services.
- **Frontend** – The frontend microservice does not need to interact with MongoDB, so there was no interaction with the **Secret** resource. We deployed this service as well using a **Deployment** resource. As we wanted to expose the service on the internet, we created a **LoadBalancer** Service for it.

We can summarize them in the following diagram:

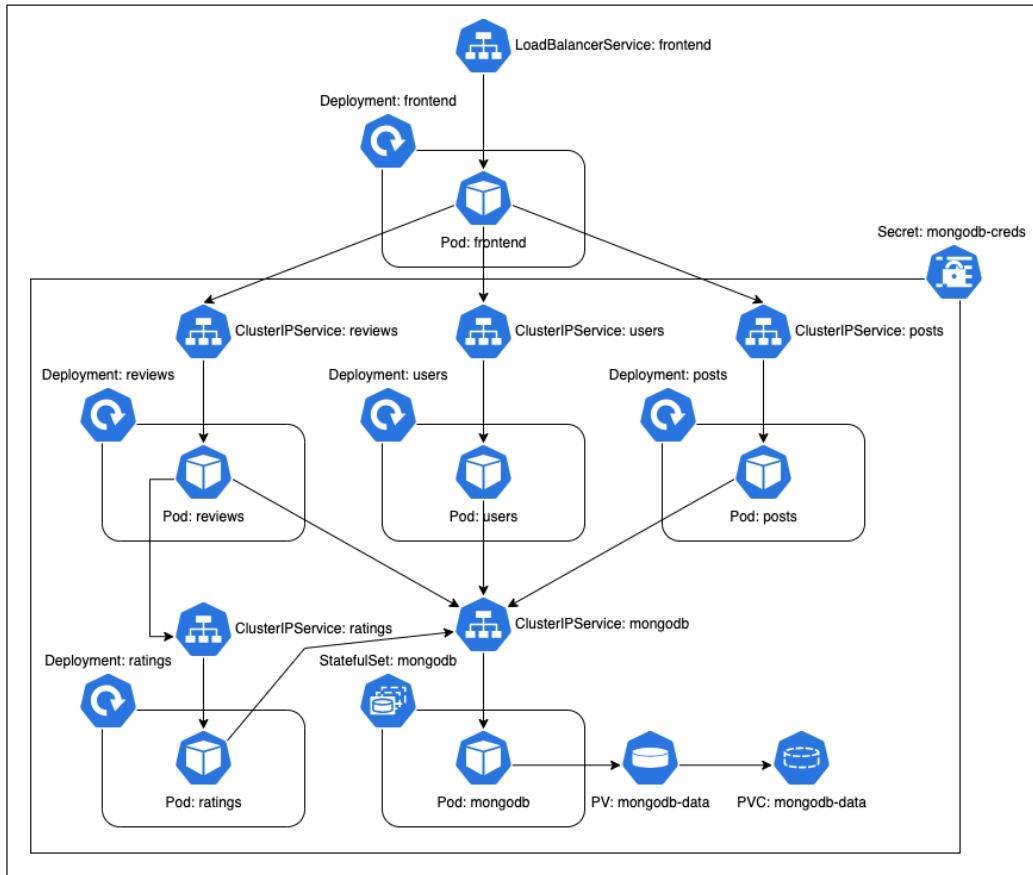


Figure 13.3 – The Blog App – Kubernetes resources and interactions

In subsequent sections, we will cover all aspects of implementing this workflow, starting with vulnerability scanning.

Container vulnerability scanning

Perfect software is costly to write and maintain, and every time someone makes changes to running software, the chances of breaking something are high. Apart from other bugs, changes also add a lot of software vulnerabilities. You cannot avoid these as software developers. Cybersecurity experts and cybercriminals are at constant war with each other, evolving with time. Every day, a new set of vulnerabilities are found and reported.

In containers, vulnerabilities can exist on multiple fronts and may be completely unrelated to what you're responsible for. Well, developers write code, and excellent ones do it securely. Still, you never know whether a base image may contain vulnerabilities your developers might completely overlook. In modern DevOps, vulnerabilities are expected, and the idea is to mitigate them as much as possible. We should reduce vulnerabilities, but doing so manually is time-consuming, leading to toil.

Several tools are available on the market that provide container vulnerability scanning. Some of them are open source tools such as **Anchore**, **Clair**, **Dagda**, **OpenSCAP**, Sysdig's **Falco**, or **Software-as-a-Service (SaaS)** services available with **Google Container Registry (GCR)**, **Amazon Elastic Container Registry (ECR)**, and **Azure Defender**. For this chapter, we'll discuss **Anchore Grype**.

Anchore Grype (<https://github.com/anchore/grype>) is a container vulnerability scanner that scans your images for known vulnerabilities and reports their severity. Based on that, you can take appropriate actions to prevent vulnerabilities by including a different base image or modifying the layers to remove vulnerable components.

Anchore Grype is a simple **Command-Line Interface (CLI)**-based tool that you can install as a binary and run anywhere—within your local system or your CI/CD pipelines. You can also configure it to fail your pipeline if the vulnerability level increases above a particular threshold, thereby embedding security within your automation—all this happening without troubling your development or security team.

Now, let's go ahead and see Anchore Grype in action.

Installing Anchore Grype

As we want to implement vulnerability scanning within our CI pipelines, let's modify the `mdo-posts` repository we created in *Chapter 11*.

Let's clone the repository first using the following command and `cd` into the `workflows` directory:

```
$ git clone git@github.com:<your_github_user>/mdo-posts.git  
$ cd mdo-posts/.github/workflows/
```

Anchore Grype offers an installation script within its GitHub repository that you can download and run, and it should set it up for you. We'll modify the `build.yaml` file to include the following step before the `Login to Docker Hub` step so that we can install Grype within our CI workflow:

```
- name: Install Grype  
  id: install-grype  
  run: curl -ssfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s  
    -- -b /usr/local/bin
```

Next, we need to use Grype to scan our images for vulnerabilities.

Scanning images

To run container vulnerability scanning, we can use the following command:

```
$ grype <container-image>
```

This will report a list of vulnerabilities with severities—**Negligible**, **Low**, **Medium**, **High**, **Critical**, or **Unknown**—within the image. We can also set a threshold within Grype to fail when any vulnerabilities are equal to or worse than it. For example, if we don't want to allow any **Critical** vulnerabilities in the container, we can use the following command:

```
$ grype -f critical <container-image>
```

To do so, we will add the following step within the `build.yaml` file after the `Build the Docker image` step:

```
- name: Scan Image for Vulnerabilities
  id: vul-scan
  run: grype -f critical ${{ secrets.DOCKER_USER }}/mdo-posts:$!(git rev-parse --short
"$GITHUB_SHA")
```

As we've made all the changes, let's push the modified CI pipeline using the following commands:

```
$ cp ~/modern-devops/ch13/grype/build.yaml .
$ git add --all
$ git commit -m "Added grype"
$ git push
```

As soon as we push the image, we will see the following in the GitHub Actions tab:

The screenshot shows a GitHub Actions build log for a job named "build". The log details the execution of various steps, including "Set up job", "Run actions/checkout@v2", "Install Grype", "Login to Docker Hub", "Build the Docker image", and "Scan Image for Vulnerabilities". The "Scan Image for Vulnerabilities" step failed with an exit code of 1, which triggered an error message: "Error: Process completed with exit code 1.". The log also shows the output of the Grype scan, listing several vulnerabilities found in installed Python packages.

```

build
failed 1 minute ago in 39s
Search logs

> Set up job
1s

> Run actions/checkout@v2
1s

> Install Grype
2s

> Login to Docker Hub
0s

> Build the Docker image
28s

-> Scan Image for Vulnerabilities
13s

1 ▶ Run grype -f critical ***/mdo-posts:$(git rev-parse --short "$GITHUB_SHA")
2 1 error occurred:
3 NAME      INSTALLED FIXED-IN TYPE    VULNERABILITY      SEVERITY
4       * discovered vulnerabilities at or above the severity threshold
5 pip        23.0.1          python  CVE-2018-20225   High
6
7 python    3.7.17           binary  CVE-2022-48565   Critical
8 python    3.7.17           binary  CVE-2023-36632   High
9 python    3.7.17           binary  CVE-2022-48566   High
10 python   3.7.17           binary  CVE-2022-48560  High
11 python   3.7.17           binary  CVE-2023-40217  Medium
12 python   3.7.17           binary  CVE-2023-27043  Medium
13 python   3.7.17           binary  CVE-2022-48564  Medium
14 python   3.7.17           binary  CVE-2007-4559   Medium
15 python   3.7.17           binary  GHSA-r9hx-vwmv-q579 High
16 python   3.7.17           python  CVE-2022-40897  Medium
17 setuptools 57.5.0          65.5.1   python
18 setuptools 57.5.0          65.5.1   python
19 Error: Process completed with exit code 1.

Push the Docker image
0s

Post Run actions/checkout@v2
1s

Complete job
0s

```

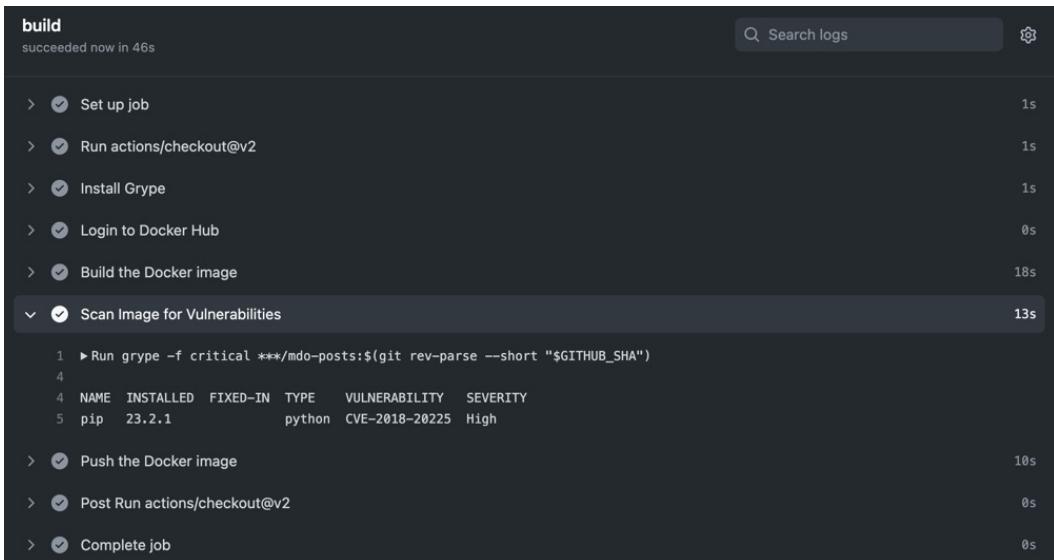
Figure 13.4 – Vulnerability scan failure

As we can see, Grype has reported several vulnerabilities with one being **Critical**. It has also failed the CI pipeline. That is automated vulnerability scanning in action. This will discover vulnerabilities and only allow builds to end up in your container registry if they meet minimum security standards.

We need to fix the issue here, so let's look at a more recent image and see whether it can fix the problem. Therefore, instead of using `python:3.7-alpine`, we will use `python:alpine3.18`. Let's do that and push our code to GitHub using the following commands:

```
$ cd ~/mdo-posts && cp ~/modern-devops/ch13/grype/Dockerfile .
$ git add --all
$ git commit -m "Updated base image"
$ git push
```

Let's revisit GitHub Actions and see what we get in the build output:



The screenshot shows a GitHub Actions build log for a 'build' workflow. The build succeeded in 46s. The steps listed are: Set up job (1s), Run actions/checkout@v2 (1s), Install Grype (1s), Login to Docker Hub (0s), Build the Docker image (18s), Scan Image for Vulnerabilities (13s). The 'Scan Image for Vulnerabilities' step shows the command: 'Run grype -f critical ***/mdo-posts:\${git rev-parse --short "\$GITHUB_SHA")'. It lists one vulnerability: NAME: pip, INSTALLED: 23.2.1, FIXED-IN: python, VULNERABILITY: CVE-2018-20225, SEVERITY: High. The final steps are Push the Docker image (10s), Post Run actions/checkout@v2 (0s), and Complete job (0s).

Figure 13.5 – Vulnerability scan success

The vulnerability scan did not stop our CI build this time, as no Critical vulnerabilities were found.

Tip

Continually update the base image with time, as newer ones contain fewer vulnerabilities and fix older ones.

Now that we've secured the image for vulnerabilities, our CI pipeline is complete. You can replicate this process for other microservices as needed. Let's proceed to discuss CD pipelines.

If you remember, in the last chapter, following the GitOps model, we stored the manifests of all resources on Git. However, due to security concerns with Kubernetes Secrets, we used **SealedSecrets** to manage them securely.

However, this may not be the ideal solution for all teams due to the following inherent issues:

- SealedSecrets are reliant on the controller that encrypts them. If we lose this controller, we also lose the ability to recreate the secret, essentially losing the Secret forever.
- Access to the Secret is limited to logging in to the cluster and using `kubectl`, which doesn't provide non-admins with the ability to manage secrets. While this approach might suit some teams, it may not suit others.

Therefore, we will explore managing secrets using a Secrets management tool to establish a standardized method for centrally managing secrets with more granular control over access. Let's delve into this topic in the next section.

Managing secrets

Software always requires access to sensitive information such as user data, credentials, **Open Authorization (OAuth)** tokens, passwords, and other information known as secrets. Developing and managing software while keeping all these aspects secure has always been a concern. The CI/CD pipelines might deal with them as they build and deliver working software by combining code and other dependencies from various sources that may include sensitive information. Keeping these bits secure is of utmost importance; therefore, the need arises to use modern DevOps tools and techniques to embed security within the CI/CD pipelines themselves.

Most application code requires access to sensitive information. These are called **secrets** in the DevOps world. A secret is any data that helps someone prove their identity, authenticate, and authorize privileged accounts, applications, and services. Some of the potential candidates that constitute secrets are listed here:

- Passwords
- API tokens, GitHub tokens, and any other application key
- **Secure Shell (SSH)** keys
- **Transport Layer Security (TLS)**, **Secure Sockets Layer (SSL)**, and **Pretty Good Privacy (PGP)** private keys
- One-time passwords

A good example could be a container requiring access to an API key to authenticate with a third-party API or a username and password to authenticate with a backend database. Developers need to understand where and how to store secrets so that they are not exposed inadvertently to people who are not supposed to view them.

When we run a CI/CD pipeline, it becomes imperative to understand how we place those secrets as, in CI/CD pipelines, we build everything from the source. “*Do not store secrets with code*” is a prominent piece of advice we’ve all heard.

Tip

Never store hardcoded secrets within CI/CD pipelines or store secrets in a source code repository such as Git.

How can we access secrets without including them in our code to run a fully automated GitOps-based CI/CD pipeline? Well, that's something we need to figure out.

Tip

When using containers, the thing to avoid is baking the secrets within an image. While this is a prominent piece of advice, many developers do this inadvertently, leading to many security holes. It is very insecure, and you should avoid doing it at all costs.

You can overcome this problem by using some form of **secrets management solution**. A secrets management solution or a **key management solution** helps store and manage your secrets and secure them with encryption at rest and in transit. There are secrets management tools within cloud providers, such as **Secret Manager** in GCP and **Amazon Web Services (AWS)**, or you can use a third-party tool, such as **HashiCorp Vault**, if you want to go cloud agnostic. All these solutions provide APIs to create and query secrets at runtime, and they secure the API via HTTPS to allow encryption in transit. That way, you don't need to store your secrets with code or bake it within an image.

In this discussion, we'll use the **Secret Manager** solution offered by GCP to store secrets, and we will access them while running the CI/CD pipeline. Secret Manager is Google Cloud's secrets management system, which helps you store and manage secrets centrally. It is incredibly secure and uses **Hardware Security Modules (HSMs)** to harden your secrets further.

In this chapter, we will look at improving the CI/CD pipeline of our Blog Application, which we discussed in the last chapter, and will use the same sample application. Therefore, let's go ahead and create the `mongodb-creds` Secret in Google Cloud Secret Manager.

Creating a Secret in Google Cloud Secret Manager

Let's create a secret called `external-secrets`, where we will pass the MongoDB credentials in JSON format. To do so, run the following command:

```
$ echo -ne \  
'{"MONGO_INITDB_ROOT_USERNAME": "root", "MONGO_INITDB_ROOT_PASSWORD": "itsasecret"}' \  
| gcloud secrets create external-secrets --locations=us-central1 \  
--replication-policy=user-managed --data-file=-- \  
Created version [1] of the secret [external-secrets].
```

In the preceding command, we echo a JSON containing `MONGO_INITDB_ROOT_USERNAME` and `PASSWORD` directly into the `gcloud secrets create` command. We have specified a particular location to avoid replicating it in other regions as a cost-saving measure. However, it's highly recommended to replicate secrets to prevent potential loss in case of a zonal outage. The JSON is stored as a new version of our secret. Secret Manager utilizes versioning for secrets, so any new value assigned to the secret (`external-secrets`) is versioned and stored within Secret Manager. You can reference a specific version either by its version number or by using the `latest` keyword to access the most recent version.

As seen in the output, we've created the first version of our secret (version 1). Typically, this is done during development and should remain outside the CI/CD process. Instead of storing the Secret resource manifest in your source code repository, you can keep it in Secret Manager.

Now that we've created the secret, we must access it within our application. To achieve this, we require a tool to access the secret stored in Secret Manager from the Kubernetes cluster. For this purpose, we will use **External Secrets Operator**.

Accessing external secrets using External Secrets Operator

External Secrets Operator (<https://external-secrets.io/latest/>) is a Kubernetes operator used in Kubernetes clusters to manage external secrets securely. It is designed to automate the retrieval and management of secrets stored in external secret stores such as AWS Secret Manager, GCP Secret Manager, Hashicorp Vault, and so on, and inject them into Kubernetes pods as Kubernetes Secrets. Operators are a way to extend Kubernetes functionality and automate tasks.

How it works

External Secrets Operator serves as a bridge between the Kubernetes cluster and external secret management systems. We define an `ExternalSecret` custom resource within the Kubernetes cluster, which the operator monitors. When an `ExternalSecret` resource is created or updated, the operator interacts with the external secret store specified in the `ClusterSecretStore` CRD to retrieve the secret data. It then creates or updates the corresponding Kubernetes Secrets. This process is illustrated in the following diagram:

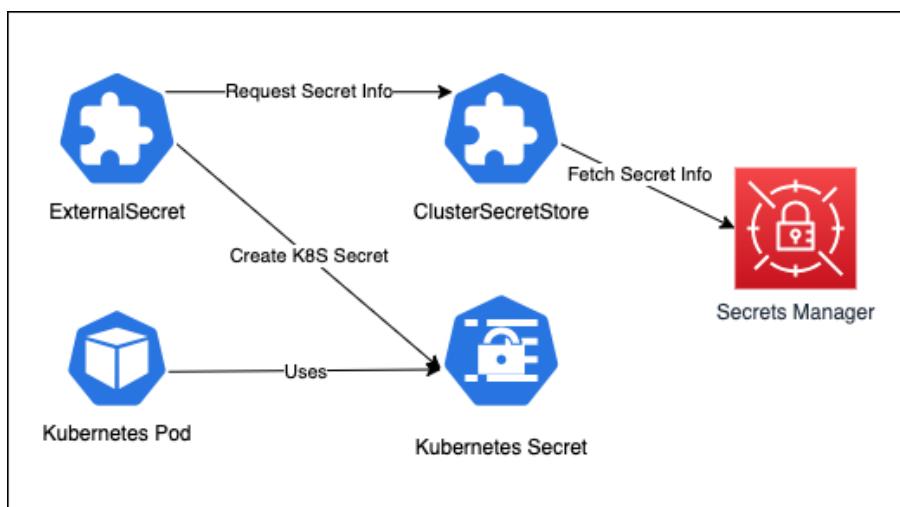


Figure 13.6 – External Secret Operator

Now, this process has a lot of benefits, some of which are as follows:

- **Enhanced Security:** Secrets remain in a dedicated, secure secret store
- **Automation:** Automates the retrieval and rotation of secrets
- **Simplified Deployment:** Eases the management of secrets within Kubernetes applications
- **Compatibility:** Works with various external secret stores, making it versatile

Now, let's go ahead and install External Secrets Operator on our Kubernetes cluster.

Installing External Secrets Operator

External Secrets Operator is available as a **Helm chart**, and Argo CD supports it. A Helm chart is a collection of preconfigured Kubernetes resources (such as Deployments, Services, ConfigMaps, and more) organized into a package that makes it easy to deploy and manage applications in Kubernetes. Helm is a package manager for Kubernetes that allows you to define, install, and upgrade even the most complex Kubernetes applications in a repeatable and standardized way. Therefore, we must create an Argo CD application pointing to the Helm chart to install it. To do so, we will create the following `manifests/argocd/external-secrets.yaml` manifest file:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: external-secrets
  namespace: argocd
spec:
  project: default
  source:
    chart: external-secrets/external-secrets
    repoURL: https://charts.external-secrets.io
    targetRevision: 0.9.4
    helm:
      releaseName: external-secrets
  destination:
    server: "https://kubernetes.default.svc"
    namespace: external-secrets
```

The application manifest creates an `external-secrets` application on the `argocd` namespace within the `default` project. It downloads the `0.9.4` revision from the `external-secrets` Helm chart repository and deploys the chart on the Kubernetes cluster on the `external-secrets` namespace.

To install this application, we need to apply this manifest using Terraform. Therefore, to do so, we make the following entry in the `app.tf` file:

```
data "kubectl_file_documents" "external-secrets" {
  content = file("../manifests/argocd/external-secrets.yaml")
}
```

```

resource "kubectl_manifest" "external-secrets" {
  depends_on = [
    kubectl_manifest.argocd,
  ]
  for_each  = data.kubectl_file_documents.external-secrets.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}

```

To deploy this, we must check these files into source control. Let's clone the mdo-environments repository that we created in the last chapters.

If you haven't followed the last chapters, you can do the following to set a baseline. Feel free to skip the next section if you've already set up your environment in *Chapter 12, Continuous Deployment/Delivery with Argo CD*.

Setting up the baseline

To ensure continuity with the last chapters, let's start by creating a service account for Terraform to interact with our GCP project using the following commands:

```

$ gcloud iam service-accounts create terraform \
--description="Service Account for terraform" \
--display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com

```

You will see a file called `key-file` created within your working directory. Now, create a new repository called mdo-environments with a `README.md` file on GitHub, rename the main branch to `prod`, and create a new branch called `dev` using GitHub. Navigate to https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new and create a secret named `GCP_CREDENTIALS`. For the value, print the `key-file` file, copy its contents, and paste it into the `values` field of the GitHub secret.

Next, create another secret, `PROJECT_ID`, and specify your GCP project ID within the `values` field.

Next, we need to create a GCS bucket for Terraform to use as a remote backend. To do this, run the following command:

```
$ gsutil mb gs://tf-state-mdt-terraorm-$PROJECT_ID
```

So, now that all the prerequisites are met, we can clone our repository and copy the baseline code. Run the following commands to do this:

```
$ cd ~ && git clone git@github.com:<your_github_user>/mdo-environments.git  
$ cd mdo-environments/  
$ git checkout dev  
$ cp -r ~/modern-devops/ch13/baseline/* .  
$ cp -r ~/modern-devops/ch13/baseline/.github .
```

As we're now on the baseline, let's proceed further to install external secrets with Terraform.

Installing external secrets with Terraform

Let's configure our local repository to install the external secrets manifest. To do so, copy the application manifest and `app.tf` file using the following commands:

```
$ cp ~/modern-devops/ch13/install-external-secrets/app.tf terraform/app.tf  
$ cp ~/modern-devops/ch13/install-external-secrets/external-secrets.yaml \  
manifests/argocd/
```

Now that we're all set up and ready, let's go ahead and commit and push our code using the following commands:

```
$ git add --all  
$ git commit -m "Install external secrets operator"  
$ git push
```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. To access the workflow, go to `https://github.com/<your_github_user>/mdo-environments/actions`. Soon, the workflow will apply the configuration, create the Kubernetes cluster, and deploy Argo CD, the Sealed Secrets controller, and External Secrets Operator.

Once the workflow is successful, we can do the following to access the Argo Web UI.

We must first authenticate with the GKE cluster. To do so, run the following command:

```
$ gcloud container clusters get-credentials \  
mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID
```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```
$ kubectl get svc argocd-server -n argocd  
NAME          TYPE      EXTERNAL-IP    PORTS          AGE  
argocd-server LoadBalancer 34.122.51.25  80/TCP,443/TCP 6m15s
```

So, now we know that Argo CD is accessible on `https://34.122.51.25/`.

Next, we will run the following commands to reset the admin password:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow two minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
-o jsonpath="{.data.password}" | base64 -d && echo
```

As we now have the credentials, log in, and you will see the following page:

The screenshot shows the Argo CD Web UI home page. At the top, there is a header with buttons for '+ NEW APP', 'SYNC APPS', 'REFRESH APPS', a search bar, and a 'Log out' button. To the right of the search bar are icons for 'APPLICATIONS TILES' and a refresh symbol. Below the header, there is a sorting dropdown set to 'name' and a 'Items per page: 10' dropdown. The main area displays three application cards:

- blog-app**: Project: default, Labels: argocd.argoproj.io/application-set..., Status: Degraded (red heart), Synced (green circle). Repository: https://github.com/bharatmicosys... Target: dev, Path: manifests/blog-app, Destination: in-cluster, Names: blog-app. Created: 09/04/2023 18:41:56 (9 minutes ago), Last Sync: 09/04/2023 18:50:28 (a few seconds ago). Buttons: SYNC, C, X.
- external-secrets**: Project: default, Labels: argocd.argoproj.io/application-set..., Status: Healthy (green heart), Synced (green circle). Repository: https://charts.external-secrets.io Target: 0.9.4, Chart: external-secrets, Destination: in-cluster, Names: external-secrets. Created: 09/04/2023 18:41:56 (9 minutes ago), Last Sync: 09/04/2023 18:49:09 (a minute ago). Buttons: SYNC, C, X.
- sealed-secrets**: Project: default, Labels: argocd.argoproj.io/application-set..., Status: Healthy (green heart), Synced (green circle). Repository: https://github.com/bharatmicosys... Target: dev, Path: manifests/sealed-secrets, Destination: in-cluster, Names: sealed-secrets. Created: 09/04/2023 18:41:56 (9 minutes ago), Last Sync: 09/04/2023 18:42:00 (8 minutes ago). Buttons: SYNC, C, X.

Figure 13.7 – Argo CD Web UI – home page

As we can see, there are three applications – **sealed-secrets**, **external-secrets**, and **blog-app**. While the **sealed-secrets** and **external-secrets** apps are all synced up and green, **blog-app** has degraded. That is because, in my case, I've started fresh and created a new cluster. Therefore, there is no way the Sealed Secrets operator can decrypt the SealedSecret manifest that we created in the last chapter, as it was generated by a different Sealed Secrets controller.

We don't need the Sealed Secrets operator; we will use Google Cloud Secret Manager instead. So, let's remove it from our cluster using the following commands:

```
$ rm -rf manifests/sealed-secrets  
$ git add --all  
$ git commit -m "Removed sealed secrets"  
$ git push
```

We've removed the Sealed Secrets operator, and the Argo CD Web UI should reflect that shortly. However, the Blog Application will remain degraded as the `mongodb-creds` Secret is still missing. In the next section, we will use External Secrets Operator to generate the `mongodb-creds` Secret.

Generating the MongoDB Kubernetes Secret using External Secrets Operator

To generate the `mongodb-creds` secret, we would need to create the following resources:

- A `Secret` resource – This is a standard Kubernetes Secret resource containing the service account credentials for Kubernetes to connect with GCP Secret Manager.
- A `ClusterSecretStore` resource – This resource contains configuration for connecting with the secret store (GCP Secret Manager in this case) and uses the `Secret` resource for the service account credentials.
- An `ExternalSecret` resource – This resource contains configuration to generate the required Kubernetes Secret (`mongodb-creds`) out of the extracted Secret from the secret store.

So, let's go ahead and define the `Secret` resource first:

To create the `Secret` resource, we first need to create a GCP service account to interact with Secret Manager using the following commands:

```
$ cd ~  
$ gcloud iam service-accounts create external-secrets
```

As we're following the principle of least privilege, we will add the following role-binding to provide access only to the `external-secrets` secret, as follows:

```
$ gcloud secrets add-iam-policy-binding external-secrets \  
--member "serviceAccount:external-secrets@$PROJECT_ID.iam.gserviceaccount.com" \  
--role "roles/secretmanager.secretAccessor"
```

Now, let's generate the service account key file using the following command:

```
$ gcloud iam service-accounts keys create key.json \  
--iam-account=external-secrets@$PROJECT_ID.iam.gserviceaccount.com
```

Now, copy the contents of the `key.json` file into a new GitHub Actions secret called `GCP_SM_CREDENTIALS`. We will use GitHub Actions to set this value during runtime dynamically; therefore, the following secret manifest will contain a placeholder:

```
apiVersion: v1
data:
  secret-access-credentials: SECRET_ACCESS_CREDS_PH
kind: Secret
metadata:
  name: gcpsm-secret
type: Opaque
```

Let's look at the `ClusterSecretStore` resource next:

```
apiVersion: external-secrets.io/v1alpha1
kind: ClusterSecretStore
metadata:
  name: gcp-backend
spec:
  provider:
    gcpsm:
      auth:
        secretRef:
          secretAccessKeySecretRef:
            name: gcpsm-secret
            key: secret-access-credentials
      projectID: PROJECT_ID_PH
```

The manifest defines the following:

- A `ClusterSecretStore` resource called `gcp-backend`
- A provider configuration of the `gcpsm` type using auth information in the `gcpsm-secret` secret we defined before

Now, let's look at the `ExternalSecret` resource manifest:

```
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  name: mongodb-creds
  namespace: blog-app
spec:
  secretStoreRef:
    kind: SecretStore
    name: gcp-backend
  target:
    name: mongodb-creds
  data:
  - secretKey: MONGO_INITDB_ROOT_USERNAME
    remoteRef:
```

```
key: external-secrets
property: MONGO_INITDB_ROOT_USERNAME
- secretKey: MONGO_INITDB_ROOT_PASSWORD
remoteRef:
  key: external-secrets
  property: MONGO_INITDB_ROOT_PASSWORD
```

The manifest defines an `ExternalSecret` resource with the following specs:

- It is named `mongodb-creds` in the `blog-app` namespace.
- It refers to the `gcp-backend` `ClusterSecretStore` that we defined.
- It maps `MONGO_INITDB_ROOT_USERNAME` from the `external-secrets` Secret Manager secret to the `MONGO_INITDB_ROOT_USERNAME` key of the `mongodb-creds` Kubernetes secret. It does the same for `MONGO_INITDB_ROOT_PASSWORD`.

Now, let's deploy these resources by using the following commands:

```
$ cd ~/mdo-environments
$ cp ~/modern-devops/ch13/configure-external-secrets/app.tf terraform/app.tf
$ cp ~/modern-devops/ch13/configure-external-secrets/gcpsm-secret.yaml \
manifests/argocd/
$ cp ~/modern-devops/ch13/configure-external-secrets/mongodb-creds-external.yaml \
manifests/blog-app/
$ cp -r ~/modern-devops/ch13/configure-external-secrets/.github .
$ git add --all
$ git commit -m "Configure External Secrets"
$ git push
```

This should trigger a GitHub Actions workflow again, and soon, we should see `ClusterSecretStore` and `ExternalSecret` created. To check that, run the following commands:

```
$ kubectl get secret gcpsm-secret
NAME      TYPE      DATA   AGE
gcpsm-secret  Opaque    1     1m
$ kubectl get clustersecretstore gcp-backend
NAME      AGE      STATUS      CAPABILITIES      READY
gcp-backend  19m    Valid      ReadWrite      True
$ kubectl get externalsecret -n blog-app mongodb-creds
NAME      STORE      REFRESHINTERVAL      STATUS      READY
mongodb-creds  gcp-backend  1h0m0s      SecretSynced      True
$ kubectl get secret -n blog-app mongodb-creds
NAME      TYPE      DATA   AGE
mongodb-creds  Opaque    2     4m45s
```

The same should be reflected in the blog-app application on Argo CD, and the application should come up clean, as shown in the following screenshot:

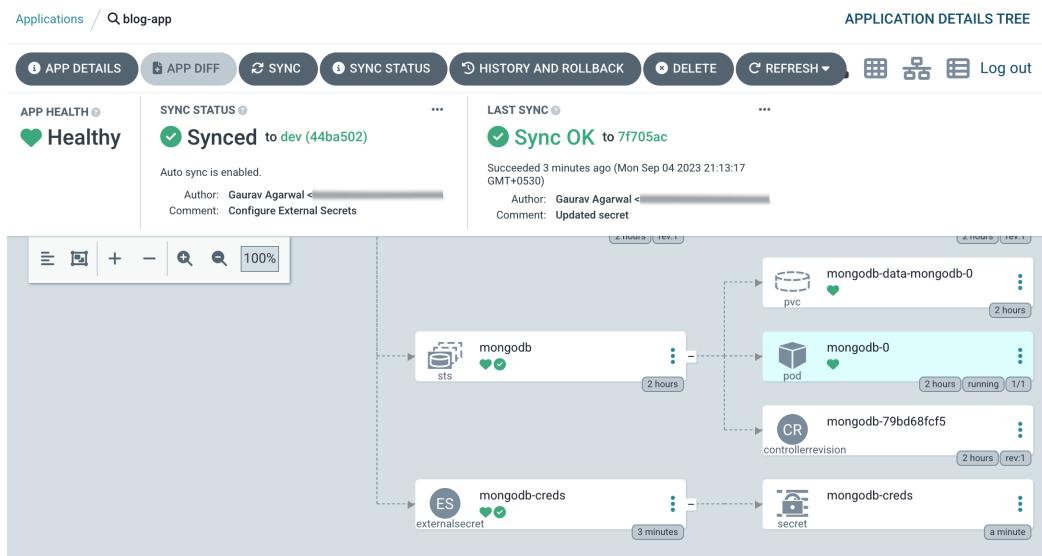


Figure 13.8 – blog-app showing as Healthy

You can then access the application by getting the frontend service external IP using the following command:

```
$ kubectl get svc -n blog-app frontend
NAME      TYPE      EXTERNAL-IP      PORT(S)      AGE
frontend  LoadBalancer  34.122.58.73  80:30867/TCP  153m
```

You can access the application by visiting `http://<EXTERNAL_IP>` from a browser:

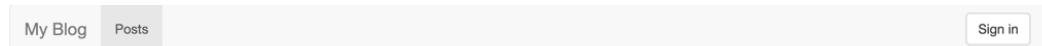


Figure 13.9 – Blog App home page

And as we can see, we can access the Blog App successfully. That is proper secret management, as we did not store the secret in the source code repository (Git). We did not view or log the secret while applying it, meaning there is no trace of this secret anywhere in the logs, and only the application or people who have access to the namespace where this application is running can access it. Now, let's look at another crucial aspect: testing your application.

Testing your application within the CD pipeline

Until now, we've deployed our application on a Kubernetes cluster and manually verified that it is running. We have two options moving forward: either proceed with manual testing or create automated tests, also known as a **test suite**. While manual testing is the traditional approach, DevOps heavily emphasizes automating tests to integrate them into your CD pipeline. This way, we can eliminate many repetitive tasks, often called **toil**.

We've developed a Python-based integration test suite for our application, covering various scenarios. One significant advantage of this test suite is that it treats the application as a black box. It remains unaware of how the application is implemented, focusing solely on simulating end user interactions. This approach provides valuable insights into the application's functional aspects.

Furthermore, since this is an integration test, it assesses the entire application as a cohesive unit, in contrast to the unit tests we ran in our CI pipeline, where we tested each microservice in isolation.

Without further delay, let's integrate the integration test into our CD pipeline.

CD workflow changes

Till now, we have the following within our CD workflow:

```
.
├── create-cluster.yml
└── dev-cd-workflow.yaml
└── prod-cd-workflow.yaml
```

Both the Dev and Prod CD workflows contain the following jobs:

```
jobs:
  create-environment-and-deploy-app:
    name: Create Environment and Deploy App
    uses: ./github/workflows/create-cluster.yml
    secrets: inherit
```

As we can see, they are both calling the `create-cluster.yml` workflow, which creates our environment and deploys our application. We need to run integration tests both within the Dev and Prod environments; therefore, we need to change both workflows to include the `Run Integration Tests` step as follows:

```
run-tests:
  name: Run Integration Tests
  needs: [deploy-app]
  uses: ./github/workflows/run-tests.yml
  secrets: inherit
```

As we can see, the step calls the `run-tests.yml` workflow. That is the workflow that will be doing the integration tests. Let's look at the workflow to understand it better:

```
name: Run Integration Tests
on: [workflow_call]
jobs:
  test-application:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./tests
    steps:
      - uses: actions/checkout@v2
      - name: Extract branch name
        run: echo "branch=${GITHUB_HEAD_REF:-${GITHUB_REF#refs/heads/}}" >> $GITHUB_OUTPUT
        id: extract_branch
      - id: gcloud-auth
        name: Authenticate with gcloud
        uses: 'google-github-actions/auth@v1'
        with:
          credentials_json: '${{ secrets.GCP_CREDENTIALS }}'
      - name: Set up Cloud SDK
        id: setup-gcloud-sdk
        uses: 'google-github-actions/setup-gcloud@v1'
      - name: Get kubectl credentials
        id: 'get-credentials'
        uses: 'google-github-actions/get-gke-credentials@v1'
        with:
          cluster_name: mdo-cluster-${{ steps.extract_branch.outputs.branch }}
          location: ${secrets.CLUSTER_LOCATION}
      - name: Compute Application URL
        id: compute-application-url
        run: external_ip=$(kubectl get svc -n blog-app frontend --output jsonpath='{.status.loadBalancer.ingress[0].ip}') && echo $external_ip && sed -i "s/localhost/$external_ip/g" integration-test.py
        - id: run-integration-test
          name: Run Integration Test
          run: python3 integration-test.py
```

The workflow performs the following tasks:

1. It is triggered exclusively through a `workflow call`.
2. It has the `./tests` working directory.
3. It checks out the committed code.
4. It installs the `gcloud` CLI and authenticates with Google Cloud using the `GCP_CREDENTIALS` service account credentials.
5. It connects `kubectl` to the Kubernetes cluster to retrieve the application URL.
6. Using the application URL, it executes the integration test.

Now, let's proceed to update the workflow and add tests using the following commands:

```
$ cp -r ~/modern-devops/ch13/integration-tests/.github .
$ cp -r ~/modern-devops/ch13/integration-tests/tests .
$ git add --all
$ git commit -m "Added tests"
$ git push
```

This should trigger the Dev CD GitHub Actions workflow again. You should see something like the following:

The screenshot shows a GitHub Actions workflow run summary. On the left, there's a sidebar with links for 'Summary', 'Jobs' (which lists 'Create Environment and Deploy ...' and 'Run Integration Tests'), 'Run details', 'Usage', and 'Workflow file'. The main area has a header 'Triggered via push 1 minute ago' with a user icon 'bharatmicrosystems pushed 50be8c7 dev'. To the right are columns for 'Status' (Success), 'Total duration' (1m 10s), and 'Artifacts' (empty). Below this is a card for 'dev-cd-workflow.yml' triggered by 'push'. It contains two steps: 'Create ... / deploy-terraform' (22s) and 'Run Inte... / test-application' (27s). Both steps are marked with green checkmarks.

Figure 13.10 – Added tests workflow run

As we can see, there are two steps in our workflow, and both are now successful. To explore what was tested, you can click on the **Run Integration Tests** step, and it should show you the following output:

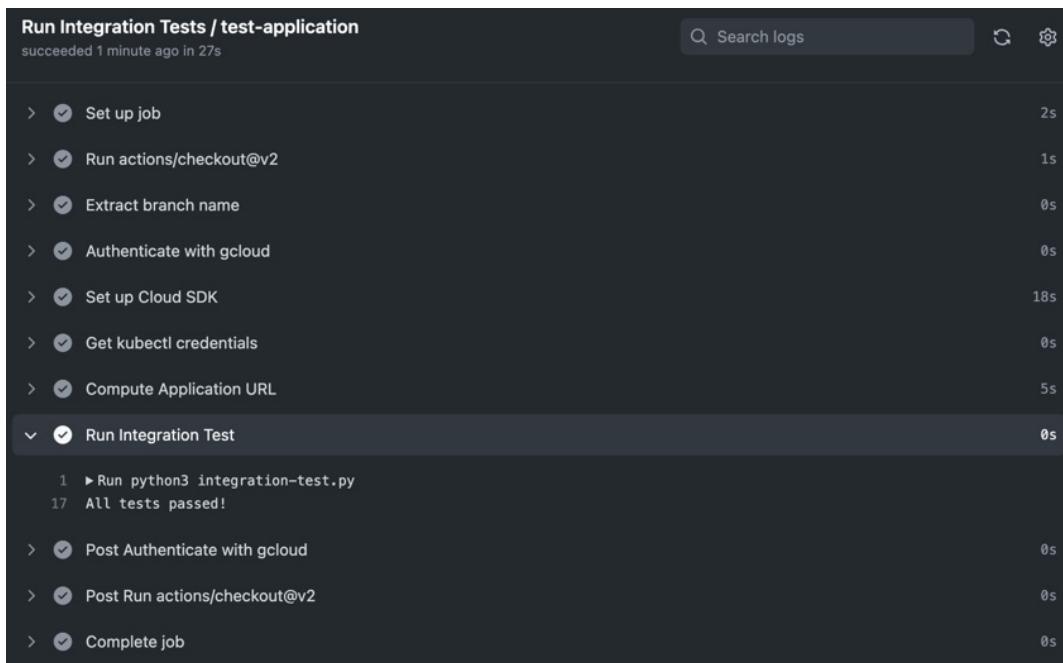


Figure 13.11 – The Run Integration Tests workflow step

As we can see, the **Run Integration Tests** step reports that all tests have passed.

While images are being built, deployed, and tested using your CI/CD toolchain, there is nothing in between to prevent someone from deploying an image in your Kubernetes cluster. You might be scanning all your images for vulnerabilities and mitigating them, but somewhere, someone might bypass all controls and deploy containers directly to your cluster. So, how can you prevent such a situation? The answer to that question is through binary authorization. Let's explore this in the next section.

Binary authorization

Binary authorization is a deploy-time security mechanism that ensures that only trusted binary files are deployed within your environments. In the context of containers and Kubernetes, binary authorization uses signature validation and ensures that only container images signed by a trusted authority are deployed within your Kubernetes cluster.

Using binary authorization gives you tighter control over what is deployed in your cluster. It ensures that only tested containers and those approved and verified by a particular authority (such as security tooling or personnel) are present in your cluster.

Binary authorization works by enforcing rules within your cluster via an admission controller. This means you can create rulesets only to allow images signed by an attestation authority to be deployed in your cluster. Your **quality assurance (QA)** team can be a good attestor in a practical scenario. You can also embed the attestation within your CI/CD pipelines. The attestation means your images have been tested and scanned for vulnerabilities and have passed a minimum standard to be ready to be deployed to the cluster.

GCP provides binary authorization embedded within GKE, based on the open source project **Kritis** (<https://github.com/grafeas/kritis>). It uses a **public key infrastructure (PKI)** to attest and verify images—so your images are signed by an attestor authority using the private key, and Kubernetes verifies the images by using the public key. The following diagram explains this beautifully:

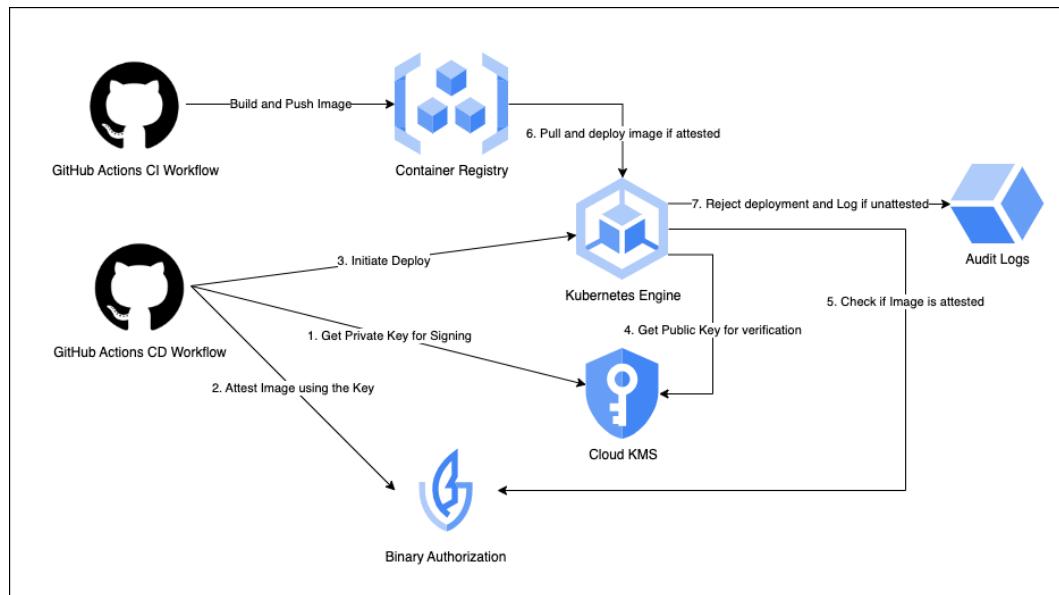


Figure 13.12 – Binary authorization process

In the hands-on exercise, we will set up binary authorization and a PKI using Google Cloud KMS. Next, we will create a QA attestor and an attestation policy for all binary auth-enabled GKE clusters, ensuring that only attested images can be deployed. Since our application is now tested, the next step is to attest the tested images. So, let's proceed to set up binary authorization within our Dev CD workflow in the next section.

Setting up binary authorization

As we're using GitOps right from the beginning, we will use Terraform to set up binary authorization for us. We'll start by setting up some GitHub Actions secrets. Go to https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions and create the following secrets:

```
ATTESTOR_NAME=quality-assurance-attestor
KMS_KEY_LOCATION=us-central1
KMS_KEYRING_NAME=qa-attestor-keyring
KMS_KEY_NAME=quality-assurance-attestor-key
KMS_KEY_VERSION=1
```

We'll then create a `binaryauth.tf` file with the following resources.

We'll begin by creating a Google KMS key ring. Since binary authorization utilizes PKI for creating and verifying attestations, this key ring will enable our attester to digitally sign attestations for images. Please note the `count` attribute defined in the following code. This ensures that it is created exclusively in the `dev` environment, where we intend to use the attester for attesting images after testing our app:

```
resource "google_kms_key_ring" "qa-attestor-keyring" {
  count = var.branch == "dev" ? 1 : 0
  name   = "qa-attestor-keyring"
  location = var.region
  lifecycle {
    prevent_destroy = false
  }
}
```

We will then use a Google-provided `binary-authorization` Terraform module to create our `quality-assurance` attester. That attester uses the Google KMS key ring we created before:

```
module "qa-attestor" {
  count = var.branch == "dev" ? 1 : 0
  source = "terraform-google-modules/kubernetes-engine/google//modules/binary-
authorization"
  attestor_name = "quality-assurance"
  project_id   = var.project_id
  keyring_id   = google_kms_key_ring.qa-attestor-keyring[0].id
}
```

Finally, we will create a binary authorization policy that specifies the cluster's behavior when deploying a container. In this scenario, our objective is to deploy only attested images. However, we will make a few exceptions, allowing Google-provided system images, Argo CD, and External Secrets Operator images. We will set the `global_policy_evaluation_mode` attribute to `ENABLE` to avoid enforcing the policy on system images managed by Google.

The admission_whitelist_patterns section defines container image patterns permitted to be deployed without attestations. This includes patterns for Google-managed system images, the Argo CD registry, the External Secrets registry, and the Redis container used by Argo CD.

The defaultAdmissionRule section mandates attestation using the attestor we created. Therefore, any other images would require attestation to run on the cluster:

```
resource "google_binary_authorization_policy" "policy" {
  count = var.branch == "dev" ? 1 : 0
  admission_whitelist_patterns {
    name_pattern = "gcr.io/google_containers/*"...
    name_pattern = "gcr.io/google-containers/*"...
    name_pattern = "k8s.gcr.io/**"...
    name_pattern = "gke.gcr.io/**"...
    name_pattern = "gcr.io/stackdriver-agents/*"...
    name_pattern = "quay.io/argoproj/*"...
    name_pattern = "ghcr.io/dexidp/*"...
    name_pattern = "docker.io/redis[@:]*"...
    name_pattern = "ghcr.io/external-secrets/*"
  }
  global_policy_evaluation_mode = "ENABLE"
  default_admission_rule {
    evaluation_mode = "REQUIRE_ATTESTATION"
    enforcement_mode = "ENFORCED_BLOCK_AND_AUDIT_LOG"
    require_attestations_by = [
      module.qa-attestor[0].attestor
    ]
  }
}
```

To enforce the binary authorization policy within a cluster, we must also enable binary authorization. To do so, we add the following block within the `cluster.tf` file:

```
resource "google_container_cluster" "main" {
  ...
  dynamic "binary_authorization" {
    for_each = var.branch == "prod" ? [1] : []
    content {
      evaluation_mode = "PROJECT_SINGLETON_POLICY_ENFORCE"
    }
  }
  ...
}
```

This dynamic block is created exclusively when the branch name is `prod`. The reason for this approach is our intention to deploy our code to the Dev environment without image attestation, conduct testing, and then attest the images if the tests succeed. Therefore, only the Prod cluster should disallow unattested images. To achieve this, we will include the following steps in the Dev CD workflow:

```
binary-auth:
  name: Attest Images
  needs: [run-tests]
  uses: ./github/workflows/attest-images.yml
  secrets: inherit
```

As you can see, this calls the `attest-images.yml` workflow. Let's look at that now:

```
...
steps:
- uses: actions/checkout@v2
- id: gcloud-auth ...
- name: Set up Cloud SDK ...
- name: Install gcloud beta
  id: install-gcloud-beta
  run: gcloud components install beta
- name: Attest Images
  run:
    for image in $(cat ./images); do
      no_of_slash=$(echo $image | tr -cd '/' | wc -c)
      prefix=""
      if [ $no_of_slash -eq 1 ]; then
        prefix="docker.io/"
      fi
      if [ $no_of_slash -eq 0 ]; then
        prefix="docker.io/library/"
      fi
      image_to_attest=$image
      if [[ $image =~ "@" ]]; then
        echo "Image $image has DIGEST"
        image_to_attest="${prefix}${image}"
      else
        echo "All images should be in the SHA256 digest format"
        exit 1
      fi
      echo "Processing $image"
      attestation_present=$(gcloud beta container binauthz attestations list
--attestor-project="${{ secrets.PROJECT_ID }}" --attestor="${{ secrets.ATTESTOR_NAME }}"
--artifact-url="${{image_to_attest}}")
      if [ -z "${{attestation_present// }}"]; then
        gcloud beta container binauthz attestations sign-and-create --artifact-
url="${{image_to_attest}}" --attestor="${{ secrets.ATTESTOR_NAME }}" --attestor-project="${
secrets.PROJECT_ID }" --keyversion-project="${{ secrets.PROJECT_ID }}" --keyversion-
location="${{ secrets.KMS_KEY_LOCATION }}" --keyversion-keyring="${{ secrets.KMS_KEYRING_-
NAME }}" --keyversion-key="${{ secrets.KMS_KEY_NAME }}" --keyversion="${{ secrets.KMS_KEY_-
VERSION }}"
```

```
    fi
done
```

The YAML file performs several tasks, including the installation of `gcloud` and authentication with GCP. It also installs the `gcloud beta` CLI and, importantly, attests images.

To attest images, it searches the `blog-app.yaml` manifest for all images. For each image, it checks whether the image is in the sha256 digest format. If yes, it proceeds to attest the image.

It's worth noting that the workflow verifies that images are specified using a sha256 digest format rather than a tag in the image definition. This choice is crucial when working with binary authorization. Why? Because binary authorization requires deploying images with their sha256 digest instead of a tag. This precaution is essential because, with tags, anyone can associate a different image with the same tag as the attested image and push it to the container registry. In contrast, a digest is a hash generated from a Docker image. Therefore, as long as the image's content remains unchanged, the digest remains the same. This prevents any attempts to bypass binary authorization controls.

The format for specifying images in this manner is as follows:

```
<repo_url>/<image_name>@sha256:<sha256-digest>
```

Therefore, before pushing the changes to the remote repository, let's replace the image tags with sha256 digests. Use the following commands to do so:

```
$ grep -ir "image:" ./manifests/blog-app | \
awk {'print $3'} | sort -t: -u -k1,1 > ./images
$ for image in $(cat ./images); do
  no_of_slash=$(echo $image | tr -cd '/' | wc -c)
  prefix=""
  if [ $no_of_slash -eq 1 ]; then
    prefix="docker.io/"
  fi
  if [ $no_of_slash -eq 0 ]; then
    prefix="docker.io/library/"
  fi
  image_to_attest=$image
  if [[ $image =~ "@" ]]; then
    echo "Image $image has DIGEST"
    image_to_attest="${prefix}${image}"
  else
    DIGEST=$(docker pull $image | grep Digest | awk {'print $2'})
    image_name=$(echo $image | awk -F ':' {'print $1'})
    image_to_attest="${prefix}${image_name}@${DIGEST}"
  fi
  escaped_image=$(printf '%s\n' "${image}" | sed -e 's/[]\/$/\\&/g')
  escaped_image_to_attest=$(printf '%s\n' "${image_to_attest}" | \
  sed -e 's/[]\/$/\\&/g')
  echo "Processing $image"
```

```
grep -rl $image ./manifests | \
xargs sed -i "s/${escaped_image}/${escaped_image_to_attest}/g"
done
```

To verify whether the changes were successful, run the following command:

```
$ cat manifests/blog-app/blog-app.yaml | grep "image:"
image: docker.io/library/mongo@sha256:2a1093b275d9bc...
image: docker.io/bharamicrosystems/ndo-posts@sha256:b5bc...
image: docker.io/bharamicrosystems/ndo-reviews@sha256:073...
image: docker.io/bharamicrosystems/ndo-ratings@sha256:271...
image: docker.io/bharamicrosystems/ndo-users@sha256:5f5a...
image: docker.io/bharamicrosystems/ndo-frontend@sha256:87...
```

As we can see, the images have been updated. Now, let's proceed to push the changes to the remote repository using the following commands:

```
$ cp ~/modern-devops/ch13/binaryauth/binaryauth.tf terraform/
$ cp ~/modern-devops/ch13/binaryauth/cluster.tf terraform/
$ cp ~/modern-devops/ch13/binaryauth/variables.tf terraform/
$ cp -r ~/modern-devops/ch13/binaryauth/.github .
$ git add --all
$ git commit -m "Enabled Binary Auth"
$ git push
```

Now, let's review the Dev CD workflow on GitHub Actions, where we should observe the following:

The screenshot shows the GitHub Actions interface for a workflow named 'Dev Continuous Delivery Workflow'. The main view displays a summary of completed jobs: 'Create Environment and Deploy ...' (green checkmark), 'Run Integration Tests' (green checkmark), and 'Attest Images' (green checkmark). The 'Attest Images' job is expanded, showing its sub-tasks: 'binary-auth' (green checkmark), 'Set up job' (green checkmark, 1s), 'Run actions/checkout@v2' (green checkmark, 1s), 'Authenticate with gcloud' (green checkmark, 0s), 'Set up Cloud SDK' (green checkmark, 18s), 'Install gcloud beta' (green checkmark, 28s), 'Attest Images' (green checkmark, 27s), 'Post Authenticate with gcloud' (green checkmark, 0s), 'Post Run actions/checkout@v2' (green checkmark, 0s), and 'Complete job' (green checkmark, 0s). A 'Re-run all jobs' button and a three-dot menu are visible in the top right corner.

Figure 13.13 – Dev CD workflow – Attest Images

As is evident, the workflow has successfully configured binary authorization and attested our images. To verify, execute the following command:

```
$ gcloud beta container binauthz attestations list \
--attestor-project="$PROJECT_ID" \
--attestor="quality-assurance-attestor" | grep resourceUri
resourceUri: docker.io/bharamicrosystems/mdo-ratings@
sha256:271981faefafb86c2d30f7d3ce39cd8b977b7dd07...
resourceUri: docker.io/library/mongo@sha256:2a1093b275d9bc546135ec2e2...
resourceUri: docker.io/bharamicrosystems/mdo-posts@
sha256:b5bc1fc976a93a88cc312d24916bd1423dbb3efe25e...
resourceUri: docker.io/bharamicrosystems/mdo-frontend@
sha256:873526fe6de10e04c42566bbaa47b76c18f265fd...
resourceUri: docker.io/bharamicrosystems/mdo-users@
sha256:5f5aa595bc03c53b86dadf39c928eff4b3f05533239...
resourceUri: docker.io/bharamicrosystems/mdo-reviews@
sha256:07370e90859000ff809b1cd1fd2fc45a14c5ad46e...
```

As we can see, the attestations have been successfully created. Having deployed our application in the Dev environment, tested it, and attested all the images within, we can now proceed with deploying the code to the Prod environment. This involves merging our code with the `prod` branch, and we will implement pull request gating for this purpose.

Release gating with pull requests and deployment to production

The process of pull request gating is straightforward. At the end of the Dev CD workflow, we'll introduce a step to initiate a pull request to merge `dev` into the `prod` branch. Human approval is required to proceed with merging the pull request. This step highlights how various organizations may adopt different methods to verify and promote tested code. Some may opt for automated merging, while others may prioritize human-triggered actions. Once the code is successfully merged into the `prod` branch, it triggers the Prod CD workflow. This workflow creates the Prod environment and deploys our application. It also executes the same integration test we ran in the Dev environment to ensure the deployed application in Prod remains intact.

Here's the step we'll add to the Dev CD workflow:

```
raise-pull-request:
  name: Raise Pull Request
  needs: [binary-auth]
  uses: ./github/workflows/raise-pr.yml
  secrets: inherit
```

As we can see, this step invokes the `raise-pr.yml` file. Let's look at that:

```
...
steps:
  - uses: actions/checkout@v2
  - name: Raise a Pull Request
    id: pull-request
    uses: repo-sync/pull-request@v2
    with:
      destination_branch: prod
      github_token: ${{ secrets.GH_TOKEN }}
```

This workflow does the following:

- Checks out the code from the repository
- Raises a pull request to merge with the `prod` branch using the `GH_TOKEN` secret

To enable the workflow's functionality, we need to define a GitHub token. This token allows the workflow to act on behalf of the current user when creating the pull request. Here are the steps:

1. Go to <https://github.com/settings/personal-access-tokens/new>.
2. Create a new token with **Repository** access for the `mdo-environments` repository, granting it the `read-write` pull request permission. This approach aligns with the principle of least privilege, offering more granular control.
3. Once the token is created, copy it.
4. Now, create a GitHub Actions secret named `GH_TOKEN` and paste the copied token as the value. You can do this by visiting https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions.

Next, let's proceed to copy the workflow files using the following commands:

```
$ cd ~/mdo-environments/.github/workflows
$ cp ~/modern-devops/ch13/raise-pr/.github/workflows/dev-cd-workflow.yml .
$ cp ~/modern-devops/ch13/raise-pr/.github/workflows/raise-pr.yml .
```

We're ready to push this code to GitHub. Run the following commands to commit and push the changes to your GitHub repository:

```
$ git add --all
$ git commit -m "Added PR Gating"
$ git push
```

This should trigger a GitHub Actions workflow in your GitHub repository, and you should observe something similar to the following:

```
Raise Pull Request / raise-pull-request
succeeded 1 minute ago in 12s

> ⚡ Set up job
1s

> ⚡ Build repo-sync/pull-request@v2
5s

> ⚡ Run actions/checkout@v2
1s

> ⚡ Raise a Pull Request
2s

1 ► Run repo-sync/pull-request@v2
5 /usr/bin/docker run --name b6b489a7edf0e41b0bf618254061e36_b668ee --label 9435ib --workdir /github/workspace --rm -e "INPUT_DESTINATION_BRANCH" -e "INPUT_GITHUB_TOKEN" -e "INPUT_DESTINATION_REPOSITORY" -e "INPUT_SOURCE_BRANCH" -e "INPUT_PR_TITLE" -e "INPUT_PR_BODY" -e "INPUT_PR_TEMPLATE" -e "INPUT_PR_REVIEWER" -e "INPUT_PR_ASSIGNEE" -e "INPUT_PR_LABEL" -e "INPUT_PR_MILESTONE" -e "INPUT_PR_DRAFT" -e "INPUT_PR_ALLOW_EMPTY" -e "INPUT_DEBUG" -e "GITHUB_TOKEN" -e "HOME" -e "GITHUB_JOB" -e "GITHUB_REF" -e "GITHUB_SHA" -e "GITHUB_REPOSITORY" -e "GITHUB_REPOSITORY_OWNER" -e "GITHUB_REPOSITORY_OWNER_ID" -e "GITHUB_RUN_ID" -e "GITHUB_RUN_NUMBER" -e "GITHUB_RETENTION_DAYS" -e "GITHUB_RUN_ATTEMPT" -e "GITHUB_ACTOR_ID" -e "GITHUB_ACTOR" -e "GITHUB_TRIGGERING_ACTOR" -e "GITHUB_WORKFLOW" -e "GITHUB_HEAD_REF" -e "GITHUB_BASE_REF" -e "GITHUB_EVENT_NAME" -e "GITHUB_SERVER_URL" -e "GITHUB_API_URL" -e "GITHUB_REF_NAME" -e "GITHUB_REF_PROTECTED" -e "GITHUB_REF_TYPE" -e "GITHUB_WORKFLOW_REF" -e "GITHUB_WORKFLOW_SHA" -e "GITHUB_WORKSPACE" -e "GITHUB_ACTION" -e "GITHUB_EVENT_PATH" -e "GITHUB_ACTION_REPOSITORY" -e "GITHUB_ACTION_REF" -e "GITHUB_PATH" -e "GITHUB_ENV" -e "GITHUB_STEP_SUMMARY" -e "GITHUB_STATE" -e "GITHUB_OUTPUT" -e "RUNNER_OS" -e "RUNNER_ARCH" -e "RUNNER_NAME" -e "RUNNER_ENVIRONMENT" -e "RUNNER_TOOL_CACHE" -e "RUNNER_TEMP" -e "RUNNER_WORKSPACE" -e "ACTIONS_RUNTIME_URL" -e "ACTIONS_RUNTIME_TOKEN" -e "ACTIONS_CACHE_URL"
"RUNNER_ENVIRONMENT" -e "CI=true -v "/var/run/docker.sock":"/var/run/docker.sock" -v "/home/runner/work/_temp/_github_home":"/github/home" -v "/home/runner/work/_temp/_github_workflow":"/github/workflow" -v "/home/runner/work/_temp/_runner_file_commands":"/github/file_commands" -v "/home/runner/work/mdo-environments/mdo-environments":"/github/workspace" 9435ib:b6b489a7edf0e41b0bf618254061e36
6 ► Gather Inputs
13 ► Configure git
21 ► Ensure pull-request contains differences
22 ► Assemble hub pr parameters
24 ► Create pull request dev -> prod
27 ► Retrieving pull request details
30 ► Set outputs

> ⚡ Post Run actions/checkout@v2
0s
```

Figure 13.14 – Raising a pull request

GitHub has generated a pull request to merge the code into the `prod` branch, and the Dev CD workflow is running as anticipated. We can now review the pull request and merge the code into the `prod` branch.

Merging code and deploying to prod

As demonstrated in the previous section, the Dev CD workflow created our environment, deployed the application, tested it, and attested application images. It then automatically initiated a pull request to merge the code into the `prod` branch.

We've entered the **release gating phase**, where we require manual verification to determine whether the code is ready for merging into the `prod` branch.

Since we know the pull request has been created, let's proceed to inspect and approve it. To do so, go to https://github.com/<your_github_user>/mdo-environments/pulls, where you will find the pull request. Click on the pull request, and you will encounter the following:

Added PR Gating #2

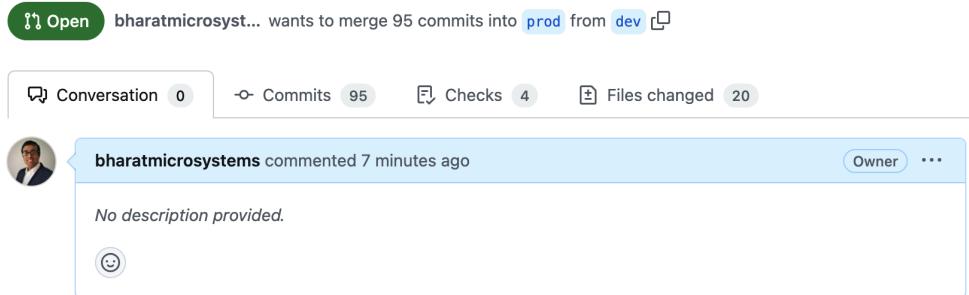


Figure 13.15 – Pull request

We see that the pull request is ready to merge. Click on **Merge pull request**, and you will see that the changes will reflect on the `prod` branch.

If you go to `https://github.com/<your_user>/mdo-environments/actions`, you'll find that the Prod CD workflow has been triggered. When you click on the workflow, you will see a workflow run like the following:



Figure 13.16 – Prod CD workflow

When we merged the pull request, it automatically triggered the Prod CD workflow as it would react to any new changes in the `prod` branch. The workflow did its job by building the Prod environment, deploying our application, and testing it. Note that binary authorization is enabled for this cluster.

To confirm that binary authorization is functioning correctly, let's perform some checks to ensure unattested images cannot be deployed.

First, let's establish a connection to the prod cluster using the following command:

```
$ gcloud container clusters get-credentials \
  mdo-cluster-prod --zone us-central1-a --project ${PROJECT_ID}
```

Let's attempt to deploy a pod to your cluster using an nginx image. Please use the following command:

```
$ kubectl run nginx --image=nginx
Error from server (VIOLATES_POLICY): admission webhook "imagepolicywebhook.image-policy.k8s.io" denied the request: Image nginx denied by Binary Authorization default admission rule. Image nginx denied by attester projects/<PROJECT_ID>/attestors/quality-assurance-attester: Expected digest with sha256 scheme, but got tag or malformed digest
```

Now, as expected, the deployment failed, but there's something else to note if you examine the reason. The failure happened because we specified a tag instead of a sha256 digest. Let's attempt to deploy the image again, but this time, with a digest.

To do so, let's retrieve the image digest and set it as a variable called DIGEST using the following command:

```
$ DIGEST=$(docker pull nginx | grep Digest | awk {'print $2'})
```

Now, let's redeploy the image using the digest with the following command:

```
$ kubectl run nginx --image=nginx@$DIGEST
Error from server (VIOLATES_POLICY): admission webhook "imagepolicywebhook.image-policy.k8s.io" denied the request: Image nginx@sha256:6926dd8... denied by Binary Authorization default admission rule. Image nginx@sha256:6926dd8... denied by attester projects/<PROJECT_ID>/attestors/quality-assurance-attester: No attestations found that were valid and signed by a key trusted by the attester
```

This time, the deployment was denied for a valid reason, confirming that binary authorization functions correctly. This ensures the security of your Kubernetes cluster, preventing the deployment of unattested images and giving you complete control over your environment. With this in place, any issues that arise won't stem from deploying untested or vulnerable images.

We've covered a lot of ground in integrating security and QA into our CI/CD pipelines. Now, let's explore some best practices for securing modern DevOps pipelines.

Security and testing best practices for modern DevOps pipelines

Tooling is not the only thing that will help you in your DevSecOps journey. Here are some helpful tips that can help you address security risks and have a more secure culture within your organization.

Adopt a DevSecOps culture

Adopting a DevSecOps approach is critical in implementing modern DevOps. Therefore, it is vital to embed security within an organization's culture. You can achieve that by implementing effective communication and collaboration between the *development*, *operations*, and *security* teams. While most organizations have a security policy, it mustn't be followed just to comply with rules and regulations. Instead, employees should cross-skill and upskill themselves to adopt a DevSecOps approach and embed security early on during development. Security teams need to learn how to write code and work with APIs, while developers need to understand security and use automation to achieve this.

Establish access control

You have heard about the **Principle of Least Privilege (PoLP)** several times in this book. Well, that is what you need to implement for a better security posture, which means you should make all attempts to grant only the required privileges to people to do their job, and nothing more. Reduce the just-in-case syndrome by making the process of giving access easier so that people don't feel hindered, and as a result, they do not seek more privileges than they require.

Implement shift left

Shifting left means embedding security into software at the earlier stages of software development. This means security experts need to work closely with developers to enable them to build secure software right from the start. The security function should not be review-only but should actively work with developers and architects to develop a security-hardened design and code.

Manage security risks consistently

You should accept risks, which are inevitable, and should have a **Standard Operating Procedure (SOP)** should an attack occur. You should have straightforward and easy-to-understand policies and practices from a security standpoint in all aspects of software development and infrastructure management, such as **configuration management**, **access controls**, **vulnerability testing**, **code review**, and **firewalls**.

Implement vulnerability scanning

Open source software today is snowballing, and most software implementations rely on ready-made open source frameworks, software libraries, and third-party software that don't come with a guarantee or liability of any kind. While the open source ecosystem is building the technological world like never before, it does have its own share of vulnerabilities, which you don't want to insert within your software through no fault of your own. Vulnerability scanning is crucial, as scans can discover any third-party dependency with vulnerabilities and alert you at the initial stage.

Automate security

Security should not hinder the speed of your DevOps teams; therefore, to keep up with the fast pace of DevOps, you should look at embedding security within your CI/CD processes. You can do code analysis, vulnerability scanning, configuration management, and infrastructure scanning with policy as code and binary authorization to allow only tested and secure software to be deployed. Automation helps identify potential vulnerabilities early on in the software development life cycle, thereby bringing down the cost of software development and rework.

Similarly, QA is the backbone of software delivery, and modern DevSecOps heavily emphasizes automating it. Here are some tips you can follow to implement a modern testing approach.

Test automation within your CI/CD pipelines

Automating testing across the board is key. This means encompassing a wide spectrum, from unit and integration testing to functional, security, and performance testing. The goal is to seamlessly embed these tests within your CI/CD pipeline, ensuring a constant stream of validation. In this journey, creating isolated and reproducible test environments becomes crucial to thwart any interference among tests. Here, methods such as containerization and virtualization are valuable tools for environment isolation.

Manage your test data effectively

Test data management is another pivotal aspect. It's imperative to handle your test data effectively, not only ensuring its consistency but also safeguarding data privacy. Leveraging data generation tools can be a game-changer in this regard, allowing you to create relevant datasets for your testing needs. Moreover, when dealing with sensitive information, the consideration of data anonymization is prudent. This ensures that you maintain the highest standards of data protection while still benefiting from comprehensive testing procedures.

Test all aspects of your application

CI is all about keeping the development process flowing smoothly. This involves frequently merging code and running tests automatically, ensuring the code base remains stable. When tests fail, immediate attention is crucial to rectify the issues promptly.

End-to-end testing is your compass to ensure the entire application workflow functions as expected. Automation frameworks play a pivotal role in replicating real user interactions, making it possible to assess your application thoroughly.

Load testing is an essential part of the process, as it evaluates how your application performs under varying loads, providing insights into its robustness and capacity. Additionally, scalability testing ensures that the system is well-equipped to handle growth, an important factor for the long-term health of your application.

Implement chaos engineering

Incorporating chaos engineering practices is a proactive strategy to uncover and address potential system weaknesses. By conducting controlled experiments, you can gauge the resilience of your system and better prepare it for unexpected challenges. These experiments involve intentionally introducing chaos into your environment to observe how your system responds. This not only helps you identify weaknesses but also provides valuable insights into how to make your system more robust and reliable.

Monitor and observe your application when it is being tested

Setting up robust monitoring and observability tools is crucial for gaining deep insights into your system's performance and behavior. These tools allow you to collect essential metrics, logs, and traces, providing a comprehensive view of your application's health and performance.

Effective testing in production

Implementing feature flags and canary releases is a prudent strategy for testing new functionality in a real production environment while minimizing risks. Feature flags allow you to enable or disable certain features at runtime, giving you control over their activation. Canary releases involve rolling out new features to a small subset of users, allowing you to monitor their impact before a full-scale release.

By utilizing feature flags, you can introduce new features to a limited audience without affecting the entire user base. This controlled approach lets you observe user interactions, collect feedback, and assess the feature's performance in a real-world scenario. Simultaneously, canary releases enable you to deploy these features to a small, representative group of users, allowing you to monitor their behavior, collect performance metrics, and identify potential issues.

Crucially, continuous monitoring is essential during this process. By closely observing the impact of the new functionality, you can quickly detect any issues that may arise. If problems occur, you have the flexibility to roll back the changes by simply turning off the feature flags or reverting to the previous version. This iterative and cautious approach minimizes the impact of potential problems, ensuring a smoother user experience and maintaining the stability of your production environment.

Documentation and knowledge sharing

Documenting testing procedures, test cases, and best practices is essential for ensuring consistency and reliability within the development and testing processes. Comprehensive documentation serves as a reference for team members, providing clear guidelines on how to conduct tests, the expected outcomes, and the best practices to follow. This documentation acts as a valuable resource for both new and existing team members, fostering a shared understanding of the testing procedures.

Encouraging knowledge sharing among team members further enhances the collective expertise of the team. By promoting open communication and sharing experiences, team members can learn from one another, gain insights into different testing scenarios, and discover innovative solutions to

common challenges. This collaborative environment promotes continuous learning and ensures that the team stays updated on the latest developments and techniques in the field of software testing.

By adhering to these best practices, teams can enhance the security and reliability of their CI/CD pipelines. Properly documented procedures and test cases enable consistent testing, reducing the likelihood of introducing errors into the code base. Knowledge sharing ensures that the team benefits from the collective wisdom and experiences of its members, leading to more informed decision-making and efficient problem-solving.

In addition, managing security risks effectively becomes possible through well-documented testing procedures and disseminating best practices. Teams can identify potential security vulnerabilities early in the development process, enabling them to address these issues before they escalate into significant threats. Regular knowledge-sharing sessions can also include discussions about security best practices, ensuring that team members are aware of the latest security threats and countermeasures.

Ultimately, these best practices contribute to a robust testing and development culture. They empower teams to deliver software faster and with confidence, knowing that their CI/CD pipelines are secure, reliable, and capable of handling the challenges of modern software development.

Summary

This chapter has covered CI/CD pipeline security and testing, and we have understood various tools, techniques, and best practices surrounding it. We looked at a secure CI/CD workflow for reference. We then understood, using hands-on exercises, the aspects that made it secure, such as secret management, container vulnerability scanning, and binary authorization.

Using the skills learned in this chapter, you can now appropriately secure your CI/CD pipelines and make your application more secure.

In the next chapter, we will explore the operational elements along with key performance indicators for running our application in production.

Questions

1. Which of these is the recommended place for storing secrets?
 - A. Private Git repository
 - B. Public Git repository
 - C. Docker image
 - D. Secret management system

2. Which one of the following is an open source secret management system?
 - A. Secret Manager
 - B. HashiCorp Vault
 - C. Anchore Grype
3. Is it a good practice to download a secret within your CD pipeline's filesystem?
4. Which base image is generally considered more secure and consists of the fewest vulnerabilities?
 - A. Alpine
 - B. Slim
 - C. Buster
 - D. Default
5. Which of the following answers are true about binary authorization? (Choose two)
 - A. It scans your images for vulnerabilities.
 - B. It allows only attested images to be deployed.
 - C. It prevents people from bypassing your CI/CD pipeline.

Answers

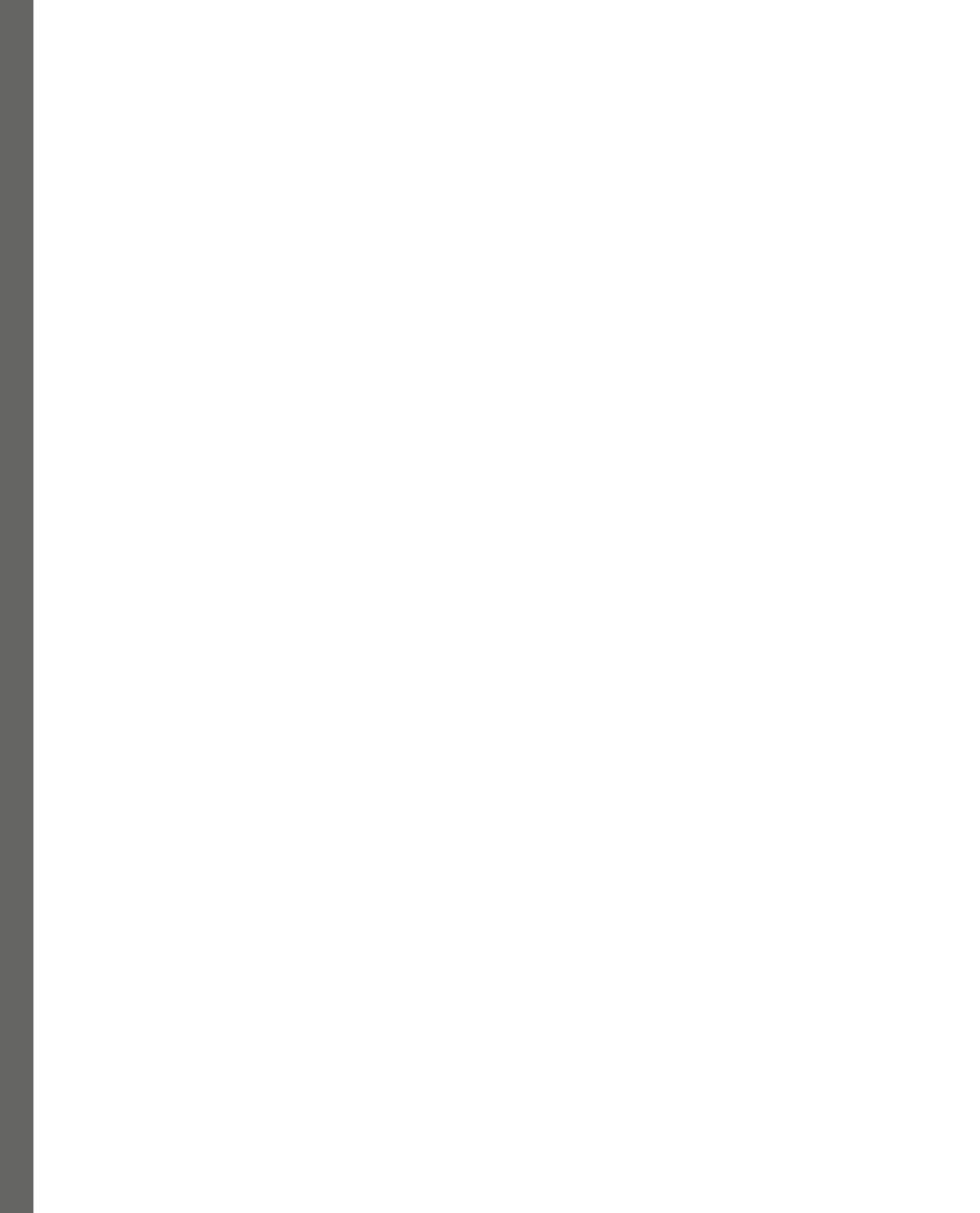
1. D
2. B
3. No
4. A
5. B and C

Part 5: Operating Applications in Production

This part provides a comprehensive guide to managing containers in production. We will start by covering key performance indicators and reliability principles and then explore Istio for advanced security, traffic management, and observability. This section will equip you with crucial skills to optimize container-based applications in production.

This part has the following chapters:

- *Chapter 14, Understanding Key Performance Indicators (KPIs) for Your Production Service*
- *Chapter 15, Operating Containers in Production with Istio*



14

Understanding Key Performance Indicators (KPIs) for Your Production Service

In the previous chapters, we looked at the core concepts of modern DevOps – **Continuous Integration (CI)** and **Continuous Deployment/Delivery (CD)**. We also looked at various tools and techniques that can help us enable a mature and secure DevOps channel across our organization. In this rather theory-focused chapter, we'll try to understand some **key performance indicators (KPIs)** for operating our application in production.

In this chapter, we're going to cover the following main topics:

- Understanding the importance of reliability
- SLOs, SLAs, and SLIs
- Error budgets
- Recovery Time Objective (RPO) and Recovery Point Objective (RTO)
- Running distributed applications in production

So, let's get started!

Understanding the importance of reliability

Developing software is one thing, and running it in production is another. The reason behind such a disparity is that most development teams cannot simulate production conditions in non-production environments. Therefore, many bugs are uncovered when the software is already running in production. Most issues encountered are non-functional – for example, the services could not scale properly with additional traffic, the amount of resources assigned to the application was suboptimal, thereby crashing the site, and many more. These issues need to be managed to make the software more reliable.

To understand the importance of software reliability, let's look at an example retail banking application. Software reliability is critically important for several reasons:

- **User satisfaction:** Reliable software ensures a positive user experience. Users expect software to work as intended, and when it doesn't, it can lead to frustration, loss of trust, and a poor reputation for the software or the organization behind it. For a bank's retail customer, it might mean customers cannot do essential transactions and, therefore, may face hassles in payments and receipts, leading to a loss in user satisfaction.
- **Business reputation:** Software failures can tarnish a company's reputation and brand image. For our bank, if the issues are frequent, customers will look for other options, resulting in considerable churn and loss of business.
- **Financial impact:** Software failures can be costly. They can result in lost sales, customer support expenses, and even legal liabilities in cases where software failures cause harm or financial losses to users. This becomes especially critical for banking applications as customers' money is involved. If transactions don't happen in time, it can result in a loss of customer business, which will hurt the bank in the long run.
- **Competitive advantage:** Reliable software can provide a competitive edge. Users are more likely to choose and stick with a bank with robust online banking software that consistently meets their needs and expectations.
- **Productivity and efficiency:** Within organizations, reliable software is essential for maintaining productivity. Imagine the pain that the customer support and front office staff would have in such a disruption! You would also need more resources to manage these issues, which can disrupt operations, leading to wasted time and resources.
- **Security:** Reliable software is often more secure. Attackers can exploit vulnerabilities and bugs in unreliable software. In the case of a bank, security is of prime importance because any breach can result in direct financial impact and loss. Ensuring reliability is a fundamental part of cybersecurity.
- **Compliance:** In some industries, especially banking, there are regulatory requirements related to software reliability. Failing to meet these requirements can result in legal and financial penalties.
- **Customer trust:** Trust is a critical factor in software usage, especially in the case of a banking application. Users must trust that their money and data will be handled securely and that the software will perform as expected. Software reliability is a key factor in building and maintaining this trust.
- **Maintainability:** Reliable software is typically easier to maintain. When software is unreliable, fixing bugs and updating becomes more challenging, potentially leading to a downward spiral of increasing unreliability.

- **Scaling and growth:** As software usage grows, reliability becomes even more critical. Software that works well for a small user base may struggle to meet the demands of a larger user base without proper reliability measures in place.

In summary, software reliability is not just a technical concern; it has wide-reaching implications for user satisfaction, business success, and even legal and financial aspects. Therefore, investing in ensuring the reliability of software is a prudent and strategic decision for organizations.

Historically, running and managing software in production was the job of the Ops team, and most organizations still use it. The Ops team comprises a bunch of **system administrators (SysAdmins)** who must deal with the day-to-day issues of running the software in production. They implement scaling and fault tolerance with software, patch and upgrade software, work on support tickets, and keep the systems running so the software application functions well.

We've all experienced the divide between Dev and Ops teams, each with its own goals, rules, and priorities. Often, they found themselves at odds because what benefited Dev (software changes and rapid releases) created challenges for Ops (stability and reliability).

However, the emergence of DevOps has changed this dynamic. In the words of Andrew Shafer and Patrick Debois, DevOps is a culture and practice in software engineering aimed at bridging the gap between software development and operations.

Looking at DevOps from an Ops perspective, Google came up with **site reliability engineering (SRE)** as an approach that embodies DevOps principles. It encourages shared ownership, the use of common tools and practices, and a commitment to learning from failures to prevent recurring issues. The primary objective is to develop and maintain a dependable application without sacrificing the speed of delivery – a balance that was once thought contradictory (that is, *create better software faster*).

The idea of SRE is a novel thought about what would happen if we allowed software engineers to run the production environment. So, Google devised the following approach for running its Ops team.

For Google, an ideal candidate for joining the SRE team should exhibit two key characteristics:

- Firstly, they quickly become disinterested in manual tasks and seek opportunities to automate them
- Secondly, they possess the requisite skills to develop software solutions, even when faced with complex challenges

Additionally, SREs should share an academic and intellectual background with the broader development organization. Essentially, SRE work, traditionally within the purview of operations teams, is carried out by engineers with strong software expertise. This strategy hinges on the natural inclination and capability of these engineers to design and implement automation solutions, thus reducing reliance on manual labor.

By design, SRE teams maintain a strong engineering focus. Without continuous engineering efforts, the operational workload escalates, necessitating an expansion of the team to manage the increasing demands. In contrast, a conventional operations-centric group scales in direct proportion to the growth of the service. If the services they support thrive, operational demands surge with increased traffic, compelling the hiring of additional personnel to perform repetitive tasks.

To avert this scenario, the team responsible for service management must incorporate coding into their responsibilities; otherwise, they risk becoming overwhelmed.

Accordingly, Google establishes a 50% upper limit on the aggregate “Ops” work allocated to all SREs, encompassing activities such as handling tickets, on-call duties, and manual tasks. This constraint guarantees that SRE teams allocate a substantial portion of their schedules to enhancing the stability and functionality of the service. While this limit serves as an upper bound, the ideal outcome is that, over time, SREs carry minimal operational loads and primarily engage in development endeavors as the service evolves to a self-sustaining state. Google’s objective is to create systems that are not merely automated but inherently self-regulating. However, practical considerations such as scaling and introducing new features continually challenge SREs.

SREs are meticulous in their approach, relying on measurable metrics to track progress toward specific goals. For instance, stating that a website is *running slowly* is vague and unhelpful in an engineering context. However, declaring that the 95th percentile of response time has exceeded the **service-level objective (SLO)** by 10% provides precise information. SREs also focus on reducing repetitive tasks, known as **toil**, by automating them to prevent burnout. Now, let’s look at some of the key SRE performance indicators.

Understanding SLIs, SLOs, and SLAs

In the realm of site reliability, three crucial parameters guide SREs: the **indicators of availability – service-level indicators (SLIs)**, the **definition of availability –SLOs**, and the **consequences of unavailability – service-level agreements (SLAs)**. Let’s start by exploring SLIs in detail.

SLIs

SLIs serve as quantifiable reliability metrics. Google defines them as “*carefully defined quantitative measures of some aspect of the level of service provided.*” Common examples include request latency, failure rate, and data throughput. SLIs are specific to user journeys, which are sequences of actions users perform to achieve specific goals. For instance, a user journey for our sample Blog App might involve creating a new blog post.

Google, the original advocate of SRE, has identified four golden signals that apply to most user journeys:

- **Latency:** This measures the time it takes for your service to respond to user requests
- **Errors:** This indicates the percentage of failed requests, highlighting issues in service reliability

- **Traffic:** Traffic represents the demand directed toward your service, reflecting its usage
- **Saturation:** Saturation assesses how fully your infrastructure components are utilized

One recommended approach by Google to calculate SLIs is by determining the ratio of good events to valid events:

$$\text{SLI} = (\text{Good Events} * 100) / \text{Valid Events}$$

A perfect SLI score of 100 implies everything functions correctly, while a score of 0 signifies widespread issues.

A valuable SLI should align closely with the user experience. For example, a lower SLI value should correspond to decreased customer satisfaction. If this alignment is absent, the SLI may not provide meaningful insights or be worth measuring.

Let's look at the following figure to understand this better:

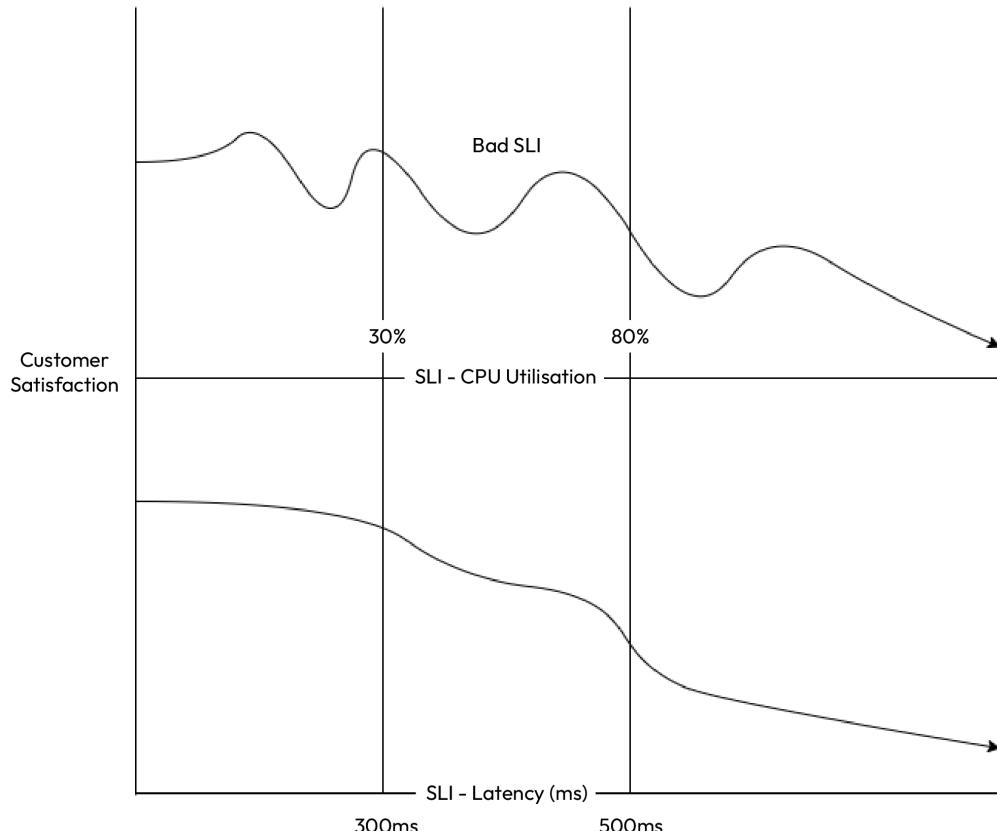


Figure 14.1 – Good versus bad SLI

As we can see, the CPU Utilisation SLI does not reflect customer satisfaction in any way; that is, there is no direct correlation between increasing CPU Utilisation and decreased customer satisfaction except after it crosses the 80% threshold. In contrast, the Latency SLI directly correlates with customer satisfaction, which reduces with increasing latency and significantly after the 300ms and 500ms levels. Therefore, it is a good idea to use Latency as an SLI over CPU Utilization.

It's also advisable to limit the number of SLIs to a manageable quantity. Too many SLIs can lead to team confusion and trigger numerous false alarms. It's best to focus on four or five metrics directly linked to customer satisfaction. For instance, instead of monitoring CPU and memory usage, prioritize metrics such as request latency and error rate.

Furthermore, prioritizing user journeys is essential, giving higher importance to journeys that significantly impact customers and lower importance to those with less of a customer impact. For example, ensuring a seamless create and update post experience in our Blog App is more critical than the reviews and ratings service. SLIs alone do not make much sense as they are just measurable indicators. We need to set objectives for SLIs. So, let's look at SLOs.

SLOs

Google's definition of SLOs states that they "*establish a target level for the reliability of your service.*" They specify the percentage of compliance with SLIs required to consider your site reliable. SLOs are formulated by combining one or more SLIs.

For instance, if you have an SLI that mandates *request latency to remain below 500ms within the last 15 minutes with a 95th percentile measurement*, an SLO would necessitate *the SLI to be met 99% of the time for a 99% SLO*.

While every organization aims for 100% reliability, setting a 100% SLO is not a practical goal. A system with a 100% SLO tends to be costly, technically complex, and often unnecessary for most applications to be deemed acceptable by their users.

In the realm of software services and systems, the pursuit of 100% availability is generally misguided because users cannot feel any practical distinction between a system that is 100% available and one that is 99.999% available. Multiple intermediary systems exist between the user and the service, such as their personal computer, home Wi-Fi, **Internet Service Provider (ISP)**, and the power grid, and these collectively exhibit availability far lower than 99.999%. Consequently, the negligible difference between 99.999% and 100% availability becomes indistinguishable amidst the background noise of other sources of unavailability. Thus, investing substantial effort to attain that last 0.001% availability yields no noticeable benefit to the end user.

In light of this understanding, a question arises: if 100% is an inappropriate reliability target, what constitutes the right reliability target for a system? Interestingly, this is not a technical inquiry but rather a product-related one, necessitating consideration of the following factors:

- **User satisfaction:** Determining the level of availability that aligns with user contentment, considering their typical usage patterns and expectations

- **Alternatives:** Evaluating the availability of alternatives available to dissatisfied users, should they seek alternatives due to dissatisfaction with the product's current level of availability
- **User behavior:** Examining how users' utilization of the product varies at different availability levels, recognizing that user behavior may change in response to fluctuations in availability

Moreover, a completely reliable application leaves no room for the introduction of new features, as any new addition has the potential to disrupt the existing service. Therefore, some margin for error must be built into your SLO.

SLOs represent internal objectives that require consensus among the team and internal stakeholders, including developers, product managers, SREs, and CTOs. They necessitate the commitment of the entire organization. Not meeting an SLO does not carry explicit or implicit penalties.

For example, a customer cannot claim damages if an SLO is not met, but it may lead to dissatisfaction within organizational leadership. This does not imply that failing to meet an SLO should be consequence-free. Falling short of an SLO typically results in fewer changes and reduced feature development, potentially indicating a decline in quality and increased emphasis on the development and testing functions.

SLOs should be realistic, with the team actively working to meet them. They should align with the customer experience, ensuring that if the service complies with the SLO, customers do not perceive any service quality issues. If performance falls below the defined SLOs, it may affect the customer experience, but not to the extent that customers raise support tickets.

Some organizations implement two types of SLOs: **achievable** and **aspirational**. The achievable SLO represents a target the entire team should reach, while the aspirational SLO sets a higher goal and is part of an ongoing improvement process.

SLAs

According to Google, SLAs are “*formal or implicit agreements with your users that outline the repercussions of meeting (or failing to meet) the contained SLOs.*”

These agreements are of a more structured nature and represent business-level commitments made to customers, specifying the actions that will be taken if the organization fails to fulfill the SLA. SLAs can be either explicit or implicit. An explicit SLA entails well-defined consequences, often in terms of service credits, in case the expected reliability is not achieved. Implicit SLAs are evaluated in terms of potential damage to the organization’s reputation and the likelihood of customers switching to alternatives.

SLAs are typically established at a level that is sufficient to prevent customers from seeking alternatives, and consequently, they tend to have lower thresholds compared to SLOs. For instance, when considering the request latency SLI, the SLO might be defined at a $300ms$ SLI value, while the SLA could be set at a $500ms$ SLI value. This distinction arises from the fact that SLOs are internal targets related to reliability, whereas SLAs are external commitments. By striving to meet the SLO, the team automatically satisfies the SLA, providing an added layer of protection for the organization in case of unexpected failures.

To understand the correlation between SLIs, SLOs, and SLAs, let's look at the following figure:

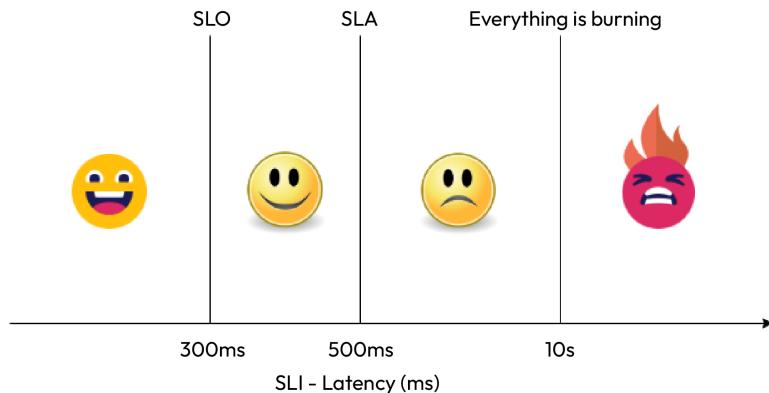


Figure 14.2 – SLIs, SLOs, and SLAs

This figure shows how customer experience changes with the level of latency. If we keep the latency SLO at *300ms* and meet it, everything is good! Anything between *300ms* to *500ms* and the customer starts experiencing some degradation in performance, but that is not enough for them to lose their cool and start raising support tickets. Therefore, keeping the SLA at *500ms* is a good strategy. As soon we cross the *500ms* threshold, unhappiness sinks in, and the customer starts raising support tickets for service slowness. If things cross the *10s* mark, then it is a cause of worry for your Ops team, and *Everything is burning* at this stage. However, as we know, the wording of SLOs is slightly different from what we imagine here. When we say that we have an SLO for *300ms* latency, it does not mean anything. A realistic SLO for an SLI mandating *request latency to remain below 300ms within the last 15 minutes with a 95th percentile measurement* would be to meet *the SLI x% of the time*. What should that x be? Should it be 99%, or should it be 95%? How do we decide this number? Well, for that, we'll have to look at **error budgets**.

Error budgets

As defined by Liz Fong-Jones and Seth Vargo, error budgets represent “*a quantitative measure shared between product and SRE teams to balance innovation and stability.*”

In simpler terms, an error budget quantifies the level of risk that can be taken to introduce new features, conduct service maintenance, perform routine enhancements, manage network and infrastructure disruptions, and respond to unforeseen situations. Typically, the monitoring system measures the uptime of your service, while SLOs establish the target you aim to achieve. The error budget is the difference between these two metrics and represents the time available to deploy new releases, provided it falls within the error budget limits.

This is precisely why a *100%* SLO is not usually set initially. Error budgets serve the crucial purpose of helping teams strike a balance between innovation and reliability. The rationale behind error

budgets lies in the SRE perspective that failures are a natural and expected part of system operations. Consequently, whenever a new change is introduced into production, there is an inherent risk of disrupting the service. Therefore, a higher error budget allows for introducing more features:

$$\text{Error Budget} = 100\% - \text{SLO}$$

For instance, if your SLO is 99%, your error budget would be 1%. If you calculate this over a month, assuming *30 days/month* and *24 hours/day*, you will have a *7.2-hour* error budget to allocate for maintenance or other activities. For a 99.9% SLO, the error budget would be *43.2 minutes* per month, and for a 99.99% SLO, it would be *4.32 minutes* monthly. You can refer to the following figure for more details:

Availability SLO	EB per Year	EB per Month
90%	36.5 days	72 hours
95%	18.25 days	36 hours
98%	7.30 days	14.4 hours
99%	3.65 days	7.20 hours
99.5%	1.83 days	3.60 hours
99.8%	17.52 hours	86.23 minutes
99.9%	8.76 hours	43.2 minutes
99.95%	4.38 hours	21.56 minutes
99.99%	52.6 minutes	4.32 minutes
99.999%	5.26 minutes	5.9 seconds
99.9999%	31.5 seconds	2.59 seconds

Figure 14.3 – Error budgets versus SLOs

These periods represent actual downtime, but if your services have redundancy, high availability measures, and disaster recovery plans in place, you can potentially extend these durations because the service remains operational while you patch or address issues with one server.

Now, whether you want to keep on adding 9s within your SLO or aim for a lower number would depend on your end users, business criticality, and availability requirements. A higher SLO is more costly and requires more resources than a lower SLO. However, sometimes, just architecting your application correctly can help you get to a better SLO target.

Now that we understand SLOs, SLIs, SLAs, and error budgets, let's talk about disaster recovery.

Disaster recovery, RTO, and RPO

Disaster recovery is a comprehensive strategy that's designed to ensure an organization's resilience in the face of unexpected, disruptive events, such as natural disasters, cyberattacks, or system failures. It involves the planning, policies, procedures, and technologies necessary to quickly and effectively restore critical IT systems, data, and operations to a functional state. A well-implemented disaster recovery plan enables businesses to minimize downtime, data loss, and financial impact, helping them maintain business continuity, protect their reputation, and swiftly recover from adversities, ultimately safeguarding their long-term success.

Every organization incorporates disaster recovery to varying degrees. Some opt for periodic backups or snapshots, while others invest in creating failover replicas of their production environment. Although failover replicas offer increased resilience, they come at the expense of doubling infrastructure costs. The choice of disaster recovery mechanism that an organization adopts hinges on two crucial KPIs – **Recovery Time Objective (RTO)** and **Recovery Point Objective (RPO)**.

RTO and RPO are crucial metrics in disaster recovery and business continuity planning. RTO represents the maximum acceptable downtime for a system or application, specifying the time within which it should be restored after a disruption. It quantifies the acceptable duration of service unavailability and drives the urgency of recovery efforts.

On the other hand, RPO defines the maximum tolerable data loss in the event of a disaster. It signifies the point in time to which data must be recovered to ensure business continuity. Achieving a low RPO means that data loss is minimized, often by frequent data backups and replication. The following figure explains RTO and RPO beautifully:

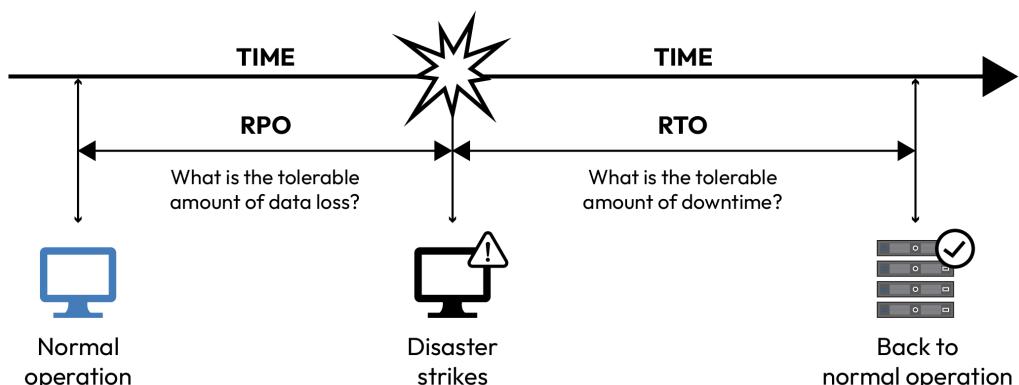


Figure 14.4 – RTO and RPO

A shorter RTO and RPO demand a more robust disaster recovery plan, which, in turn, results in higher costs for both infrastructure and human resources. Therefore, balancing RTO and RPO is essential to ensure a resilient IT infrastructure. Organizations must align their recovery strategies with these

objectives to minimize downtime and data loss, thereby safeguarding business operations and data integrity during unforeseen disruptions.

Running distributed applications in production

So far, we've been discussing KPIs for running an application in production, taking inspiration from SRE principles. Now, let's understand how we will put these thoughts in a single place to run a distributed application in production.

A **distributed application** or a **microservice** is inherently different from a monolith. While managing a monolith revolves around ensuring all operational aspects of one application, the complexity increases manyfold with microservices. Therefore, we should take a different approach to it.

From the perspective of SRE, running a distributed application in production entails focusing on ensuring the application's *reliability*, *scalability*, and *performance*. Here's how SREs approach this task:

- **SLOs:** SREs begin by defining clear SLOs that outline the desired level of reliability for the distributed application. SLOs specify the acceptable levels of *latency*, *error rates*, and *availability*. These SLOs are crucial in guiding the team's efforts and in determining whether the system is meeting its reliability goals.
- **SLIs:** SREs establish SLIs, which are quantifiable metrics that are used to measure the reliability of the application. These metrics could include response times, error rates, and other performance indicators. SLIs provide a concrete way to assess whether the application meets its SLOs.
- **Error budgets:** Error budgets are a key concept in SRE. They represent the permissible amount of downtime or errors that can occur before the SLOs are violated. SREs use error budgets to strike a balance between reliability and innovation. If the error budget is exhausted, it may necessitate a focus on stability and reliability over introducing new features.
- **Monitoring and alerting:** SREs implement robust monitoring and alerting systems to continuously track the application's performance and health. They set up alerts based on SLIs and SLOs, enabling them to respond proactively to incidents or deviations from desired performance levels. In the realm of distributed applications, using a service mesh such as **Istio** or **Linkerd** can help. They help you visualize parts of your application through a single pane of glass and allow you to monitor your application and alert on it with ease.
- **Capacity planning:** SREs ensure that the infrastructure supporting the distributed application can handle the expected load and traffic. They perform capacity planning exercises to scale resources as needed, preventing performance bottlenecks during traffic spikes. With modern public cloud platforms, automating scalability with traffic is all the more easy to implement, especially with distributed applications.
- **Automated remediation:** Automation is a cornerstone of SRE practices. SREs develop automated systems for incident response and remediation. This includes *auto-scaling*, *self-healing mechanisms*, and *automated rollback procedures* to minimize downtime.

- **Chaos engineering:** SREs often employ chaos engineering practices to introduce controlled failures into the system deliberately. This helps identify weaknesses and vulnerabilities in the distributed application, allowing for proactive mitigation of potential issues. Some of the most popular chaos engineering tools are Chaos Monkey, Gremlin, Chaos Toolkit, Chaos Blade, Pumba, ToxiProxy, and Chaos Mesh.
- **On-call and incident management:** SREs maintain on-call rotations to ensure 24/7 coverage. They follow well-defined incident management processes to resolve issues quickly and learn from incidents to prevent recurrence. Most SRE development backlogs come from this process as they learn from failures and, therefore, automate repeatable tasks.
- **Continuous improvement:** SRE is a culture of continuous improvement. SRE teams regularly conduct **post-incident reviews (PIRs)** and **root cause analyses (RCAs)** to identify areas for enhancement. Lessons learned from incidents are used to refine SLOs and improve the overall reliability of the application.
- **Documentation and knowledge sharing:** SREs document *best practices*, *runbooks*, and *operational procedures*. They emphasize knowledge sharing across teams to ensure that expertise is not siloed and that all team members can effectively manage and troubleshoot the distributed application. They also aim to automate the runbooks to ensure that manual processes are kept at a minimum.

In summary, SRE's approach to running a distributed application in production focuses on *reliability*, *automation*, and *continuous improvement*. It sets clear goals, establishes metrics for measurement, and employs proactive monitoring and incident management practices to deliver a highly available and performant service to end users.

Summary

This chapter covered SRE and the KPIs for running our service in production. We started by understanding software reliability and examined how to manage an application in production using SRE. We discussed the three crucial parameters that guide SREs: SLI, SLO, and SLA. We also explored error budgets and their importance in introducing changes within the system. Then, we looked at software disaster recovery, RPO, and RTO and how they define how complex or costly our disaster recovery measures will be. Finally, we looked at how DevOps or SRE will use these concepts to manage a distributed application in production.

In the next chapter, we will put what we've learned to practical use and explore how to manage all these aspects using a service mesh called Istio.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which of the following is a good example of an SLI?
 - A. The response time should not exceed 300ms.
 - B. The 95th percentile of response time in a 15-minute window should not exceed 300ms.
 - C. 99% of all requests should respond within 300ms.
 - D. The number of failures should not exceed 1%.
2. A mature organization should have a 100% SLO. (True/False)
3. SLOs are not tied to any customer-initiated punitive action. (True/False)
4. Which of the following should you consider while deciding on an SLO? (Choose three)
 - A. User satisfaction
 - B. Alternatives
 - C. User behavior
 - D. System capacity
5. SLAs are generally kept to a stricter SLI value than SLOs. (True/False)
6. Which of the following should you consider while defining SLIs?
 - A. CPU, memory, and disk utilization
 - B. Latency, errors, traffic, and saturation
 - C. Utilization, capacity, and scale
7. An error budget of 1% provides how much scope for downtime per month?
 - A. 72 hours
 - B. 43.2 minutes
 - C. 7.2 hours
 - D. 4.32 minutes
8. An SRE is a software developer doing Ops. (True/False)
9. What minimum percent of the time should an SRE allocate to development work?
 - A. 30%
 - B. 40%
 - C. 50%
 - D. 60%

Answers

Here are the answers to this chapter's questions:

1. B
2. False
3. True
4. A, B, C
5. False
6. B
7. C
8. True
9. C

15

Implementing Traffic Management, Security, and Observability with Istio

In the previous chapter, we covered **site reliability engineering (SRE)** and how it has helped manage production environments using DevOps practices. In this chapter, we'll dive deep into a service mesh technology called Istio, which will help us implement SRE practices and manage our application better in production.

In this chapter, we're going to cover the following main topics:

- Revisiting the Blog App
- Introduction to service mesh
- Introduction to Istio
- Understanding the Istio architecture
- Installing Istio
- Using Istio Ingress to allow traffic
- Securing your microservices using Istio
- Managing traffic with Istio
- Observing traffic and alerting with Istio

Technical requirements

For this chapter, we will spin up a cloud-based Kubernetes cluster, **Google Kubernetes Engine (GKE)**, for the exercises. At the time of writing, **Google Cloud Platform (GCP)** provides a free \$300 trial for 90 days, so you can go ahead and sign up for one at <https://console.cloud.google.com/>.

You will also need to clone the following GitHub repository for some of the exercises: <https://github.com/PacktPublishing/Modern-DevOps-Practices-2e>.

You can use the Cloud Shell offering available from GCP to follow this chapter. Go to Cloud Shell and start a new session. Run the following command to clone the repository into your home directory to access the required resources:

```
$ git clone https://github.com/PacktPublishing/Modern-DevOps-Practices-2e.git \
modern-devops
```

We also need to set the project ID and enable a few GCP APIs we will use in this chapter. To do so, run the following commands:

```
$ PROJECT_ID=<YOUR_PROJECT_ID>
$ gcloud services enable iam.googleapis.com \
container.googleapis.com \
binaryauthorization.googleapis.com \
containeranalysis.googleapis.com \
secretmanager.googleapis.com \
cloudresourcemanager.googleapis.com \
cloudkms.googleapis.com
```

If you haven't followed the previous chapters and want to start quickly with this, you can follow the next part, *Setting up the baseline*, though I highly recommend that you go through the last few chapters to get a flow. If you have been following the hands-on exercises in the previous chapters, feel free to skip this part.

Setting up the baseline

To ensure continuity with the previous chapters, let's start by creating a service account for Terraform so that we can interact with our GCP project:

```
$ gcloud iam service-accounts create terraform \
--description="Service Account for terraform" --display-name="Terraform"
$ gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="serviceAccount:terraform@$PROJECT_ID.iam.gserviceaccount.com" \
--role="roles/editor"
$ gcloud iam service-accounts keys create key-file \
--iam-account=terraform@$PROJECT_ID.iam.gserviceaccount.com
```

You will see that a file called `key-file` has been created within your working directory. Now, create a new repository called `mdo-environments` with a `README.md` file on GitHub, rename the main

branch to prod, and create a new branch called dev using GitHub. Navigate to https://github.com/<your_github_user>/mdo-environments/settings/secrets/actions/new and create a secret named **GCP_CREDENTIALS**. For the value, print the `key-file` file, copy its contents, and paste it into the **values** field of the GitHub secret.

Next, create another secret, **PROJECT_ID**, and specify your GCP project ID within the **values** field.

Next, we need to create a **GCS bucket** for Terraform to use as a remote backend. To do this, run the following commands:

```
$ gsutil mb gs://tf-state-mdm-terraorm-$PROJECT_ID
```

The next thing we need to do is set up our Secrets Manager. Let's create a secret called **external-secrets**, where we will pass the MongoDB credentials in the JSON format. To do so, run the following command:

```
$ echo -ne '{"MONGO_INITDB_ROOT_USERNAME": "root", \n    "MONGO_INITDB_ROOT_PASSWORD": "itsasecret"}' | \n    gcloud secrets create external-secrets --locations=us-central1 \n    --replication-policy=user-managed --data-file= \n    Created version [1] of the secret [external-secrets].
```

We need to create the **Secret** resource, which will interact with GCP to fetch the stored secret. First, we need to create a GCP service account to interact with Secrets Manager using the following commands:

```
$ cd ~\n$ gcloud iam service-accounts create external-secrets
```

As we're following the principle of least privilege, we will add the following role binding to provide access only to the **external-secrets** secret, as follows:

```
$ gcloud secrets add-iam-policy-binding external-secrets \n    --member "serviceAccount:external-secrets@$PROJECT_ID.iam.gserviceaccount.com" \n    --role "roles/secretmanager.secretAccessor"
```

Now, let's generate the service account key file using the following command:

```
$ gcloud iam service-accounts keys create key.json \n    --iam-account=external-secrets@$PROJECT_ID.iam.gserviceaccount.com
```

Next, copy the contents of the `key.json` file into a new GitHub Actions secret called **GCP_SM_CREDENTIALS**.

We also need to create the following GitHub Actions secrets for binary authorization to work:

```
ATTESTOR_NAME=quality-assurance-attestor
KMS_KEY_LOCATION=us-central1
KMS_KEYRING_NAME=qa-attestor-keyring
KMS_KEY_NAME=quality-assurance-attestor-key
KMS_KEY_VERSION=1
```

As the workflow automatically raises pull requests at the end, we need to define a GitHub token. This token allows the workflow to act on behalf of the current user when creating the pull request. Here are the steps:

1. Go to <https://github.com/settings/personal-access-tokens/new>.
2. Create a new token with “Repository” access for the mdo-environments repository, granting it `read-write` pull request permissions. This approach aligns with the principle of least privilege, offering more granular control.
3. Once the token is created, copy it.
4. Now, create a GitHub Actions secret named `GH_TOKEN` and paste the copied token as the value.

Now that all the prerequisites have been met, we can clone our repository and copy the baseline code. Run the following commands to do this:

```
$ cd ~ && git clone git@github.com:<your_github_user>/mdo-environments.git
$ cd mdo-environments/
$ git checkout dev
$ cp -r ~/modern-devops/ch15/baseline/* .
$ cp -r ~/modern-devops/ch15/baseline/.github .
```

As we’re now on the baseline, let’s proceed further and understand the sample Blog App that we will deploy and manage in this chapter.

Revisiting the Blog App

Since we discussed the Blog App previously, let's look at the services and their interactions again:

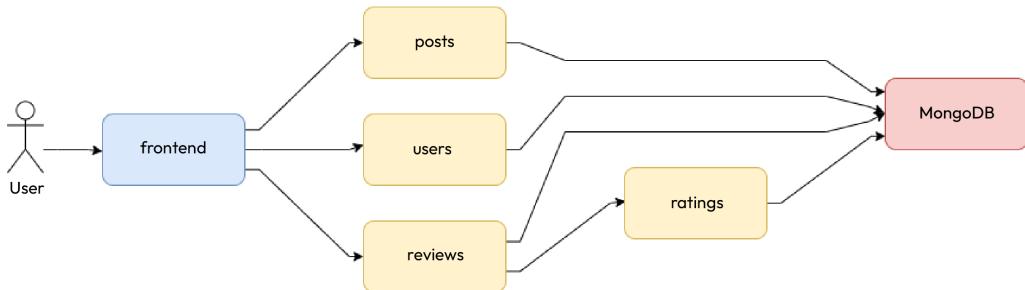


Figure 15.1 – The Blog App and its services and interactions

So far, we've created CI and CD pipelines for building, testing, and pushing our Blog App microservices containers using GitHub Actions, deploying them using Argo CD in a GKE cluster.

As you may recall, we created the following resources for the application to run seamlessly:

- **MongoDB:** We deployed an auth-enabled MongoDB database with root credentials. The credentials were injected via environment variables sourced from a Kubernetes **Secret** resource. To persist our database data, we created a **PersistentVolume** mounted to the container, which we provisioned dynamically using a **PersistentVolumeClaim**. As the container is stateful, we used a **StatefulSet** to manage it and, therefore, a headless **Service** to expose the database.
- **Posts, reviews, ratings, and users:** The *posts*, *reviews*, *ratings*, and *users* microservices interacted with MongoDB through the root credentials that were injected via environment variables sourced from the same **Secret** as MongoDB. We deployed them using their respective **Deployment** resources and exposed all of them via individual **ClusterIP Services**.
- **Frontend:** The *frontend* microservice does not need to interact with MongoDB, so there was no interaction with the Secret resource. We also deployed this service using a **Deployment** resource. As we wanted to expose the service on the internet, we created a **LoadBalancer Service** for it.

We can summarize them with the following diagram:

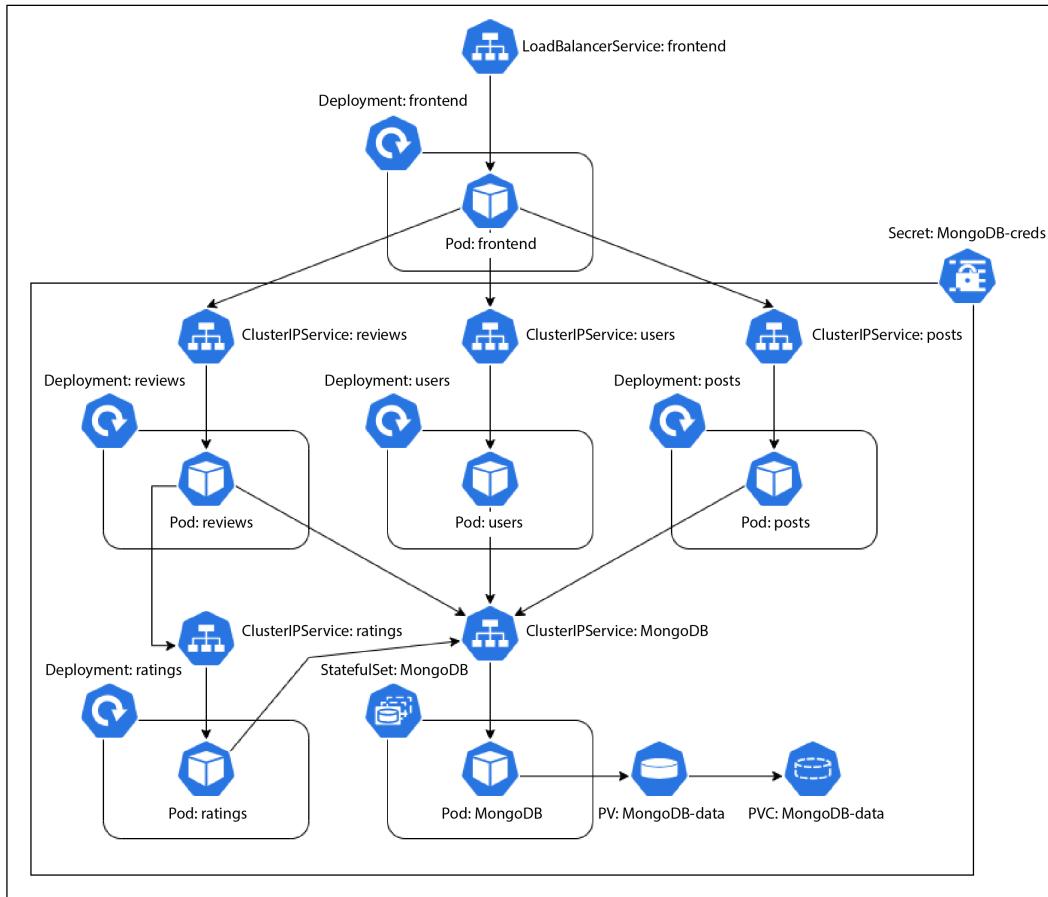


Figure 15.2 – Blog App – Kubernetes resources and interactions

Now that we understand the application, let's understand what a service mesh is and how it is beneficial in this use case.

Introduction to service mesh

Imagine being in a bustling city with a complex network of roads and highways. You're driving your car from one side of the city to the other. In this scenario, you deal with the following entities:

- **Your car:** Your car represents an individual service or application in a computer system. It has a specific purpose, just like a microservice or application in a software architecture.

- **Roads and highways:** The roads and highways are like the network connections and communication pathways between different services in your application. Services need to interact and communicate with each other to perform various functions, just as vehicles need roads to get from one place to another.
- **Traffic lights and signs:** Traffic lights, signs, and road rules help manage traffic flow, ensuring that vehicles (services) can safely and efficiently navigate the city. These are like the rules, protocols, and tools in a service mesh regulating communication and data exchange between services.
- **Traffic control center:** Think of the traffic control center as the service mesh. It's a centralized system that monitors and manages traffic flow across the city. Similarly, a service mesh is a centralized infrastructure that oversees and facilitates communication between services, ensuring they can communicate reliably and securely.
- **Traffic monitoring and optimization:** The traffic control center ensures safe travel and can optimize traffic flow. It can reroute vehicles to avoid congestion or accidents. In the context of a service mesh, it can optimize the flow of data and requests between services, ensuring efficient and resilient communication.
- **Safety and reliability:** In the city, the traffic control center helps prevent accidents and ensures everyone reaches their destinations safely. Similarly, a service mesh enhances the safety and reliability of your computer system by providing features such as load balancing, security, and fault tolerance.

So, just as the traffic control center makes your journey in a complex city more manageable and secure, a service mesh in a computer system simplifies and secures the communication between different services, ensuring that data and requests can flow smoothly, reliably, and safely.

Containers and the orchestration platforms that manage them, such as Kubernetes, have streamlined how we handle microservices. The introduction of container technology played a pivotal role in popularizing this concept by allowing for the execution and scalability of individual application components as self-contained entities, each with an isolated runtime environment.

While adopting a microservices architecture offers advantages such as accelerated development, enhanced system robustness, simplified testing, and the ability to scale different aspects of an application independently, it isn't without its challenges. Managing microservices can be a complex endeavor. Instead of dealing with a single, monolithic application, you now have multiple dynamic components, each catering to specific functionalities.

In the context of extensive applications, it's not uncommon to see hundreds of microservices interacting with each other, which can quickly become overwhelming. The primary concerns that may be raised by your security and operations teams are as follows:

- Ensuring secure communication between microservices. You need to secure numerous smaller services rather than securing a single monolithic application.

- Isolating a problematic microservice in case of an issue.
- Testing deployments with a limited percentage of traffic before a full release to establish trust.
- Consolidating application logs that are now distributed across multiple sources.
- Monitoring the health of the services becomes more intricate, with many components constituting the application.

While Kubernetes effectively addresses some management issues, it primarily serves as a container orchestration platform and excels in that role. However, it doesn't inherently solve all the complexities of a microservices architecture as they require specific solutions. Kubernetes does not inherently provide robust service management capabilities.

By default, communication between Kubernetes containers lacks security measures, and enforcing TLS between pods involves managing an extensive number of TLS certificates. Identity and access management between pods is also not applied out of the box.

While tools such as Kubernetes Network Policy can be employed to implement a firewall between pods, they function at a Layer 3 level rather than Layer 7, which is what modern firewalls operate at. This means you can identify the source of traffic but cannot inspect the data packets to make metadata-driven decisions, such as routing based on an HTTP header.

Although Kubernetes offers methods for deploying pods and conducting A/B testing and canary deployments, these processes often involve scaling container replicas. For example, deploying a new microservice version with just 10% of traffic directed to it requires at least 10 containers: 9 for the old version and 1 for the new version. Kubernetes distributes traffic evenly among pods without intelligent traffic splitting.

Each Kubernetes container within a pod maintains separate logging, necessitating a custom solution for capturing and consolidating logs.

While the Kubernetes dashboard provides features such as monitoring pods and checking their health, it does not offer insights into how components interact, the traffic distribution among pods, or the container chains that constitute the application. The inability to trace traffic flow through Kubernetes pods means you cannot pinpoint where in the chain a request encountered a failure.

To address these challenges comprehensively, a service mesh technology such as Istio can be of extreme help. This can effectively tackle the intricacies of managing microservices in Kubernetes and offer solutions for secure communication, intelligent traffic management, monitoring, and more. Let's understand what the Istio service mesh is through a brief introduction.

Introduction to Istio

Istio is a service mesh technology designed to streamline service connectivity, security, governance, and monitoring.

In the context of a microservices application, each microservice operates independently using containers, resulting in a complex web of interactions. This is where a service mesh comes into play, simplifying the discovery, management, and control of these interactions, often accomplished through a sidecar proxy. Allow me to break it down for you step by step.

Imagine a standard Kubernetes application comprising a frontend and a backend pod. Kubernetes offers built-in service discovery between pods using Kubernetes services and CoreDNS. Consequently, you can direct traffic using the service name from one pod to another. However, you won't have significant control over these interactions and runtime traffic management.

Istio steps in by injecting a sidecar container into your pod, which acts as a proxy. Your containers communicate with other containers via this proxy. This architecture allows all requests to flow through the proxy, enabling you to exert control over the traffic and collect data for further analysis. Moreover, Istio provides the means to encrypt communication between pods and enforce identity and access management through a unified control plane.

Due to this architecture, Istio boasts a range of core functionalities that enhance the traffic management, security, and observability of your microservices environment.

Traffic management

Istio effectively manages traffic by harnessing the power of the sidecar proxy, often referred to as the envoy proxy, alongside **ingress** and **egress gateways**. With these components, Istio empowers you to shape traffic and define service interaction rules. This includes implementing features such as **timeouts**, **retries**, **circuit breakers**, and much more, all through configurations within the control plane.

These capabilities open the door to intelligent practices such as **A/B testing**, **canary deployments**, and **staged rollouts** with **traffic division based on percentages**. You can seamlessly execute gradual releases, transitioning from an existing version (**Blue**) to a new one (**Green**), all with user-friendly controls.

Moreover, Istio allows you to conduct operational tests in a live production environment, offering **live traffic mirroring** to test instances. This enables you to gather real-time insights and identify potential production issues before they impact your application. Additionally, you can route requests to different language-specific microservices versions based on **geolocation or user profiles**, among other possibilities.

Security

Istio takes security seriously by securing your microservices through the envoy proxy and establishing identity access management between pods via mutual TLS. It is a robust defense against man-in-the-middle attacks through out-of-the-box **traffic encryption** between pods. This mutual authentication ensures that only trusted frontends can connect to backends, creating a strong trust relationship. Consequently, even if one of the pods is compromised, it cannot compromise the rest of your application. Istio further enhances security with **fine-grained access control policies** and introduces auditing tools currently lacking in Kubernetes, enhancing your cluster's overall security posture.

Observability

Thanks to the envoy sidecar, Istio maintains a keen awareness of the traffic flowing through the pods, enabling you to gather crucial telemetry data from the services. This wealth of data aids in gaining insights into service behavior and offers a window into future optimization possibilities for your applications. Additionally, Istio consolidates application logs and facilitates **traffic tracing** through multiple microservices. These features empower you to identify and resolve issues more swiftly, helping you isolate problematic services and expedite debugging.

Developer-friendly

Istio's most remarkable feature is its ability to relieve developers from the burdens of managing security and operational intricacies within their implementations.

Istio's Kubernetes-aware nature permits developers to continue building their applications as standard Kubernetes deployments. Istio seamlessly and automatically injects sidecar containers into the pods, sparing developers the need to worry about these technical intricacies.

Once these sidecar containers have been integrated, operations and security teams can then step in to enforce policies related to traffic management, security, and the overall operation of the application. This results in a mutually beneficial scenario for all involved parties.

Istio empowers security and operations teams to efficiently oversee microservices applications without hampering the development team's productivity. This collaborative approach ensures that each team within the organization can maintain its specialized focus and effectively contribute to the app's success. Now that we understand Istio, let's look at its architecture.

Understanding the Istio architecture

Istio simplifies microservices management through two fundamental components:

- **Data plane:** This comprises the sidecar envoy proxies that Istio injects into your microservices. These proxies take on the essential role of routing traffic between various services, and they also collect crucial telemetry data to facilitate monitoring and insights.

- **Control plane:** The control plane serves as the command center, instructing the data plane on how to route traffic effectively. It also handles the storage and management of configuration details, making it easier for administrators to interact with the sidecar proxy and take control of the Istio service mesh. In essence, the control plane functions as the intelligence and decision-making hub of Istio.

Similarly, Istio manages two types of traffic:

- **Data plane traffic:** This type of traffic consists of the core business-related data exchanged between your microservices. It encompasses the actual interactions and transactions that your application handles.
- **Control plane traffic:** In contrast, the control plane traffic consists of messages and communications between Istio components, and it is chiefly responsible for governing the behavior of the service mesh. It acts as the control mechanism that orchestrates the routing, security, and overall functioning of the microservices architecture.

The following diagram describes the Istio architecture in detail:

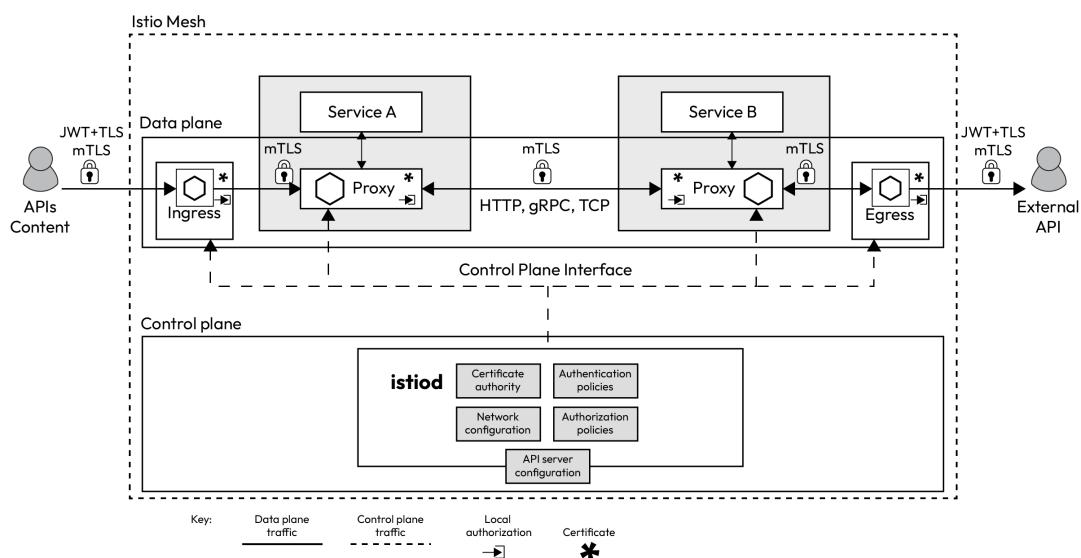


Figure 15.3 – Istio architecture

As we can see two distinct parts in the preceding diagram, the control plane and the data plane, let's go ahead and understand them.

The control plane architecture

Istio ships the control plane as a single **istiod** component. The Istio control plane, or istiod, comprises several critical components, each playing a distinct role in managing your service mesh.

Pilot

Pilot serves as the central control hub of the service mesh. It communicates with the envoy sidecars using the Envoy API and translates the high-level rules specified in Istio manifests into envoy configurations. Pilot enables service discovery, intelligent traffic management, and routing capabilities. It empowers you to implement practices such as A/B testing, Blue/Green deployments, canary rollouts, and more. Additionally, Pilot enhances the resiliency of your service mesh by configuring sidecars to handle tasks such as timeouts, retries, and circuit breaking. One of its notable features is providing a bridge between Istio configuration and the underlying infrastructure, allowing Istio to run on diverse platforms such as Kubernetes, **Nomad**, and **Consul**. Regardless of the platform, Pilot ensures consistent traffic management.

Citadel

Citadel focuses on identity and access management within your service mesh, fostering secure communication between Kubernetes pods. It safeguards your pods by ensuring encrypted communication, even if your developers have designed components with insecure TCP connections. Citadel simplifies the implementation of mutual TLS by managing the complexities of certificates. It offers user authentication, credential management, certificate handling, and traffic encryption, ensuring pods can securely validate one another when necessary.

Galley

Galley is responsible for essential configuration tasks within your service mesh. It validates, processes, and distributes configuration changes throughout the Istio control plane. For example, when you apply a new policy, Galley ingests the configuration, validates its accuracy, processes it for the intended components, and seamlessly disseminates it within the service mesh. In essence, Galley serves as the interface through which the Istio control plane interacts with the underlying APIs, facilitating the smooth management of your service mesh.

Now, let's dive deep into understanding the data plane architecture.

The data plane architecture

The data plane component of Istio is composed of **envoy proxies**, **ingress gateways**, and **egress gateways**.

Envoy proxies

Envoy proxies play a pivotal role in enabling various aspects of your service mesh. These **Layer 7** proxies are uniquely capable of making crucial decisions based on the content of the messages they

handle, and they are the sole components that directly interact with your business traffic. Here's how these envoy proxies contribute to the functionality of Istio:

- **Traffic control:** They provide fine-grained control over how traffic flows within your service mesh, allowing you to define routing rules for various types of traffic, including **HTTP**, **TCP**, **WebSockets**, and **gRPC**.
- **Security and authentication:** Envoy proxies enforce **identity and access management**, ensuring that only authorized pods can interact with one another. They implement **mutual TLS** and **traffic encryption** to prevent **man-in-the-middle attacks** and offer features such as rate limiting to safeguard against runaway costs and **denial-of-service attacks**.
- **Network resiliency:** They enhance network resiliency by supporting features such as **retries**, **failover**, **circuit breaking**, and **fault injection** to maintain the reliability and robustness of your services.

Next, let's look at Ingress and egress gateways.

Ingress and egress gateways

In Istio, ingress is a collection of one or more envoy proxies, which Pilot dynamically configures upon their deployment. These envoy proxies are crucial in controlling and routing incoming external traffic into your service mesh, ensuring that it is appropriately directed to the relevant services based on defined routing rules and policies. This dynamic configuration allows Istio to effectively manage and secure external traffic flows without requiring extensive manual intervention, ensuring that your applications can operate efficiently and securely within the service mesh.

Egress gateways are similar to ingress gateways but they work on outgoing traffic instead. To understand this better, let's use *Figure 15.3* as a reference and understand the traffic flow through **Service A** and **Service B**.

In this architecture, traffic within the service mesh follows a structured path through **Ingress**, microservices (**Service A** and **Service B**), and **Egress**, ensuring efficient routing and security measures. Let's break down the flow of a traffic packet through your service mesh.

Ingress

Traffic enters the service mesh through an ingress resource, which is essentially a cluster of envoy proxies. Pilot configures these envoy proxies upon deployment. Ingress proxies are aware of their backends due to configurations based on Kubernetes service endpoints. Ingress proxies conduct health checks, perform load balancing, and make intelligent routing decisions based on metrics such as load, packets, quotas, and traffic balancing.

Service A

Once Ingress routes the traffic to a pod, it encounters the sidecar proxy container of the Service A pod, not the actual microservice container. The envoy proxy and the microservice container share the same network namespace within the pod and have identical IP addresses and IP Table rules. The envoy proxy takes control of the pod, handling all traffic passing through it. The proxy interacts with Citadel to enforce policies, checks whether traffic needs encryption, and establishes TLS connections with the backend pod.

Service B

Service A's encrypted packet is sent to Service B, where similar steps are followed. Service B's proxy verifies the sender's identity through a TLS handshake with the source proxy. Upon establishing trust, the packet is forwarded to the Service B container, continuing the flow toward the egress layer.

Egress

The egress resource manages outbound traffic from the mesh. Egress defines which traffic can exit the mesh and employs Pilot for configuration, similar to the ingress layer. Egress resources enable the implementation of policies restricting outbound traffic to only necessary services.

Telemetry data collection

Throughout these steps, proxies collect telemetry data from the traffic. This telemetry data is sent to **Prometheus** for storage and analysis. This data can be visualized in **Grafana**, offering insights into the service mesh's behavior. The telemetry data can also be sent to external tools such as **ELK** for more in-depth analysis and machine learning applications on metrics collected.

This structured flow ensures traffic moves securely and efficiently through the service mesh while providing valuable insights for monitoring, analysis, and decision-making processes.

Now that we've understood the Istio architecture and its features, let's go ahead and see how we can install it.

Installing Istio

The general way of installing Istio is to download Istio using the provided link and run a shell, which will install Istio on our system, including the **istioctl** component. Then, we need to use **istioctl** to install Istio within a Kubernetes cluster. However, since we're using GitOps, we will use the GitOps principles to install it. Istio offers another method to install Istio – that is, using Helm. Since we know that Argo CD supports Helm, we will use that instead.

Therefore, we will create new Argo CD applications to deploy it. We will create an Argo CD application for **istio-base**, **istiod**, and **ingress**. The following YAML describes **istio-base**:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: istio-base
  namespace: argo
spec:
  project: default
  source:
    chart: base
    repoURL: https://istio-release.storage.googleapis.com/charts
    targetRevision: 1.19.1
    helm:
      releaseName: istio-base
  destination:
    server: "https://kubernetes.default.svc"
    namespace: istio-system
  syncPolicy:
    syncOptions:
    - CreateNamespace=true
  automated:
    selfHeal: true
```

As we can see, it will deploy v1.19.1 of the **istio-base** helm chart from <https://istio-release.storage.googleapis.com/charts> to the **istio-system** namespace of the Kubernetes cluster. Similarly, we will deploy **istiod** to the **istio-system** namespace using the following config:

```
...
source:
  chart: istiod
  repoURL: https://istio-release.storage.googleapis.com/charts
  targetRevision: 1.19.1
  helm:
    releaseName: istiod
destination:
  server: "https://kubernetes.default.svc"
  namespace: istio-system
...
```

Finally, we will install the **istio-ingress** component on the **istio-ingress** namespace using the following config:

```
...
source:
  chart: gateway
  repoURL: https://istio-release.storage.googleapis.com/charts
  targetRevision: 1.19.1
  helm:
```

```

releaseName: istio-ingress
destination:
  server: "https://kubernetes.default.svc"
  namespace: istio-ingress
...

```

We will also define the configuration on Terraform so that we can use push-based GitOps to create our application automatically. So, we will append the following to the `app.tf` file:

```

data "kubectl_file_documents" "istio" {
  content = file("../manifests/argocd/istio.yaml")
}
resource "kubectl_manifest" "istio" {
  depends_on = [
    kubectl_manifest.gcpsm-secrets,
  ]
  for_each = data.kubectl_file_documents.istio.manifests
  yaml_body = each.value
  override_namespace = "argocd"
}

```

Now, we can commit and push these files to our remote repository and wait for Argo CD to reconcile the changes using the following commands:

```

$ cd ~
$ cp -a ~/modern-devops/ch15/install-istio/app.tf \
~/mdo-environments/terraform/app.tf
$ cp -a ~/modern-devops/ch15/install-istio/istio.yaml \
~/mdo-environments/manifests/argocd/istio.yaml
$ git add --all
$ git commit -m "Install istio"
$ git push

```

As soon as we push the code, we'll see that the GitHub Actions workflow has been triggered. To access the workflow, go to https://github.com/<your_github_user>/mdo-environments/actions. Soon, the workflow will apply the configuration and create the Kubernetes cluster, deploy Argo CD, external secrets, our Blog App, and Istio.

Once the workflow succeeds, we must access the Argo Web UI. To do that, we need to authenticate with the GKE cluster. To do so, run the following command:

```

$ gcloud container clusters get-credentials \
  mdo-cluster-dev --zone us-central1-a --project $PROJECT_ID

```

To utilize the Argo CD Web UI, you will require the external IP address of the `argo-server` service. To get that, run the following command:

```

$ kubectl get svc argocd-server -n argocd
NAME           TYPE      EXTERNAL-IP   PORTS          AGE
argocd-server  LoadBalancer 34.122.51.25  80/TCP,443/TCP  6ml5s

```

Now, we know that Argo CD can be accessed at <https://34.122.51.25/>.

Next, we will run the following commands to reset the admin password:

```
$ kubectl patch secret argocd-secret -n argocd \
-p '{"data": {"admin.password": null, "admin.passwordMtime": null}}'
$ kubectl scale deployment argocd-server --replicas 0 -n argocd
$ kubectl scale deployment argocd-server --replicas 1 -n argocd
```

Now, allow 2 minutes for the new credentials to be generated. After that, execute the following command to retrieve the password:

```
$ kubectl -n argocd get secret argocd-initial-admin-secret \
-o jsonpath="{.data.password}" | base64 -d && echo
```

Now that we have the credentials, we can log in. We will see the following page:

The screenshot shows the Argo CD Web UI interface. At the top, there's a header with 'Applications' on the left, 'APPLICATIONS TILES' on the right, and a search bar. Below the header, there are five application tiles arranged in two rows. Each tile has a title, project information, status, repository, target revision, chart, destination, namespace, creation and last sync times, and three buttons: SYNC, REFRESH, and DELETE.

Application	Project	Status	Repository	Target Revisi...	Chart	Destination	Namespace	Created At	Last Sync
blog-app	default	Healthy	https://github.com/bharatmicrosystems/mdo-environ..	dev	manifests/blog-app	in-cluster		10/20/2023 17:13:14 (15 minutes ago)	10/20/2023 17:26:47 (a minute ago)
external-secrets	default	Healthy	https://charts.external-secrets.io	0.9.4	external-secrets	in-cluster	external-secrets	10/20/2023 17:13:08 (15 minutes ago)	10/20/2023 17:19:14 (9 minutes ago)
istio-base	default	Healthy	https://istio-release.storage.googleapis.com/charts	1.19.1	base	in-cluster	istio-system	10/20/2023 17:19:40 (8 minutes ago)	10/20/2023 17:27:44 (a few seconds ago)
istio-ingress	default	Healthy	https://istio-release.storage.googleapis.com/charts	1.19.1	gateway	in-cluster	istio-ingress	10/20/2023 17:19:40 (8 minutes ago)	10/20/2023 17:19:46 (8 minutes ago)
istiod	default	Healthy	https://istio-release.storage.googleapis.com/charts	1.19.1	istiod	in-cluster	istio-system	10/20/2023 17:19:40 (8 minutes ago)	10/20/2023 17:19:50 (8 minutes ago)

Figure 15.4 – Argo CD Web UI – home page

As we can see, the Istio applications are up and running. Though Istio is installed and running, the sidecars won't be injected unless we ask Istio to do so. We'll look at this next.

Enabling automatic sidecar injection

Since envoy sidecars are the key technology behind Istio's capabilities, they must be added to your existing pods to enable Istio to manage them. Updating each pod's configuration to include these sidecars can be challenging. To address this challenge, Istio offers a solution by enabling the automatic injection of these sidecars. To allow automatic sidecar injection on a namespace, we must add a label – that is, `istio-injection: enabled`. To do so, we will modify the `blog-app.yaml` file and add the label to the namespace resource:

```
apiVersion: v1
kind: Namespace
metadata:
  name: blog-app
  labels:
    istio-injection: enabled
...
```

Now, we can commit this resource to Git and push the changes remotely using the following commands:

```
$ cd ~
$ cp -a ~/modern-devops/ch15/install-istio/blog-app.yaml \
~/mdo-environments/manifests/blog-app/blog-app.yaml
$ git add --all
$ git commit -m "Enable sidecar injection"
$ git push
```

In the next Argo CD sync, we will soon find the label attached to the namespace. As soon as the label is applied, we need to restart our deployments and stateful sets, at which point new pods will come up with the injected sidecars. Use the following commands to do so:

```
$ kubectl -n blog-app rollout restart deploy frontend
$ kubectl -n blog-app rollout restart deploy posts
$ kubectl -n blog-app rollout restart deploy users
$ kubectl -n blog-app rollout restart deploy reviews
$ kubectl -n blog-app rollout restart deploy ratings
$ kubectl -n blog-app rollout restart statefulset mongodb
```

Now, let's list the pods in the `blog-app` namespace using the following command:

	READY	STATUS	RESTARTS	AGE
frontend-759f58f579-gqkp9	2/2	Running	0	109s
mongodb-0	2/2	Running	0	98s
posts-5cdcb5cdf6-6wjrr	2/2	Running	0	108s
ratings-9888d6fb5-j2712	2/2	Running	0	105s
reviews-55ccb7fdb9-vw72m	2/2	Running	0	106s
users-5dbd56c4c5-stgjp	2/2	Running	0	107s

As we can see, the pods now show two containers instead of one. The extra container is the envoy sidecar. Istio's installation and setup are complete.

Now that our application has the Istio sidecar injected, we can use Istio ingress to allow traffic to our application, which is currently exposed via a load balancer service.

Using Istio ingress to allow traffic

We need to create a Blog App ingress gateway to associate our application with the Istio ingress gateway. It is necessary for configuring our application to route traffic through the Istio ingress gateway as we want to leverage Istio's traffic management and security features.

Istio deploys the Istio ingress gateway as a part of the installation process, and it's exposed on a load balancer by default. To determine the load balancer's IP address and ports, you can run the following commands:

```
$ kubectl get svc istio-ingress -n istio-ingress
NAME           EXTERNAL-IP      PORT(S)
istio-ingress   34.30.247.164   80:30950/TCP,443:32100/TCP
```

As we can see, Istio exposes various ports on your load balancer, and as our application needs to run on port 80, we can access it using `http://<IngressLoadBalancerExternalIP>:80`.

The next step would be to use this ingress gateway and expose our application. For that, we need to create **Gateway** and **VirtualService** resources.

Istio gateway is a **custom resource definition (CRD)** that helps you define how incoming external traffic can access services in your mesh. It acts as an entry point to your service and a load balancer for incoming traffic. When external traffic arrives at a gateway, it determines how to route it to the appropriate services based on the specified routing rules.

When we define an Istio gateway, we also need to define a **VirtualService** resource that uses the gateway and describes the routing rules for the traffic. Without a **VirtualService** resource, the Istio gateway will not know where and how to route the traffic it receives. A **VirtualService** resource is not only used for routing traffic from gateways but also for routing traffic within different services of the mesh. It allows you to define sophisticated routing rules, including traffic splitting, retries, timeouts, and more. Virtual services are often associated with specific services or workloads and determine how traffic should be routed to them. You can use virtual services to control how traffic is distributed among different versions of a service, enabling practices such as A/B testing, canary deployments, and Blue/Green deployments. Virtual services can also route traffic based on HTTP headers, paths, or other request attributes. In the current context, we will use the **VirtualService** resource to filter traffic based on paths and route them all to the **frontend** microservice.

Let's look at the definition of the **Gateway** resource first:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: blog-app-gateway
```

```
namespace: blog-app
spec:
  selector:
    istio: ingress
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

As we can see, we define a `Gateway` resource that uses the Istio ingress gateway (defined by the `istio: ingress` selector) and listens on HTTP port 80. It allows connection to all hosts as we've set that to `"*"`. For gateways to work correctly, we need to define a `VirtualService` resource. Let's look at that next:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: blog-app
  namespace: blog-app
spec:
  hosts:
  - "*"
  gateways:
  - blog-app-gateway
  http:
  - match:
    - uri:
        exact: /
    - uri:
        prefix: /static
    - uri:
        prefix: /posts
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        exact: /register
    - uri:
        exact: /updateprofile
  route:
  - destination:
      host: frontend
      port:
        number: 80
```

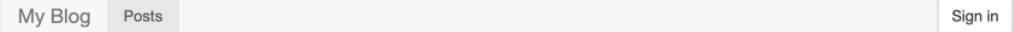
The `VirtualService` resource listens on all hosts and applies to `blog-app-gateway` as specified. It allows `/static`, and `/posts` as a prefix match. This means all requests with a URI that begins with them would be routed. The `/login`, `/logout`, `/register`, `/updateprofile`, and `/` paths have an exact match, which means that the exact URI is matched and allowed. These are routed to the frontend service on port 80.

We must also modify the `frontend` service within the `blog-app.yaml` file to change the service type to `ClusterIP`. This will remove the attached load balancer from the service, and all requests will be routed via the ingress gateway.

Now, let's go ahead and apply these changes using the following commands:

```
$ cd ~/mdo-environments  
$ cp ~/modern-devops/ch15/istio-ingressgateway/gateway.yaml \  
manifests/blog-app/gateway.yaml  
$ cp ~/modern-devops/ch15/istio-ingressgateway/blog-app.yaml \  
manifests/blog-app/blog-app.yaml  
$ git add --all  
$ git commit -m "Added gateway"  
$ git push
```

We will wait 5 minutes for the sync to work, after which we can go to `http://<Ingress LoadBalancerExternalIP>` to access our Blog App. You should see the following page. This shows that the application is working correctly:



Welcome to My Blog

Figure 15.5 – Blog App – home page

You can play around with the application by registering, logging in, creating a post, and writing a review. Try updating the post and reviews to see whether all aspects of the application are working. Now, let's look at the security aspects of our microservices.

Securing your microservices using Istio

Running microservices in production offers numerous advantages, such as independent scalability, enhanced agility, reduced scope of change, frequent deployments, and reusability. However, they also introduce unique challenges, particularly in terms of security.

In a monolithic architecture, the security focus revolves around safeguarding a single application. However, in a typical enterprise-grade microservices application, hundreds of microservices may

need to interact securely with each other. Kubernetes serves as an excellent platform for hosting and orchestrating microservices. Nevertheless, the default communication between microservices is insecure, as they typically use plaintext HTTP. This may not meet your security requirements. To apply the same security principles to microservices as you would to a traditional enterprise monolith, you must ensure the following:

- **Encrypted communications:** All interactions between microservices must be encrypted to prevent potential man-in-the-middle attacks
- **Access control:** Access control mechanisms need to be in place to ensure that only authorized microservices can interface with each other
- **Telemetry and audit logging:** Capturing, logging, and auditing telemetry data is crucial to understanding traffic behavior and proactively detecting intrusions

Istio simplifies addressing these security concerns and provides these essential security features out of the box. With Istio, you can enforce strong **identity and access management**, mutual **TLS** and **encryption, authentication and authorization**, and comprehensive **audit logging** – all within a unified control plane. This means you can establish robust security practices for your microservices, promoting the safety and reliability of your applications in a dynamic and distributed environment.

In the context of Istio, you should be aware that it automatically injects sidecar proxies into your pods and modifies the IP tables of your Kubernetes cluster to ensure that all connections occur through these proxies. This setup is designed to enforce TLS encryption by default, enhancing the security of your microservices without requiring specific configurations. The communication between these envoy proxies within the service mesh is automatically secured through TLS.

While the default setup offers a foundational level of security and effectively prevents man-in-the-middle attacks, it's advisable to further bolster the security of your microservices by applying specific policies. Before delving into the detailed features, having a high-level understanding of how security functions in Istio is beneficial.

Istio incorporates the following key components for enforcing security:

- **Certificate authority (CA):** This component manages keys and certificates, ensuring secure and authenticated communication within the service mesh.
- **Configuration API Server:** The Configuration API Server distributes authentication policies, authorization policies, and secure naming information to the envoy proxies. These policies define how services can authenticate and authorize each other and manage secure communication.
- **Sidecar proxies:** Sidecar proxies, deployed alongside your microservices, are crucial in enforcing security policies. They act as policy enforcement points, implementing the policies supplied to them.
- **Envoy proxy extensions:** These extensions enable the collection of telemetry data and auditing, providing insights into traffic behavior and helping to identify and mitigate security issues.

With these components working in concert, Istio ensures a robust security framework for your microservices, which can be further fine-tuned by defining and enforcing specific security policies tailored to your application's needs.

As our application currently runs on HTTP, it would be a great idea to implement TLS in our Blog App and expose it over HTTPS. Let's start by creating a secure ingress gateway for this.

Creating secure ingress gateways

Secure ingress gateways are nothing but **TLS-enabled ingress gateways**. To enable TLS on an ingress gateway, we must provide it with a **private key** and a **certificate chain**. We will use a self-signed certificate chain for this exercise, but you must use a proper CA certificate chain in production. A CA certificate is a digital certificate that's granted by a reputable CA, such as Verisign or Entrust, within a **public key infrastructure (PKI)**. It plays a pivotal role in guaranteeing the security and reliability of digital interactions and transactions.

Let's start by creating a **root certificate** and **private key** to sign certificates for our application by using the following command:

```
$ openssl req -x509 -sha256 -nodes -days 365 \
-newkey rsa:2048 -subj '/O=example Inc./CN=example.com' \
-keyout example.com.key -out example.com.crt
```

Using the generated root certificate, we can now generate the **server certificate** and the key using the following commands:

```
$ openssl req -out blogapp.example.com.csr \
-newkey rsa:2048 -nodes -keyout blogapp.example.com.key \
-subj "/CN=blogapp.example.com/O=blogapp organization"
$ openssl x509 -req -sha256 -days 365 \
-CA example.com.crt -CAkey example.com.key -set_serial 1 \
-in blogapp.example.com.csr -out blogapp.example.com.crt
```

The next step is to generate a Kubernetes TLS secret within the `istio-ingress` namespace for our ingress gateway to read it. However, as we don't want to store the TLS key and certificate in our Git repository, we will use **Google Secrets Manager** instead. Therefore, let's run the following command to do so:

```
$ echo -ne "{\"MONGO_INITDB_ROOT_USERNAME\": \"root\", \
\"MONGO_INITDB_ROOT_PASSWORD\": \"itsasecret\", \
\"blogapptlskey\": \"$(base64 blogapp.example.com.key -w 0)\", \
\"blogapptlscert\": \"$(base64 blogapp.example.com.crt -w 0)\"}" | \
gcloud secrets versions add external-secrets --data-file=-
Created version [2] of the secret [external-secrets].
```

Now, we must create an external secret manifest to fetch the keys and certificates from Secrets Manager and generate a TLS secret. The following manifest will help us achieve that:

```
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  name: blogapp-tls-credentials
  namespace: istio-ingress
spec:
  secretStoreRef:
    kind: ClusterSecretStore
    name: gcp-backend
  target:
    template:
      type: kubernetes.io/tls
      data:
        tls.crt: "{{ .blogapptlscert | base64decode | toString }}"
        tls.key: "{{ .blogapptlskey | base64decode | toString }}"
      name: blogapp-tls-credentials
    data:
      - secretKey: blogapptlskey
        remoteRef:
          key: external-secrets
          property: blogapptlskey
      - secretKey: blogapptlscert
        remoteRef:
          key: external-secrets
          property: blogapptlscert
```

Now, let's create a directory within our Environment Repository and copy the external secret manifest there. Use the following commands for that:

```
$ mkdir ~/mdo-environments/manifests/istio-ingress
$ cp ~/modern-devops/ch15/security/blogapp-tls-credentials.yaml \
~/mdo-environments/manifests/istio-ingress
```

Next, we need to modify the ingress gateway resource to configure TLS. To do so, we must modify the `Gateway` resource to the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: blog-app-gateway
  namespace: blog-app
spec:
  selector:
    istio: ingress
  servers:
    - port:
        number: 443
```

```
name: https
protocol: HTTPS
tls:
  mode: SIMPLE
  credentialName: blogapp-tls-credentials
hosts:
- "*"
```

The gateway configuration is similar to the previous one, but instead of port 80, we're using port 443 for HTTPS. We also have a `tls` section with a SIMPLE mode, which means it is a standard TLS connection. We've specified `credentialName`, pointing to the secret we created using the TLS key and certificate. Since all the setup is now ready, let's commit and push the code using the following commands:

```
$ cp ~/modern-devops/ch15/security/gateway.yaml \
~/mdo-environments/manifests/blog-app/
$ cp ~/modern-devops/ch15/security/run-tests.yml \
~/mdo-environments/.github/workflows/
$ git add --all
$ git commit -m "Enabled frontend TLS"
$ git push
```

Wait for `blog-app` to sync. Once we've done this, we can access our application at `https://<IngressLoadBalancerExternalIP>`. With that, the connection coming into our application has been encrypted.

Though we've secured connection coming into our mesh, securing all internal service interactions with services using TLS within your service mesh would be good as an additional security layer. We'll implement that next.

Enforcing TLS within your service mesh

As we know by now, by default, Istio provides TLS encryption for communication between workloads that have sidecar proxies injected. However, it's important to note that this default setting operates in compatibility mode. In this mode, traffic between two services with sidecar proxies injected is encrypted. However, workloads without sidecar proxies can still communicate with backend microservices over plaintext HTTP. This design choice is made to simplify the adoption of Istio, as teams newly introducing Istio don't need to immediately address the issue of making all source traffic TLS-enabled.

Let's create and get a shell to a pod in the `default` namespace. The backend traffic will be plaintext because the namespace does not have automatic sidecar injection. We will then `curl` the frontend microservice from there and see whether we get a response. Run the following command to do so:

```
$ kubectl run -it --rm --image=curlimages/curl curly -- curl -v http://frontend.blog-app
*   Trying 10.71.246.145:80...
*   Connected to frontend (10.71.246.145) port 80
> GET / HTTP/1.1
```

```

> Host: frontend
> User-Agent: curl/8.4.0
> Accept: */*
< HTTP/1.1 200 OK
< server: envoy
< date: Sat, 21 Oct 2023 07:19:18 GMT
< content-type: text/html; charset=utf-8
< content-length: 5950
< x-envoy-upstream-service-time: 32
<!doctype html>
<html la="g=en">
...

```

As we can see, we get an HTTP 200 response back.

This approach balances security and compatibility, allowing a gradual transition to a fully encrypted communication model. Over time, as more services have sidecar proxies injected, the overall security posture of the microservices application improves. However, as we are starting fresh, enforcing strict TLS for our Blog App would make sense. So, let's do that.

To enable strict TLS on a workload, namespace, or the entire cluster, Istio provides peer authentication policies using the `PeerAuthentication` resource. As we only need to implement strict TLS on the Blog App, enabling it at the namespace level would make sense. To do that, we will use the following `PeerAuthentication` resource:

```

apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: blog-app
spec:
  mtls:
    mode: STRICT

```

Now, let's apply this using the following commands:

```

$ cp ~/modern-devops/ch15/security/strict-mtls.yaml \
~/mdo-environments/manifests/blog-app/
$ git add --all
$ git commit -m "Enable strict TLS"
$ git push

```

Argo CD should pick up the new configuration and apply the strict TLS policy as soon as we push the changes. Wait for the Argo CD sync to be in a clean state, and run the following commands to check whether strict TLS is working:

```

$ kubectl run -it --rm --image=curlimages/curl  curly -- curl -v http://frontend.blog-app
*   Trying 10.71.246.145:80...
*   Connected to frontend.blog-app (10.71.246.145) port 80
> GET / HTTP/1.1
> Host: frontend.blog-app

```

```
> User-Agent: curl/8.4.0
> Accept: */
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
```

As we can see, the request has now been rejected as it is a plaintext request, and the backend will only allow TLS. This shows that strict TLS is working fine. Now, let's move on and secure our services even better.

From our design, we know how services interact with each other:

- The `frontend` microservice can only connect to the `posts`, `reviews`, and `users` microservices
- Only the `reviews` microservice can connect to the `ratings` microservice.
- Only the `posts`, `reviews`, `users`, and `ratings` microservices can connect to the `mongodb` database

Therefore, we can define these interactions and only allow these connections explicitly. Therefore, the `frontend` microservice will not be able to connect with the `mongodb` database directly, even if it tries to.

Istio provides the `AuthorizationPolicy` resource to manage this. Let's implement the preceding scenario using that.

Let's start with the `posts` microservice:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: posts
  namespace: blog-app
spec:
  selector:
    matchLabels:
      app: posts
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The `AuthorizationPolicy` has multiple sections. It starts with `name` and `namespace`, which are `posts` and `blog-app`, respectively. The `spec` section contains `selector`, where we specify that we need to apply this policy to all pods with the `app: posts` label. We use an `ALLOW` action for this. Note that Istio has an implicit `deny-all` policy for all pods that match the selector, and any `ALLOW` rules will be applied on top of that. Any traffic that does not match the `ALLOW` rules will be denied by default. We have rules to define what traffic to allow; here, we're using the `from >`

source > principals and setting the frontend service account on this. So, in summary, this rule will apply to the posts microservice and only allow traffic from the frontend microservice.

Similarly, we will apply the same policy to the reviews microservice, as follows:

```
...
  name: reviews
...
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The users microservice also only needs to accept traffic from the frontend microservice:

```
...
  name: users
...
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/blog-app/sa/frontend"]
```

The ratings microservice should accept traffic only from the reviews microservice, so we will make a slight change to the principals, as follows:

```
...
  name: ratings
...
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/blog-app/sa/reviews"]
```

Finally, the mongodb service needs a connection from all microservices apart from frontend, so we must specify multiple entries in the principal section:

```
...
  name: mongodb
...
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/blog-app/sa/posts", "cluster.local/ns/blog-app/sa/reviews", "cluster.local/ns/blog-app/sa/ratings", "cluster.local/ns/blog-app/sa/users"]
```

Since we've used service accounts to understand where the requests are coming from, we must also create and assign service accounts to respective services. So, we will modify the blog-app.yaml file and add service accounts for each service, something like the following:

```
apiVersion: v1
kind: ServiceAccount
```

```
metadata:
  name: mongodb
  namespace: blog-app
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
...
spec:
...
  template:
...
    spec:
      serviceAccountName: mongodb
      containers:
...

```

I've already replicated the same in the new `blog-app.yaml` file. Let's commit the changes and push them to GitHub so that we can apply them to our cluster:

```
$ cp ~/modern-devops/ch15/security/authorization-policies.yaml \
~/mdo-environments/manifests/blog-app/
$ cp ~/modern-devops/ch15/security/blog-app.yaml \
~/mdo-environments/manifests/blog-app/
$ git add --all
$ git commit -m "Added auth policies"
$ git push
```

Now, we must wait for the sync to complete and then verify the setup. First, we'll get a shell to the frontend pod and try to use `wget` to connect with the backend microservices. We will try to connect with each microservice and see what we get. If we get HTTP 200 or 404, this means the backend is allowing connections, while if we get HTTP 403 or Error, this signifies the backend is blocking connections. Run the following commands to do so:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep frontend | awk {'print $1'}) -- /bin/sh
/ # wget posts:5000
Connecting to posts:5000 (10.71.255.204:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget reviews:5000
Connecting to reviews:5000 (10.71.244.177:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
```

```
wget: error getting response: Resource temporarily unavailable
/ # exit
command terminated with exit code 1
```

As we can see, we get an HTTP 404 response from the posts, reviews, and users microservices. The ratings microservice returns a 403 Forbidden response, and the mongodb service reports that the resource is unavailable. This means that our setup is working correctly.

Let's try the same with the posts microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep posts | awk {'print $1'}) -- /bin/sh
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html      100% |*****|     85  0:00:00 ETA
'index.html' saved
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget reviews:5000
Connecting to reviews:5000 (10.71.244.177:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the posts microservice can communicate successfully with mongodb, but the rest of the microservices return 403 Forbidden. This is what we were expecting. Now, let's do the same with the reviews microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app | \
grep reviews | awk {'print $1'}) -- /bin/sh
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 404 Not Found
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html      100% |*****|     85  0:00:00 ETA
'index.html' saved
/ # wget users:5000
Connecting to users:5000 (10.71.241.255:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the `reviews` microservice can successfully connect with the `ratings` microservice and `mongodb`, while getting a `403` response from other microservices. This is what we expected. Now, let's check the `ratings` microservice:

```
$ kubectl -n blog-app exec -it $(kubectl get pod -n blog-app \
| grep ratings | awk '{print $1}') -- /bin/sh
/ # wget mongodb:27017
Connecting to mongodb:27017 (10.68.0.18:27017)
saving to 'index.html'
index.html      100% |*****|     85  0:00:00 ETA
'index.html' saved
/ # wget ratings:5000
Connecting to ratings:5000 (10.71.242.178:5000)
wget: server returned error: HTTP/1.1 403 Forbidden
/ # exit
command terminated with exit code 1
```

As we can see, the `ratings` microservice can only connect successfully with the `mongodb` database and gets a `403` response for other services.

Now that we've tested all the services, the setup is working fine. We've secured our microservices to a great extent! Now, let's look at another aspect of managing microservices with Istio – traffic management.

Managing traffic with Istio

Istio offers robust traffic management capabilities that form a core part of its functionality. When leveraging Istio for microservice management within your Kubernetes environment, you gain precise control over how these services communicate with each other. This empowers you to define the traffic path within your service mesh meticulously.

Some of the traffic management features at your disposal are as follows:

- Request routing
- Fault injection
- Traffic shifting
- TCP traffic shifting
- Request timeouts
- Circuit breaking
- Mirroring

The previous section employed an ingress gateway to enable traffic entry into our mesh and used a virtual service to distribute traffic to the services. With virtual services, the traffic distribution happens in a round-robin fashion by default. However, we can change that using destination rules. These rules provide us with an intricate level of control over the behavior of our mesh, allowing for a more granular management of traffic within the Istio ecosystem.

Before we delve into that, we need to update our Blog App so that it includes a new version of the ratings service deployed as `ratings-v2`, which will return black stars instead of orange stars. I've already updated the manifest for that in the repository. Therefore, we just need to copy that to the `mdo-environments` repository, commit it, and push it remotely using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app/  
$ cp ~/modern-devops/ch15/traffic-management/blog-app.yaml .  
$ git add --all  
$ git commit -m "Added ratings-v2 service"  
$ git push
```

Wait for the application to sync. After this, we need to do a few things:

1. Go to the Blog App home page > **Sign In** > **Not a User? Create an account** and create a new account.
2. Click on the **Actions** tab > **Add a Post**, add a new post with a title and content of your choice, and click **Submit**.
3. Use the **Add a Review** text field to add a review, provide a rating, and click **Submit**.
4. Click on **Posts** again and access the post that we had created.

Now, keep refreshing the page. We will see that we get orange stars half the time and black stars for the rest. Traffic is splitting equally across v1 and v2 (that is, the orange and black stars):

The screenshot shows a blog post interface. At the top, there are navigation tabs: 'My Blog' (selected), 'Posts', and 'Actions'. Below the tabs, the title 'Welcome to my first post' is displayed. Underneath the title, it says 'Author: gaurav@example.com'. A message 'Welcome to the Blog! I hope you are enjoying reading it!' follows. A section titled 'Reviews:' contains two entries. The first review is 'This is a great first post! Loved it!' with a 5-star rating icon and edit/copy icons. The second review is 'This could have been better!' with a 3-star rating icon and edit/copy icons. Both reviews are attributed to 'gaurav@example.com'. Below the reviews, there's a section titled 'Add a Review:' with a 'Review' button and a text input field placeholder 'Enter your review'. At the bottom, there's a rating section with a minus sign, five yellow stars, and a green button labeled 'Five Stars'. A 'Submit' button is also present.

Figure 15.6 – Round robin routing

This occurs due to the absence of destination rules, which leaves Istio unaware of the distinctions between v1 and v2. Let's define destination rules for our microservices to rectify this, clearly informing Istio of these versions. In our case, we have one version for each microservice, except for the ratings microservice, so we'll define the following destination rules accordingly.

Let's start by defining the destination rule of the frontend microservice:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: frontend
  namespace: blog-app
spec:
  host: frontend
  subsets:
  - name: v1
    labels:
      version: v1
```

The provided YAML manifest introduces a `DestinationRule` resource named `frontend` within the `blog-app` namespace. This resource is associated with the host named `frontend`. Subsequently, we define subsets labeled as `v1`, targeting pods with the `version: v1` label. Consequently, configuring our virtual service to direct traffic to the `v1` destination will route requests to pods bearing the `version: v1` label.

This same configuration approach can be replicated for the `posts`, `users`, and `reviews` microservices. However, the `ratings` microservice requires a slightly different configuration due to the deployment of two versions, as follows:

```
...
spec:
  host: ratings
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

The YAML manifest for the `ratings` microservice closely resembles that of the other microservices, with one notable distinction: it features a second subset labeled as `v2`, corresponding to pods bearing the `version: v2` label.

Consequently, requests routed to the `v1` destination target all pods with the `version: v1` label, while requests routed to the `v2` destination are directed to pods labeled `version: v2`.

To illustrate this in a practical context, we will proceed to define virtual services for each microservice. Our starting point will be defining the virtual service for the `frontend` microservice, as illustrated in the following manifest:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: frontend
  namespace: blog-app
spec:
  hosts:
  - frontend
  http:
  - route:
    - destination:
      host: frontend
      subset: v1
```

The provided YAML manifest outlines a `VirtualService` resource named `frontend` within the `blog-app` namespace. This resource configures the host `frontend` with an HTTP route destination, directing all traffic to the `frontend` host and specifying the `v1` subset. Consequently, all requests targeting the `frontend` host will be routed to the `v1` destination that we previously defined.

We will replicate this configuration approach for the posts, reviews, and users microservices, creating corresponding `VirtualService` resources. In the case of the `ratings` microservice, the decision is made to route all traffic to the `v1` (orange stars) version. Therefore, we apply a similar `VirtualService` resource for the `ratings` microservice as well.

Now, let's copy the manifests to the `mdo-environments` repository and commit and push the code to the remote repository using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app
$ cp ~/modern-devops/ch15/traffic-management/destination-rules.yaml .
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-v1.yaml .
$ git add --all
$ git commit -m "Route to v1"
$ git push
```

Wait for Argo CD to sync the changes. Now, all requests will route to `v1`. Therefore, you will only see orange stars in the reviews, as shown in the following screenshot:

The screenshot shows a web application interface. At the top, there is a navigation bar with tabs: 'My Blog' (which is active and highlighted in blue) and 'Posts'. Below the navigation bar, the main content area displays a post titled 'Welcome to my first post'. The author is listed as 'Author: gaurav@example.com'. The post content is 'Welcome to the Blog! I hope you are enjoying reading it!'.

Reviews:

- This is a great first post! Loved it! ★★★★★
- gaurav@example.com
- This could have been better! ★★★★☆
- gaurav@example.com

Add a Review:

Review
Enter your review

- ★★★★★ Five Stars
Submit

Figure 15.7 – Route to v1

Now, let's try to roll out v2 using a canary rollout approach.

Traffic shifting and canary rollouts

Consider a scenario where you've developed a new version of your microservice and are eager to introduce it to your user base. However, you're understandably cautious about the potential impact on the entire service. In such cases, you may opt for a deployment strategy known as **canary rollouts**, also known as a **Blue/Green deployment**.

The essence of a canary rollout lies in its incremental approach. Instead of an abrupt transition, you methodically shift traffic from the previous version (referred to as **Blue**) to the new version (**Green**). This gradual migration allows you to thoroughly test the functionality and reliability of the new release with a limited subset of users before implementing it across the entire user base. This approach minimizes the risks associated with deploying new features or updates and ensures a more controlled and secure release process. The following figure illustrates this process beautifully:

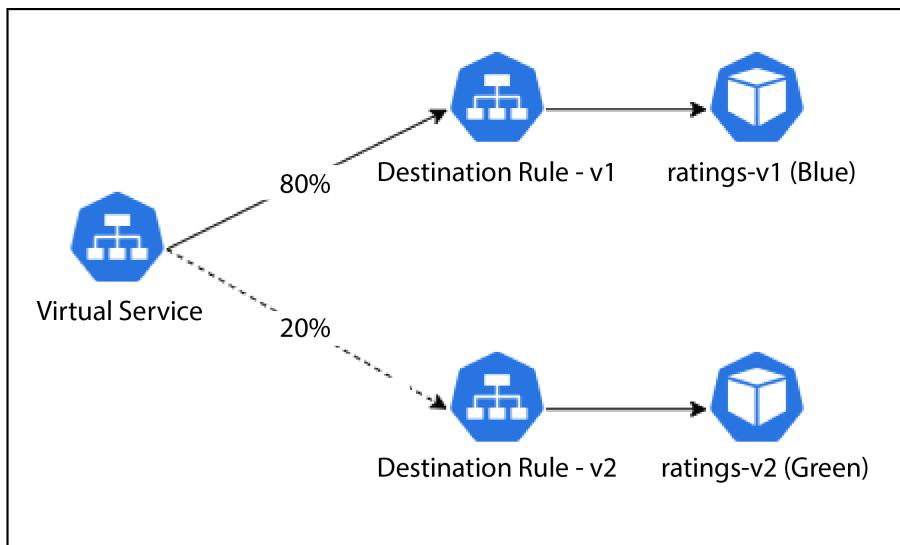


Figure 15.8 – Canary rollouts

Here's how a *canary rollout* strategy works:

- Initial release:** The existing version (referred to as the *baseline* or *current version*) continues to serve the majority of users.
- Early access:** A small group of users or systems, typically selected as a representative sample, is identified as the *canary group*. They receive the new version.
- Monitoring and evaluation:** The software's performance and behavior in the canary group are closely monitored. Metrics, logs, and user feedback are collected to identify issues or anomalies.

4. **Gradual expansion:** If the new version proves stable and performs as expected in the canary group, its exposure is incrementally expanded to a broader user base. This expansion can occur in stages, with a small percentage of users being “promoted” to the new version at each stage.
5. **Continuous monitoring:** Throughout the rollout, continuous monitoring and analysis are critical to identify and address any emerging issues promptly. If problems are detected, the rollout can be halted or reversed to protect the majority of users.
6. **Full deployment:** Once the new version has been successfully validated through the canary rollout stages, it is eventually made available to the entire user base.

So, let's roll out the `ratings-v2` service to 20% of our users. For that, we'll use the following `VirtualService` resource:

```
...
http:
- route:
  - destination:
    host: ratings
    subset: v1
    weight: 80
  - destination:
    host: ratings
    subset: v2
    weight: 20
```

As we can see, we've modified the `ratings` virtual service to introduce a second destination pointing to the `v2` subset. A noteworthy addition in this configuration is the introduction of the `weight` attribute. For the `v1` destination, we have assigned a weight of 80, while the `v2` destination carries a weight of 20. This means that 20% of the traffic will be directed to the `v2` version of the `ratings` microservice, providing a controlled and adjustable distribution of traffic between the two versions.

Let's copy the manifest and then commit and push the changes to the remote repository using the following commands:

```
$ cd ~/mdo-environments/manifests/blog-app
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-canary.yaml \
virtual-services.yaml
$ git add --all
$ git commit -m "Canary rollout"
$ git push
```

Following the completion of the Argo CD sync, if you refresh the page 10 times, you'll observe that black stars appear twice out of those 10 times. This is a prime example of a canary rollout in action. You can continue monitoring the application and gradually adjust the weights to shift traffic toward `v2`. Canary rollouts effectively mitigate risks during production rollouts, providing a method to address the fear of the unknown, especially when implementing significant changes.

However, another approach exists to test your code in a production environment that involves using live traffic without exposing your application to end users. This method is known as traffic mirroring. We'll delve into it in the following discussion.

Traffic mirroring

Traffic mirroring, also called shadowing, is a concept that has recently gained traction. It is a powerful approach that allows you to assess your releases in a production environment without posing any risk to your end users.

Traditionally, many enterprises maintained a staging environment that closely mimicked the production setup. The Ops team deployed new releases to the staging environment in this setup while testers generated synthetic traffic to simulate real-world usage. This approach provided a means for teams to evaluate how the code would perform in the production environment, assessing its functional and non-functional aspects before promoting it to production. The staging environment served as the ground for performance, volumetric, and operational acceptance testing. While this approach had its merits, it was not without its challenges. Maintaining static test environments, which involved substantial costs and resources, was one of them. Creating and sustaining a replica of the production environment required a team of engineers, leading to high overhead.

Moreover, synthetic traffic often deviated from real live traffic since the former relied on historical data, while the latter reflected current user interactions. This discrepancy occasionally led to overlooked scenarios.

On the other hand, traffic mirroring offers a solution that similarly enables operational acceptance testing while going a step further. It allows you to conduct this testing using live, real-time traffic without any impact on end users.

Here's how traffic mirroring operates:

1. Deploy a new version of the application and activate traffic mirroring.
2. The old version continues to respond to requests as usual but concurrently sends an asynchronous copy of the traffic to the new version.
3. The new version processes the mirrored traffic but refrains from responding to end users.
4. The ops team monitors the behavior of the new version and reports any issues to the development team.

This process is depicted in the following figure:

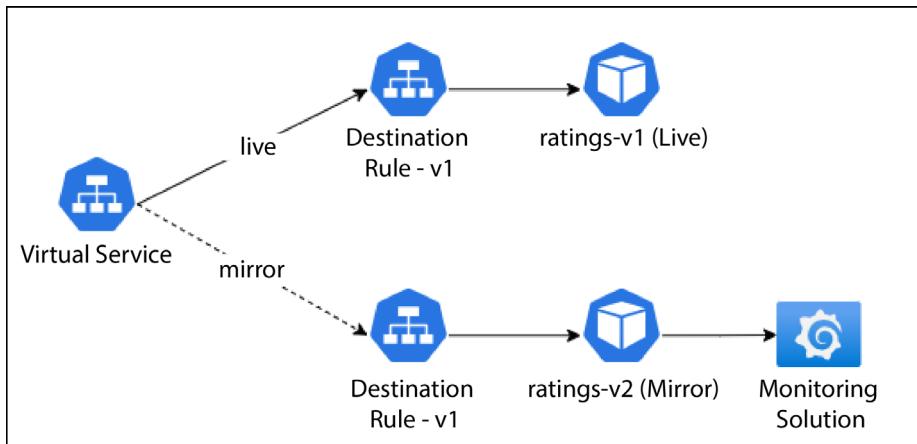


Figure 15.9 – Traffic mirroring

Traffic mirroring revolutionizes the testing process by enabling teams to uncover issues that might remain hidden in a traditional staging environment. Additionally, you can utilize monitoring tools such as Prometheus and Grafana to record and monitor the outcomes of your testing efforts, enhancing the overall quality and reliability of your releases.

Now, without further ado, let's configure traffic mirroring for our `ratings` service. Traffic mirroring is managed through the `VirtualService` resource, so let's modify the `ratings` virtual service to the following:

```
...  
  http:  
    - route:  
      - destination:  
          host: ratings  
          subset: v1  
          weight: 100  
      mirror:  
        host: ratings  
        subset: v2  
        mirror_percent: 100
```

In this configuration, we set up a single destination targeting `v1` with a `weight` value of 100. Additionally, we defined a `mirror` section that directs traffic to `ratings:v2` with a `mirror_percent` value of 100. This signifies that all traffic initially routed to `ratings:v1` is mirrored and simultaneously sent to `v2`.

Let's commit the changes and push them to the remote repository using the following commands:

```
$ cp ~/modern-devops/ch15/traffic-management/virtual-services-mirroring.yaml \
virtual-services.yaml
$ git add --all
$ git commit -m "Mirror traffic"
$ git push
```

Following the completion of the Argo CD synchronization process, we'll proceed to refresh the page five times. Subsequently, we can inspect the logs of the `ratings:v1` service using the following command:

```
$ kubectl logs $(kubectl get pod -n blog-app | \
grep "ratings-" | awk '{print $1}') -n blog-app
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534ccb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534ccb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:25] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
```

With traffic mirroring active, it's expected that the same set of logs observed in the `ratings:v1` service will also be mirrored in the `ratings:v2` service. To confirm this, we can list the logs for the `ratings:v2` service using the following command:

```
$ kubectl logs $(kubectl get pod -n blog-app | \
grep "ratings-v2" | awk '{print $1}') -n blog-app
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:19] "GET /review/6534ccb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:23] "GET /review/6534ccb32485f5a51cbdcef1/rating
HTTP/1.1" 200 -
127.0.0.6 - - [22/Oct/2023 08:33:25] "GET /review/6534cba72485f5a51cbdcef0/rating
HTTP/1.1" 200 -
```

Indeed, the logs and timestamps match precisely, providing clear evidence of concurrent log entries in `ratings:v1` and `ratings:v2`. This observation effectively demonstrates the mirroring functionality in operation, showcasing how traffic is duplicated for real-time monitoring and analysis in both versions.

Traffic mirroring is a highly effective method for identifying issues that often elude detection within traditional infrastructure setups. It is a potent approach for conducting operational acceptance testing of your software releases. This practice simplifies testing and safeguards against potential customer incidents and operational challenges.

There are other aspects of traffic management that Istio provides, but covering all of them is beyond the scope of this chapter. Please feel free to explore other aspects of it by visiting the Istio documentation: <https://istio.io/latest/docs/tasks/traffic-management/>.

As we already know, Istio leverages envoy proxies as sidecar components alongside your microservice containers. Given that these proxies play a central role in directing and managing the traffic within your service mesh, they also collect valuable telemetry data.

This telemetry data is subsequently transmitted to Prometheus, a monitoring and alerting tool, where it can be stored and effectively visualized. Tools such as Grafana are often employed in conjunction with Prometheus to provide insightful and accessible visualizations of this telemetry data, empowering you to monitor and manage your service mesh effectively. Therefore, we'll go ahead and explore the observability portion of Istio in the next section.

Observing traffic and alerting with Istio

Istio provides several tools to visualize traffic through our mesh through Istio add-ons. While **Prometheus** is the central telemetry data collection, storage, and query layer, **Grafana** and **Kiali** provide us with interactive graphical tools to interact with that data.

Let's start this section by installing the observability add-ons using the following commands:

```
$ cd ~
$ mkdir ~/mdo-environments/manifests/istio-system
$ cd ~/mdo-environments/manifests/istio-system/
$ cp ~/modern-devops/ch15/observability/*.yaml .
$ git add --all
$ git commit -m "Added observability"
$ git push
```

As soon as we push the code, Argo CD should create a new `istio-system` namespace and install the add-ons. Once they have been installed, we can start by accessing the Kiali dashboard.

Accessing the Kiali dashboard

Kiali is a powerful observability and visualization tool for microservices and service mesh management. It offers real-time insights into the behavior of your service mesh, helping you monitor and troubleshoot issues efficiently.

As the Kiali service is deployed on a cluster IP and hence not exposed externally, let's do a port forward to access the Kiali dashboard using the following command:

```
$ kubectl port-forward deploy/kiali -n istio-system 20001:20001
```

Once the port forward session has started, click on the web preview icon of Google Cloud Shell, choose **Change port to 20001**, and click **preview**. You will see the following dashboard. This dashboard provides valuable insights into the applications running across the mesh:

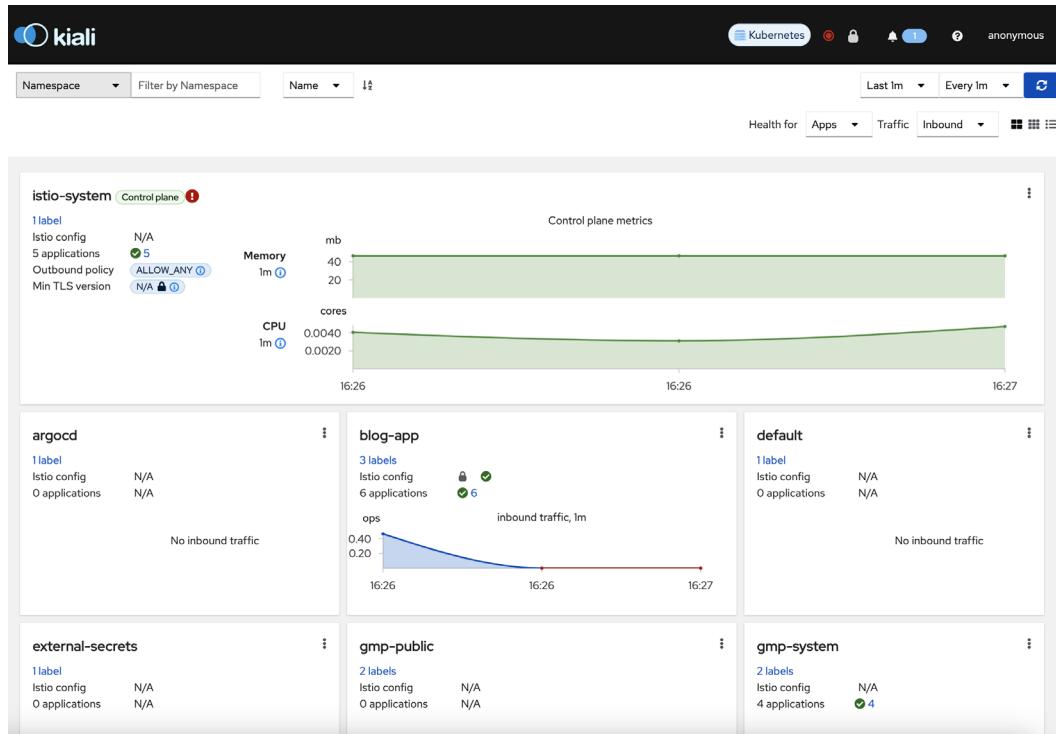


Figure 15.10 – Kiali dashboard

To visualize service interactions, we can switch to the graph view by clicking on the **Graph** tab and selecting the **blog-app** namespace. We will see the following dashboard, which provides an accurate view of how traffic flows, the percentage of successful traffic, and other metrics:

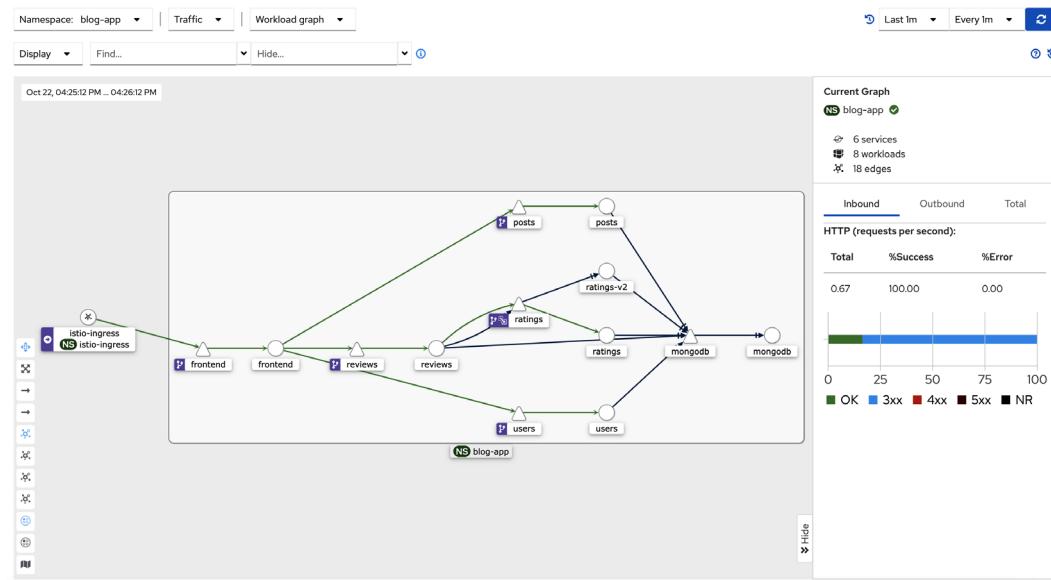


Figure 15.11 – Kiali service interaction graph

While Kiali dashboards provide valuable insights regarding our mesh and help us observe service interactions in real time, they lack the capability of providing us with advanced monitoring and alerting capabilities. For that, we can use Grafana.

Monitoring and alerting with Grafana

Grafana is a leading open source platform for observability and monitoring, offering dynamic dashboards and robust alerting capabilities. It enables users to visualize data from diverse sources while setting up alerts for proactive issue detection.

As we've already installed Grafana with the necessary add-ons, let's access it by opening a port forward session. Ensure you terminate the existing Kiali port-forwarding session or use a different port. Run the following command to do so:

```
$ kubectl port-forward deploy/grafana -n istio-system 20001:3000
```

Once the port forwarding session has started, access the Grafana page like we did for Kiali, and go to **Home > Dashboards > Istio > Istio Service Dashboard**. We should see a dashboard similar to the following:

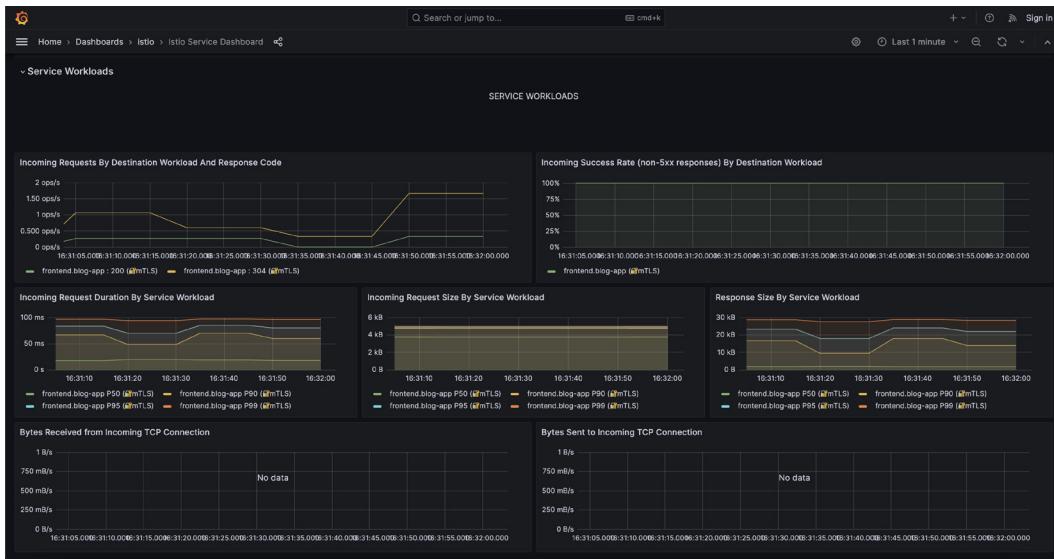


Figure 15.12 – Istio service dashboard

This dashboard provides rich visualizations regarding some standard SLIs we may want to monitor, such as the request’s *success rate*, *duration*, *size*, *volume*, and *latency*. It helps you observe your mesh meticulously, and you can also build additional visualizations based on your requirements by using the **Prometheus Query Language (PromQL)**, which is simple to learn and apply.

However, monitoring and visualization must be complemented by alerting for complete reliability. So, let’s delve into that.

Alerting with Grafana

To initiate the alerting process, it’s crucial to establish clear criteria. Given the limited volume at hand, simulating an accurate SLO breach can be challenging. For simplicity, our alerting criteria will trigger when traffic volume surpasses one transaction per second.

The initial phase of this process involves crafting the query to retrieve the necessary metrics. We will employ the following query to achieve this objective:

```
round(sum(irate(istio_requests_total{connection_security_policy="mutual_tls",destination_service=~"frontend.blog-app.svc.cluster.local",reporter=~"destination",source_workload=~"istio-ingress",source_workload_namespace=~"istio-ingress"}[5m])) by (source_workload, source_workload_namespace, response_code), 0.001)
```

The provided query determines the traffic rate for all transactions passing through the Istio ingress gateway to the frontend microservice.

The next step involves creating the alert rules with the query in place. To do this, navigate to **Home > Alerting > Alert rules**. Then, fill in the form, as illustrated in the following screenshot:

The screenshot shows the Istio Alerting UI interface for defining an alert rule. The process is divided into four steps:

- 1 Set an alert rule name**: A text input field contains the rule name "request-exceed-limit".
- 2 Set a query and alert condition**:
 - A Prometheus query builder is used to define the alert condition. The query is: `round(sum(irate(istio_requests_total{connection_security_policy="mutual_tls",destination_service=~"frontend.blog-app.svc.cluster.local",reporter=~"destination",source_workload=~"istio-ingress",source_workload_namespace=~"istio-ingress"}[5m])) by (source_workload,source_workload_namespace, response_code), 0.001)`.
 - Below the query, there are two configuration sections:
 - B Reduce**: Function is set to "Last", Mode to "Strict".
 - C Threshold**: Input B is compared to 1 using the "IS ABOVE" operator.
 - A "Preview" button is available at the bottom of the query section.
- 3 Alert evaluation behavior**:
 - A dropdown for "Folder" is set to "istio".
 - An "Evaluation group (interval)" dropdown is set to "request".
 - The "for" interval is set to "2m".
 - A radio button for "Pause evaluation" is selected.
 - A link to "Edit evaluation group" is present.
 - A "Configure no data and error handling" link is visible.
- 4 Add details for your alert rule**:
 - "Summary and annotations" section:
 - Description**: ALERT - Traffic crossed limit.
 - Summary**: The traffic crossed the configured limit of 1 request per sec.
 - Choose**: Text.
 - Buttons at the bottom include "+ Add annotation" and "Set dashboard and panel".

Figure 15.13 – Defining alert rules

The alert rule is configured to monitor for violations at a 1-minute interval for 2 consecutive minutes. Once the alert rule has been established, triggering the alert is as simple as refreshing the Blog App home page about 15–20 times rapidly every 1 to 2 minutes. This action should activate the alert. To observe this process, navigate to **Home > Alerting > Alert rules**. You will notice the alert in a **Pending** state in the first minute. This means it has detected a violation in one of its checks and will wait for another violation within the 2-minute duration before triggering the alert.

In a production environment, setting longer check intervals, typically around 5 minutes, with alerting intervals of 15 minutes is typical. This approach helps avoid excessive alerting for self-resolving transient issues, ensuring the SRE team is not inundated with false alerts. The goal is to maintain a balance and prevent the team from treating every alert as a potential false alarm. The following screenshot shows a pending alert:

The screenshot shows the Grafana interface for managing alert rules. The title bar says "Alert rules" and "Rules that determine whether an alert will fire". There is a pink error icon with the text "1 error". The top navigation bar includes "Search by data sources", "State" (with dropdowns for "All data sources", "Firing", "Normal", "Pending", "Rule type" (Alert, Recording), and "Health" (Ok, No Data, Error)), and "View as" (Grouped, List, State). Below the search bar is a search input field and a "Create alert rule" button. The main content area shows "1 rule | 1 pending". The rule details are as follows:

Mimir / Cortex / Loki	1 pending	1m	1
No rules found.			

At the bottom right are "Export" and "Create alert rule" buttons.

Figure 15.14 – Alert pending

After the 2-minute monitoring period, you should observe the alert being triggered, as depicted in the following screenshot. This indicates that the alert rule has successfully identified a sustained violation of the defined criteria and is now actively notifying relevant parties or systems:

State	Name	Health
▼ Firing for 28s	request-exceed-limit	ok
Silence Show state history		
Evaluate	Every 1m	
For	2m	
Last evaluation	a few seconds ago	
Evaluation time	0s	
Description	ALERT - Traffic crossed limit	
Summary	The traffic crossed the configured limit of 1 request per sec	

Figure 15.15 – Alert firing

Since no specific alert channels have been configured in this context, the fired alerts will be visible within the Grafana dashboard only. It is highly advisable to set up a designated alert destination for sending alerts to your designated channels, using a tool such as **PagerDuty** to page on-call engineers or **Slack** notifications to alert your on-call team. Proper alert channels ensure that the right individuals or teams are promptly notified of critical issues, enabling rapid response and issue resolution.

Summary

As we conclude this chapter and wrap up this book, our journey has taken us through an array of diverse concepts and functionalities. While we've covered substantial ground in this chapter, it's essential to recognize that Istio is a rich and multifaceted technology, making it a challenge to encompass all its intricacies within a single chapter.

This chapter marked our initiation into the world of service mesh, shedding light on its particular advantages in the context of microservices. Our exploration extended to various dimensions of Istio, beginning with installing Istio and extending our sample Blog App to utilize it using automatic sidecar injection. We then moved on to security, delving into the intricacies of securing ingress gateways with mTLS, enforcing strict mTLS among microservices, and harnessing authorization policies to manage traffic flows.

Our journey then led us to traffic management, where we introduced essential concepts such as destination rules and virtual services. These enabled us to carry out canary rollouts and traffic mirroring, demonstrating the power of controlled deployments and real-time traffic analysis. Our voyage culminated in observability, where we harnessed the Kiali dashboard to visualize service interactions and ventured deep into advanced monitoring and alerting capabilities using Grafana.

As we end this remarkable journey, I want to extend my heartfelt gratitude to you for choosing this book and accompanying me through its pages. I trust you've found every part of this book enjoyable and enlightening. I hope this book has equipped you with the skills necessary to excel in the ever-evolving realm of modern DevOps. I wish you the utmost success in all your present and future endeavors.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Which approach would you use to install Istio among the available options using GitOps methodology?
 - A. Istioctl
 - B. Helm charts
 - C. Kustomize
 - D. Manifest bundle
2. What configuration is necessary for Istio to inject sidecars into your workloads automatically?
 - A. Apply the `istio-injection-enabled: true` label to the namespace
 - B. No configuration is needed – Istio automatically injects sidecars into all pods
 - C. Modify the manifests so that they include the Istio sidecars and redeploy them
3. Istio sidecars automatically communicate with each other using mTLS. (True/False)
4. Which resource enforces policies that dictate which services are permitted to communicate with each other?
 - A. AuthenticationPolicy
 - B. AuthorizationPolicy
 - C. PeerAuthentication
5. Which of the following resources would you use for canary rollouts? (Choose two)
 - A. VirtualService
 - B. IngressGateway
 - C. DestinationRule
 - D. Egress Gateway
6. Why would you use traffic mirroring in production? (Choose three)
 - A. Real-time monitoring for production performance and behavior analysis
 - B. To route traffic to a new version to duplicate traffic to test the performance of your backend service

- C. Safe testing of changes or updates without risking production disruptions
 - D. Streamlined troubleshooting and debugging for issue identification and resolution
7. Which observability tool would you use to visualize real-time service interactions?
- A. Prometheus
 - B. Grafana
 - C. Kiali
 - D. Loki

Answers

Here are the answers to this chapter's questions:

1. B
2. A
3. True
4. B
5. A and C
6. A, C, and D
7. C

Appendix:

The Role of AI in DevOps

The recent developments in **artificial intelligence (AI)** with the launch of generative AI using ChatGPT have taken the tech industry by storm. It has made many existing AI players pivot, and most companies are now looking at the best ways to use it in their products. Naturally, DevOps and the tooling surrounding it are no exceptions, and slowly, AI is gaining firm ground in this discipline, which historically relied upon more traditional automation methods. Before we delve into how AI changes DevOps, let's first understand what AI is.

This appendix will cover the following topics:

- What is AI?
- The role of AI in the DevOps infinity loop

What is AI?

AI emulates human intelligence in computing. You know how our computers do fantastic things, but they need to be told everything to do? Well, AI doesn't work like that. It learns a ton from looking at lots of information, like how we learn from our experiences. That way, it can figure out patterns independently and make decisions without needing someone to tell it what to do every time. This makes AI intelligent because it can keep learning new things and get better at what it does.

Imagine if your computer could learn from everything it sees, just like you remember from everything around you. That's how AI works—it's a computer's way of getting more intelligent. Instead of needing step-by-step instructions, AI learns from vast amounts of information. This makes it great at spotting patterns in data and deciding things on its own. And when it comes to DevOps, AI can be of great help! Let's look at that next.

The role of AI in the DevOps infinity loop

As we are already aware, instead of following a linear path of software delivery, DevOps practices generally follow an infinity loop, as shown in the following figure:



Figure A.1 – DevOps infinity loop

DevOps practices heavily emphasize automation to ensure that this infinity loop operates smoothly, and we need tools. Most of these tools help build, deploy, and operate your software. You will typically start writing code in an **integrated development environment (IDE)** and then check code into a central source code repository such as Git. There will be a continuous integration pipeline that will build code from your Git repository and push it to an artifact repository. Your QA team might write automated tests to ensure the artifact is tested before it is deployed to higher environments using a continuous deployment pipeline.

Before the advent of AI, setting up all of the toolchains and operating them relied on traditional coding methods; that is, you would still write code to automate the processes, and the automation would behave more predictably and do what it was told to. However, with AI, things are changing.

AI is transforming DevOps by automating tasks, predicting failures, and optimizing performance. In other words, by leveraging AI's capabilities, DevOps teams can achieve greater efficiency, reduce errors, and deliver software faster and more reliably.

Here are some key roles of AI in DevOps:

- **Automating Repetitive Tasks:** AI can automate repetitive and tedious tasks, such as code testing, deployment, and infrastructure provisioning. This frees up DevOps engineers to focus on more strategic and creative work, such as developing new features and improving application performance.
- **Predicting and Preventing Failures:** AI can analyze vast amounts of data, including logs, performance metrics, and user feedback, to identify patterns and predict potential failures. This proactive approach allows DevOps teams to address issues before they impact users or cause major disruptions.

- **Optimizing Resource Utilization:** AI can analyze resource usage data to optimize infrastructure allocation and prevent resource bottlenecks. This ensures that applications have the resources they need to perform optimally, minimizing downtime and improving overall system efficiency.
- **Enhancing Security:** AI can be used to detect and prevent security threats by analyzing network traffic, identifying anomalous behavior, and flagging suspicious activity. This helps DevOps teams maintain a robust security posture and protect sensitive data.
- **Improving Collaboration and Communication:** AI can facilitate collaboration and communication among DevOps teams by providing real-time insights, automating workflows, and enabling seamless communication channels. This breaks down silos and promotes a more cohesive DevOps culture.

Let's look at the areas of the DevOps infinity loop and see how AI impacts them.

Code development

This area is where we see the most significant impact of generative AI and other AI technologies. AI revolutionizes code development by automating tasks such as code generation, bug detection, optimization, and testing. Through autocomplete suggestions, bug detection algorithms, and predictive analytics, AI accelerates coding, enhances code quality, and ensures better performance while aiding in documentation and code security analysis. Its role spans from assisting in writing code to predicting issues, ultimately streamlining the software development life cycle, and empowering developers to create more efficient, reliable, and secure applications.

Many tools employ AI in code development, and one of the most popular tools in this area is **GitHub Copilot**.

GitHub Copilot is a collaborative effort between **GitHub** and **OpenAI**, introducing a code completion feature that utilizes OpenAI's **Codex**. Codex, trained on vast code repositories from GitHub, quickly generates code based on the current file's content and cursor location. Compatible with popular code editors such as **Visual Studio Code**, **Visual Studio**, **Neovim**, and **JetBrains IDEs**, Copilot supports languages such as **Python**, **JavaScript**, **TypeScript**, **Ruby**, and **Go**.

Praised by GitHub and users alike, Copilot generates entire code lines, functions, tests, and documentation. Its functionality relied on the context provided and the extensive code contributions by developers on GitHub, regardless of their software license. Dubbed the world's first AI pair programmer by **Microsoft**, it is a paid tool and charges a subscription fee of \$10 per month or \$100 per year per user after a 60-day trial period.

With Copilot, you can start by writing comments on what you intend to do, and it will generate the required code for you. This speeds up development many times, and most of the time, you just need to review and test your code to see whether it does what you intend it to do. A great power indeed! It can optimize existing code and provide feedback by generating code snippets. It can also scan your code for security vulnerabilities and suggest alternative approaches.

If you don't want to pay that \$10, you can also look at free alternatives such as **Tabnine**, **Captain Stack**, **GPT-Code Clippy**, **Second Mate**, and **Intellicode**. Paid alternatives include Amazon's **Code Whisperer** and Google's **ML-enhanced code completion**.

AI tools not only help enhance the development workflow but also help in software testing and quality assurance. Let's look at that next.

Software testing and quality assurance

Traditionally, software testing has taken more of a manual approach because most developers don't want software testing as a full-time profession. Though automation testing has gained ground recently, the knowledge gap has hindered this process in most organizations. Therefore, AI would most impact the testing function as it bridges the human-machine gap.

AI-integrated testing techniques revolutionize every stage of the **software testing life cycle (STLC)**; some of them are as follows:

- **Test script generation:** Traditionally, creating test scripts was time-consuming, involving deep system understanding. AI and **machine learning (ML)** now expedite this process by analyzing requirements, existing test cases, and application behavior to craft more optimized test scripts, offering ready-to-use templates with preconfigured code snippets and comprehensive comments, and translating plain language instructions into complete test scripts using **natural language processing (NLP)** techniques.
- **Test data generation:** AI-equipped testing tools provide detailed and ample test data for comprehensive coverage. They achieve this by generating synthetic data from existing sets for specific test objectives, transforming data to create diverse testing scenarios, refining existing data for higher precision and relevance, and scanning large code bases for context comprehension.
- **Intelligent test execution:** AI alleviates test execution challenges by automatically categorizing and organizing test cases, efficiently selecting tests for various devices, operating systems, and configurations, and smartly executing regression tests for critical functionalities.
- **Intelligent test maintenance:** AI/ML minimizes test maintenance challenges by implementing self-healing mechanisms to handle broken selectors and analyzing UI and code change relationships to identify affected areas.
- **Root cause analysis:** AI aids in understanding and rectifying issues by analyzing logs, performance metrics, and anomalies to pinpoint impact areas, tracing issues back to affected user stories and feature requirements, and utilizing knowledge repositories for comprehensive root cause analysis.

Multiple tools in the market help you achieve all of it; some of the most popular ones are the following:

- **Katalon platform:** A comprehensive quality management tool that simplifies test creation, execution, and maintenance across various applications and environments. It boasts AI features such as **TrueTest, StudioAssist, self-healing, visual testing, and AI-powered test failure analysis.**
- **TestCraft:** Built on **Selenium**, TestCraft offers both manual and automated testing capabilities with a user-friendly interface and AI-driven element identification, allowing tests to run across multiple browsers in parallel.
- **Applitools:** Known for its AI-based visual testing, Applitools efficiently identifies visual bugs, monitors app visual aspects, and provides accurate visual test analytics using AI and ML.
- **Function:** Utilizes AI/ML for functional, performance, and load testing with simplicity, allowing test creation through plain English input, self-healing, test analytics, and multi-browser support.
- **Mabl:** An AI-powered tool offering low-code testing, intuitive intelligence, data-driven capabilities, end-to-end testing, and valuable insights generation, promoting team collaboration.
- **AccelQ:** Automates test designs, plans, and execution across the UI, mobile, API, and PC software, featuring **automated test generation, predictive analysis, and comprehensive test management.**
- **Testim:** Uses ML to expedite test creation and maintenance, allowing for quick end-to-end test creation, smart locators for resilient tests, and a blend of recording functions and coding for robust test creation.

As we've already seen the benefits of AI in development and testing, let's move on to software delivery.

Continuous integration and delivery

In **continuous integration (CI)** and **continuous delivery (CD)**, AI brings a transformative edge by optimizing and automating various stages of the software development pipeline. AI augments CI by automating code analysis, identifying patterns, and predicting potential integration issues. It streamlines the process by analyzing code changes, suggesting appropriate test cases, and facilitating faster integration cycles. Through ML, AI can understand historical data from past builds, recognizing patterns that lead to failures, thereby aiding in more efficient debugging and code quality improvement.

In CD, AI optimizes deployment pipelines by automating release strategies, predicting performance bottlenecks, and suggesting optimizations for smoother delivery. It analyzes deployment patterns, user feedback, and system performance data to recommend the most efficient delivery routes. Additionally, AI-driven CD tools enhance risk prediction, allowing teams to foresee potential deployment failures and make informed decisions to mitigate risks before they impact production environments. Ultimately, AI's role in CI/CD accelerates the development cycle, improves software quality, and enhances the reliability of software releases.

Here are some AI-powered tools used in software release and delivery:

- **Harness:** Harness utilizes AI to automate software delivery processes, including continuous integration, deployment, and verification. It employs ML to analyze patterns from deployment pipelines, predict potential issues, and optimize release strategies for better efficiency and reliability.
- **GitClear:** GitClear employs AI algorithms to analyze code repositories and provides insights into developer productivity, code contributions, and team performance. It helps understand code base changes, identify bottlenecks, and optimize development workflows.
- **Jenkins:** Thanks to its plugin-based architecture, Jenkins, a widely used automation server, employs a lot of AI plugins and extensions to enhance its capabilities in CI/CD. AI-powered plugins help automate tasks, optimize build times, and predict build failures by analyzing historical data.
- **CircleCI:** CircleCI integrates AI and ML to optimize CI/CD workflows. It analyzes build logs, identifies patterns leading to failures, and provides recommendations to improve build performance and reliability.

These AI-powered tools improve software release and delivery processes' speed, quality, and reliability by automating tasks, optimizing workflows, predicting issues, and providing valuable insights for better decision-making.

Now, let's look at the next stage in the process—software operations.

Software operations

AI is pivotal in modern software operations, revolutionizing how systems are monitored, managed, and optimized. By leveraging ML algorithms, AI helps automate routine tasks such as monitoring system performance, analyzing logs, and identifying anomalies in real time. It enables predictive maintenance by detecting patterns that precede system failures, allowing for proactive intervention and preventing potential downtime. Additionally, AI-powered tools streamline incident management by correlating alerts, prioritizing critical issues, and providing actionable insights, enhancing the overall resilience and reliability of software operations.

Moreover, AI augments decision-making processes by analyzing vast amounts of data to identify trends, forecast resource requirements, and optimize infrastructure utilization. AI adapts to changing environments through its continuous learning capabilities, enabling software operations teams to stay ahead of evolving challenges and complexities. Overall, AI's role in software operations ensures greater efficiency, improved system performance, and proactive problem resolution, contributing significantly to the seamless functioning of IT infrastructures.

Here are some AI-powered tools used in software operations:

- **Dynatrace:** Dynatrace utilizes AI for application performance monitoring and management. It employs AI algorithms to analyze vast amounts of data, providing real-time insights into application performance, identifying bottlenecks, and predicting potential issues before they impact end users.
- **PagerDuty:** PagerDuty integrates AI-driven incident management, alerting, and on-call scheduling. It uses ML to correlate events and alerts, reducing noise and providing intelligent notifications for critical incidents.
- **Opsani:** Opsani leverages AI for autonomous optimization of cloud applications. It analyzes application performance, dynamically adjusts configurations, and optimizes resources to maximize performance and cost-efficiency.
- **Moogsoft:** Moogsoft offers AI-driven IT operations and AIOps platforms. It uses ML to detect anomalies, correlate events, and automate incident resolution, helping teams proactively manage and resolve issues in complex IT environments.
- **Sumo Logic:** Sumo Logic employs AI for log management, monitoring, and analytics. It uses ML to identify patterns, anomalies, and security threats within logs and operational data, enabling proactive troubleshooting and security incident detection.
- **New Relic:** New Relic utilizes AI for application and infrastructure monitoring. Its AI-powered platform helps identify performance issues, predict system behavior, and optimize resource utilization for better application performance.
- **LogicMonitor:** LogicMonitor uses AI for infrastructure monitoring and observability. It analyzes metrics and performance data to provide insights into system health, predict potential issues, and optimize resource allocation in complex environments.
- **OpsRamp:** OpsRamp employs AI for IT operations management, offering capabilities for monitoring, incident management, and automation. It uses ML to detect anomalies, automate routine tasks, and optimize workflows for better operational efficiency.

These AI-powered tools assist in automating tasks, predicting and preventing issues, optimizing resource allocation, and enhancing overall system reliability in software operations.

The integration of AI into DevOps practices is still in its early stages, but its potential impact is significant. By automating tasks, optimizing processes, and enhancing collaboration, AI can revolutionize the way software is developed, deployed, and managed. As AI technology continues to develop, we can expect to see even more ways in which AI is used to improve the DevOps process.

Summary

AI revolutionizes DevOps practices by infusing intelligence into every development and operations cycle stage. It streamlines processes, enhances efficiency, and ensures smoother collaboration between development and operations teams. AI automates routine tasks, predicts potential bottlenecks, and optimizes workflows, transforming how software is built, tested, deployed, and monitored. From automating code analysis to predicting system failures, AI empowers DevOps by enabling quicker decision-making, reducing errors, and fostering a more agile and responsive software development environment.

In essence, AI acts as a silent partner, continuously learning from data, suggesting improvements, and helping DevOps teams foresee and address issues before they impact the software's performance. It's the catalyst that drives agility and innovation, allowing DevOps to evolve from a mere collaboration between teams to a symbiotic relationship where AI enhances the capabilities of both development and operations, paving the way for more efficient and reliable software delivery.

Index

A

A/B testing 22, 378, 475
access controls 446
adapter pattern 156-158
advanced Kubernetes resources
 need for 162, 163
Agile 338
alerting
 with Grafana 510-512
aliases
 using 203
Amazon CloudWatch 212
Amazon EC2 211, 212
Amazon Elastic Container Registry (ECR) 118, 416
Amazon Elastic Container Service (Amazon ECS) 209, 211
 architecture 211-214
 command-line interface (CLI) 214
 containers running on ECS, load balancing 223-225
 EC2 tasks, scheduling on 217
 Fargate tasks, scheduling on 218-221
 task definitions, creating 215-217

Amazon Web Services (AWS) 7, 129, 421
Amazon Web Services Fargate
 (AWS Fargate) 209, 211
ambassador pattern 146, 147
Anchore 416
Anchore Grype 416
 images, scanning 417-419
 installing 416
 URL 416
Ansible 22, 53
 architecture 279
 benefits 277, 278
 configuration file, setting up 285, 286
 control node 279
 control node, connecting with inventory servers 280-282
 features 277
 installing, in control node 283
 inventory file, setting up 283, 284
 inventory, setting up 280
 managed nodes 279
 modules 286
 setting up 279
 tasks 286, 287
Ansible facts 298
 used, for finding metadata 298, 299

Ansible playbooks 287, 289

application packages, installing 291, 292
applications, configuring 292-294
applying 289
combining 294
executing 294-296
features 287, 288
packages, updating 290
repositories, updating 290
services, installing 291, 292
syntax, checking 289
working with 290

Ansible provisioner 320**Ansible roles** 300

apache directory 303
common directory 304, 305
defaults directory 301
directory 301
mysql directory 303, 304

Ansible variables 296

dictionary variables 298
list variables 297
metadata, finding with Ansible
 facts 298, 299
registering 299
simple variables 297
values, sourcing 298

Apache 280**Apache and MySQL images**

building with Packer and Ansible
 provisioners, prerequisites 317, 318

Apache and MySQL playbooks

creating 316, 317

Apache license 116**API server** 129**Application Load Balancer (ALB)** 223**application migration, from virtual machines to containers** 22

application, containerizing 24
application requirement assessment 23
container infrastructure design 23
containerized applications, testing 24
deployment and rollout 25
discovery phase 22

application repository 53, 54**applications breaking into pieces approach**

benefits 29

applications, commonly deployed with containers

applications, following microservices architecture 28
applications, performing batch processing tasks or scheduled jobs 28
CI/CD tools 28
development and testing environments 28
Internet of Things (IoT) applications 28
machine learning and data
 analytics applications 28
stateful applications 28
web applications 28

apt package 315**Argo CD** 30, 379, 388, 389

Application 393, 394
ApplicationSet 394-396
architecture 389, 390
installing 390
Kubernetes manifests 393
setting up 390
Terraform configuration 391

Argo CD Web UI

accessing 396, 397

audit logging 488

-
- authentication** 488
 - setting up, with remote Git repository 40
 - authorization** 488
 - automatic sidecar injection**
 - enabling 484
 - automation** 463
 - significance 336, 337
 - Availability Zones (AZs)** 211
 - AWS Auto Scaling groups (ASGs)** 212
 - AWS AZs** 211
 - AWS CLI**
 - installing 214
 - AWS Code Commit** 367
 - AWS Code Deploy** 379
 - AWS ECS** 21
 - AWS EKS** 21
 - AWS Fargate** 211
 - AWS Lambda** 21
 - AWS Region** 211
 - AWS virtual private cloud (VPC)** 211
 - Azure** 129
 - for authentication and authorization 243-245
 - Azure ACS** 21
 - Azure AKS** 21
 - Azure Container Registry (ACR)** 119
 - Azure Defender** 416
 - Azure DevOps** 225, 367
 - Azure Functions** 21
 - Azure Kubernetes Service (AKS)** 209, 225
 - Azure resource group**
 - creating 249, 250
 - Azure RM** 257
 - Azure service principals** 243
 - Azure Storage backend**
 - using 257
 - Azure Storage resources**
 - creating 257, 258
 - Azure Terraform provider**
 - using 245, 246
 - Azure Virtual Network (VNet)** 280
 - Azure VMs** 280

B

- backend configuration**
 - creating, in Terraform 258-260
- best-effort controlled rollout** 174, 175
- best practices, Docker Compose** 87-89
- binaries** 34
- binary authorization** 434, 435
 - setting up 436-441
- bind mounts** 65
- Bitbucket** 39
- Blog App** 335, 338, 379, 414, 471, 472
 - CD workflow changes 431-434
 - CI pipeline, creating for 340-342
 - deploying 402-406
 - deployment configuration 380, 381
 - Frontend microservice 339
 - interactions 339
 - Posts Management microservice 338
 - Ratings Management microservice 339
 - Reviews Management microservice 338
 - testing, within CD pipeline 431
 - User Management microservice 338
- Blue/Green deployments** 22, 314, 378, 502
- Bootstrap** 339
- btrfs** 65
- build**
 - automating, with triggers 365, 367
- build performance**
 - best practices 367-370
- busybox session** 204

C

CaaS services 225, 226
cAdvisor
 configuring 78
calico 19
canary deployments 314, 378, 475
canary rollouts 502, 503
capacity planning 463
capital expenditure (CapEx) 7
CD models
 CI/CD workflow 376, 377
 complex deployment models 378
 simple deployment model 377
certificate authority (CA) 488
cgroups 15, 17
chaos engineering 464
Chef 277, 311
CI/CD pipelines 241
 securing 410-414
 testing 410-414
CI/CD workflow 376
CircleCI 53, 379
circuit breakers 475
circuit breaking 479
Citadel 478
Clair 416
Cloud Build 226, 367
Cloud Code 226
cloud computing 7, 239
 benefits 7, 8
 Infrastructure-as-a-Service (IaaS) 8
 Platform-as-a-Service (PaaS) 8
 Software-as-a-Service (SaaS) 8
cloud controller manager 129
Cloud Logging 226
Cloud Monitoring 226

Cloud Native Computing Foundation (CNCF) 127, 227
cloud service providers (CSPs) 7
CloudWatch
 container logs, querying from 218
cluster
 Jenkins, connecting with 355-357
ClusterIP Service resources 176-179
ClusterIP Services 380, 414, 471
ClusterRoleBinding resource 358
CMD directive 97, 98
code 34
 pulling 44, 45
 rebasing 44, 45
Code Build 367
code review 446
command-line interface (CLI) 389
communication and collaboration 6
complex deployment model, CD 378
 A/B testing 378
 Blue/Green deployments 378
 canary deployments 378, 379
config management 315
config map 147, 148
 example application 148-152
Configuration API Server 488
configuration as code (CaC) 6, 53
configuration management 276, 446
 challenges 277
 functioning 276, 277
Consul 257, 478
container architecture 15-17
Container-as-a-Service (CaaS) 21
containerd 132
Containerd 15
containerization 9
container log management 73

-
- container logs**
 - browsing, with ECS CLI 222, 223
 - container monitoring**
 - challenges 77, 78
 - container networking 18**
 - bridge network 18
 - host network 18
 - none 18
 - overlay 19
 - underlay 18
 - Container Network Interface (CNI) 219**
 - container registry 213**
 - containers 15, 19, 20, 30**
 - building 99-104
 - CI/CD pipeline example 21
 - modern DevOps practices 20-22
 - need for 11-13
 - removing 72
 - restarting 72
 - running 68
 - running, from versioned images 68, 69
 - troubleshooting 69-71
 - Containers as a Service (CaaS) 209**
 - containers, optimizing with distroless images 113**
 - cost 114, 115
 - performance 114
 - security 114
 - container template 356**
 - container vulnerability scanning 415**
 - continuous declarative IaC**
 - Environment repository, using 382
 - continuous delivery (CD) 5**
 - Continuous Deployment/
Delivery (CD) 409, 453**
 - advantages 375, 376
 - automation 374
 - significance 374
 - continuous integration and continuous delivery (CI/CD) pipelines 4**
 - continuous integration**
 - (CI) 5, 337, 373, 409, 453
 - continuous integration (CI) pipeline 335**
 - creating, with GitHub Actions 340-342
 - controller manager 129**
 - control plane, Istio 477, 478**
 - Citadel 478
 - Galley 478
 - Pilot 478
 - control plane nodes 128**
 - control plane traffic 477**
 - CoreOS Quay 119**
 - cos 257**
 - cost analysis 241**
 - Custom Resource Definitions (CRDs) 227, 485**
 - custom rollout strategies**
 - best-effort controlled rollout 174, 175
 - ramped slow rollout 172-174
 - CVS 34**

D

- daemon 69**
- Dagda 416**
- data centers 7**
- data plane, Istio 476, 478**
 - egress gateways 479
 - envoy proxies 478, 479
 - ingress gateways 479
- data plane traffic 477**
- denial-of-service attacks 479**
- Deployment resources 166-169, 359, 414, 471**
- development environment 46**
- devicemapper 65, 66**

- DevOps** 4, 338, 455
infinity loop 4
- DevOps team** 4
- DevSecOps** 409
- dictionary variables** 298
- directive** 97
- disaster recovery (DR)** 127, 462
- distributed application** 463
- DNS** 127
- Docker** 15, 19, 61, 376
installing 62, 63
layered filesystem 94-96
- Docker architecture** 92, 93
application code 92
container 92
dependencies 92
Docker Compose 92
Docker Engine 92
Dockerfile 92
image 92
networks 93
volumes 92
- Docker client** 94
- Docker Compose**
best practices 87-89
for declarative container management 83
sample application, deploying with 83-85
- docker-compose file**
creating 85-87
- Docker container metrics** 82, 83
- Docker containers** 19, 64, 340
running, in background 69
- Docker daemon** 19, 94
- Docker data storage options** 64
bind mounts 65
tmpfs mounts 65
volumes 64
- Dockerfile** 20, 97
creating 97
working 97
- DockerHub** 20, 93, 344, 346, 361, 376
- Docker images** 94
building 104
flattening 111-113
history 96
managing 105-111
- Docker logging**
best practices, to address challenges 76, 77
- Docker monitoring**
with Prometheus 77
- Docker registries** 94, 116
- docker run command** 68
- dockershim** 132
- Docker storage drivers** 65
btrfs 65
configuring 66, 67
devicemapper 65, 66
overlay2 65, 66
vfs 65
zfs 65
- Docker Trusted Registry** 116

E

- EC2 tasks**
scheduling, on ECS 217
- ECS CLI**
installing 214
used, for browsing container logs 222, 223
- ECS cluster**
spinning up 214, 215
- ECS services** 221
deleting 223
scheduling 221, 222

egress gateways 475, 480
Elastic Container Registry (ECR) 219
Elastic Container Service (ECS) 211
Elastic Kubernetes Service (EKS) 225
elastic network interface (ENI) 213
ELK 480
endpoints controller 129
end-to-end (E2E) development
 experience 225
ingress gateways 479
ENTRYPOINT 98
environment repository 53-57
Environment repository 381
 creating 382
 dev branch 382
 prod branch 382
 setting up 382-388
 used, for continuous declarative IaC 382
envoy proxies 478
envoy proxies, contribution to
 functionality of Istio
 network resiliency 479
 security and authentication 479
 traffic control 479
envoy proxy extensions 488
error budgets 460, 463
 versus SLOs 461
etcd 127, 130
Event Tracing for Windows (ETW) events 73
external secrets
 installing, with Terraform 425-427
External Secrets Operator 422, 423
 external secrets, accessing 422, 423
 installing 423, 424
 URL 422
 used, for generating MongoDB
 Kubernetes Secret 427-430

F

failover 479
Falco 416
Fargate tasks
 scheduling, on ECS 218-221
fault injection 479
filestores 194
fine-grained access control policies 476
firewalls 446
flannel 19
Flask 338
FQDN-based routing 187
FROM directive 97
frontend microservice 380
Fully Qualified Domain Name (FQDN) 117, 176
Function-as-a-Service (FaaS) 21

G

Galley 478
GCP 129
gcs 257
GCS bucket 386
generally available (GA) resource 135
geolocation-based routing 22
Gerrit 39
Git 10, 34
 code changes, staging 36, 37
 commit history, displaying 37, 38
 installing 35
 last commit, amending 38, 39
 local repository, connecting to
 remote repository 41-44
 versus GitOps 58, 59
GitBash 33

- Git branches** 46
 creating 46
 managing 47
- Gitflow** 55
 structure 55, 56
- GitHub** 39, 339
 flow 56
- GitHub Actions** 335, 339, 377
 CI pipeline, creating 340-342
 community actions 340
 continuous delivery (CD) 340
 continuous integration (CI) 340
 custom scripts 340
 integration 340
 multi-platform support 340
 scheduled jobs 340
 workflow automation 340
- GitHub Actions workflow**
 creating 343-349
- GitHub flow** 55
- GitHub repository**
 creating 342
- GitHub secrets** 344
- GitOps** 33, 51, 241, 335
 benefits 51, 52
 principles 52
 pull model 53
 push model 53
- Git repository** 34
 application repository 54
 environment repository 56, 57
 initializing 35
 structuring 54
- GKE instance** 384
- Golang** 105
- Google Cloud Platform**
 (GCP) 7, 162, 409, 468
- Google Cloud Run** 21, 209, 226
- Google Cloud Secret Manager**
 secret, managing 421, 422
- Google Cloud Storage (GCS)** 119
- Google Container Registry**
 (GCR) 63, 119, 416
- Google Functions** 21
- Google Kubernetes Engine**
 (GKE) 162, 209, 226, 352, 409, 468
 spinning up 162, 229, 352
- Google Pub/Sub** 227
- Google Secrets Manager** 489
- Grafana** 227, 480, 507, 509
 for alerting 510-512
- Graphviz** 271
- Graylog Extended Log Format**
 (GELF) endpoint 73
- gRPC** 479
- H**
- hard disks** 194
- Hardware Security Modules (HSMs)** 421
- HashiCorp** 311
- HashiCorp Configuration Language**
 (HCL)-based interface 241
- HashiCorp Vault** 421
- Helm chart** 423
- high availability (HA)** 127, 211
- highly available NGINX container**
 best setting 71, 72
- HorizontalPodAutoscaler (HPA)** 230
- Horizontal Pod autoscaling** 189-192
- horizontal scaling** 217
- host metrics** 82
- http** 257
- HTTP** 479
- HTTPS** 40

hybrid model 54
hypervisor 14

I

Identity Access Management (IAM) 212, 386

immutable infrastructure

- cons 315
- pros 314, 315
- using, scenarios 313
- with Packer 310, 311

incident management 464

Infrastructure-as-a-Service (IaaS) 8

Infrastructure as Code

- (IaC) 5, 56, 215, 239, 240, 241, 369

ingress gateways 475, 479, 481

ingress resources 182-186

- name-based routing 187, 188
- path-based routing 186, 187

init containers 144-146

instance profile 212

instances 212

instant messaging (IM) tools 6

internet gateway 212

Internet Service Provider (ISP) 458

I/O operations per second (IOPS) 64

Istio 22, 170, 227, 463, 467, 475

- developer-friendly 476
- for traffic management 497-501
- installing 480-483
- observability 476
- security 476
- traffic management 475
- used, for securing microservices 487-489

Istio architecture

- control plane 477, 478
- data plane 476, 478

istio-base 481
istioctl component 480
istiod component 478, 481
Istio ingress
 using, to allow traffic 485-487

J

Jenkins 10, 30, 53, 335, 349-351, 377

- connecting, with cluster 355-357
- installing 357-361

Jenkins CaC (JCacC) file

- creating 353

Jenkins Configuration as Code feature 353

Jenkins Global Security 353

- configuring 353, 354

Jenkins job

- running 361-364

JFrog Artifactory 63

Jinja2 markup 296

Jinja2 templates

- registering 300

K

Kafka 227

Kaniko 335, 350, 352

Kaniko executor script 361

key management solution 421

key performance indicators (KPIs) 453

Kiali 507

Kiali dashboard

- accessing 508

Knative 209, 211

- architecture 227, 228

- installing 229-231

- open source CaaS 226, 227

- Python Flask application,
deploying on 231-233
- Python Flask application, load
testing on 233, 234
- Knative project** 226
- Kritis**
- URL 435
- kubeadm** 130
- kube-apiserver** 127
- Kube API server** 129
- kubectl** 127, 131
- kubectl apply** 204
- kubectl bash completion**
using 205
- kubectl delete** 204
- kubectl --dry-run**
using 203, 204
- kubelet** 127, 130
- kube-proxy** 130
- Kubernetes** 21, 127, 257, 350, 377
need for 127, 128
- Kubernetes API server** 358
- Kubernetes architecture** 128, 129
- Kubernetes cluster** 352
- Kubernetes Deployments** 163
Deployment resources 166-169
ReplicaSet resources 164-166
- Kubernetes Deployment strategies** 169
A/B testing 170
blue/green 169
canary 170
Recreate 169-171
RollingUpdate 169-172
- Kubernetes Engine** 21
- Kubernetes in Docker (KinD)** 130
installing 132-134
- Kubernetes manifests** 135
- Kubernetes Secret resource** 359, 414, 471
- Kubernetes service account resource** 358
- Kubernetes Services** 175
ClusterIP Service resources 176-179
LoadBalancer Service resources 181, 182
NodePort Service resources 179-181
- Kube scheduler** 129
- L**
- launch configuration** 212
- layered filesystem** 94-96
- Linkerd** 170, 463
- Linux Apache MySQL and PHP
(LAMP) stack** 309, 316
- listener** 224
- list variables** 297
- liveness probes** 141
- live traffic mirroring** 475
- load balancers** 126, 128, 129
LoadBalancer Service 359, 380, 414, 471
LoadBalancer Service resources 181, 182
- load balancing** 223
- local repositories** 39
- logging drivers** 73
configuring 74-76
- log management** 73
- M**
- MagicDNS solution** 228
- manifests/argocd directory** 393
- man-in-the-middle attacks** 479
- matrix of hell scenario** 13
- Mercurial** 34
- merge conflict** 44
- microservice** 463
securing, with Istio 487-489
- microservices architecture** 9

Minikube 130

installing 131

modern cloud-native applications 9

automation 9

containerization 9

DevOps practices and tools 9

dynamic orchestration 9

microservices architecture 9

usage of cloud-native data services 9

modern DevOps

versus traditional DevOps 10, 11

module repository 253

MongoDB 338, 339, 380, 414, 471

MongoDB Kubernetes Secret

generating, with External Secrets

Operator 427-430

monitoring and logging 6

multi-stage builds 106-108, 368

mutable infrastructure 311

cons 314

pros 314

mutual TLS and encryption 488

mutual TLS and traffic encryption 479

MySQL 280

N

name-based routing 187, 188

namespaces 178

NAT rule 327

Network Address Translation (NAT) 18

Network Interface Card (NIC) 324

Network Load Balancer (NLB) 223

NGINX 97

nginx ingress controller 183

node controller 129

node exporter

configuring 79

NodePort Service resources 179-181

Nomad 478

O

OAuth scope 384

on-call rotations 464

Open Authorization (OAuth) tokens 420

OpenSCAP 416

open source CaaS

with Knative 226, 227

OpenVZ 15

Oracle Cloud Infrastructure Registry 119

Oracle Functions 21

Oracle OCI 21

Oracle OKE 21

Oracle VirtualBox 14

oss 257

overlay2 65, 66

overlay network 127

P

Packer 309

build process 313

custom image, building 312

features 312

for immutable infrastructure 310, 311

installing 315

Packer configuration

defining 318-321

Packer workflow

for image building 321-323

PagerDuty 513

path-based routing 186, 187

persistent disks 128, 129

PersistentVolume 359, 380, 414, 471

PersistentVolumeClaim 358, 380, 414, 471

PersistentVolumes

- dynamic provisioning 199-202
- managing 194
- static provisioning 195-199

personal package archive (PPA) 283**pg** 257**Pilot** 478**Platform-as-a-Service (PaaS)** 8**pod multi-container design patterns** 143

- adapter pattern 156-158
- ambassador pattern 146, 147
- sidecar pattern 152, 153

pod reliability

- ensuring 140

pods 134, 211, 405

- deleting 135
- running 134-137
- troubleshooting 138-140

policy as code 241**port forwarding**

- using 137, 138

post-commit webhook 365**post-incident reviews (PIRs)** 464**principle of least privilege
(PoLP)** 212, 243, 446**private Docker registry**

- hosting 116-118

private key 489**private module** 241**probes**

- liveness probes 141
- readiness probes 141
- startup probes 141
- working 142, 143

Prod CD workflow 444**production environment** 46**project ID** 386**Prometheus** 77, 227, 480, 507

- configuring, to scrape metrics 79
- for Docker monitoring 77
- installing 78

**Prometheus Query Language
(PromQL)** 77, 510**public key** 40**public key infrastructure (PKI)** 435, 489**public registries** 118, 119**pull model** 53**pull request-based gating** 56**pull request gating** 441, 442**pull requests** 48

- working with 48-50

Puppet 277, 311**push model** 53**Python Flask application**

- building 231, 232

- deploying, on Knative 231-233

- load testing, running on Knative 233, 234

Q**quality assurance (QA) team** 4, 435**R****ramped slow rollout strategy** 172-174**readiness probes** 141**rebasin** 44**Recovery Point Objective (RPO)** 462**Recovery Time Objective (RTO)** 462**Recreate strategy** 170, 171**Red/Black deployments** 378**Red Hat** 277**release gating phase** 443**Remote** 257**remote backends** 256

-
- remote Git repository**
 authentication, setting up with 40
 creating 40
- remote repositories** 39
- ReplicaSet resources** 164-166
- ReplicaSets** 221
- replication controller** 129
- resource groups**
 cleaning up 267
- resources**
 cleaning up 271
 inspecting 264, 265
- retries** 475, 479
- return on investment (ROI)** 29
- reusability**
 designing for 296
- Rkt** 15
- role-based access control (RBAC)** 54, 212
- RollingUpdate strategy** 171, 172
- root cause analyses (RCAs)** 464
- root certificate** 489
- route table** 212
- RUN directives** 97, 98
- S**
- S3** 257
- SaltStack** 277, 311
- sample application**
 deploying, with Docker Compose 83-85
- sample container application**
 launching 79-81
- Scalable Jenkins** 349-352
- schedulers** 130
- Sealed Secrets** 398
 components 398
 creating 401, 402
 kubeseal, installing 401
- operator, installing 399, 400
 workflow 399
- SealedSecrets** 381, 419
- Secret** 153, 471
 baseline, setting up 424
 creating, in Google Cloud Secret Manager 421, 422
 example application 153-156
 managing 420
- Secret Manager** 421
- Secret resource** 414
- secrets management solution** 421
- secure ingress gateways**
 creating 489-491
- Secure Shell (SSH)** 277
- security and testing best practices, modern DevOps pipelines** 445
 access control, establishing 446
 all aspects of application, testing 447
 application, monitoring and observing 448
 chaos engineering, implementing 448
 DevSecOps culture, adopting 446
 documentation and knowledge sharing 448, 449
 effective testing, in production 448
 security automation 447
 security risks, managing consistently 446
 shift left, implementing 446
 test automation, within CI/CD pipelines 447
 test data, managing effectively 447
 vulnerability scanning, implementing 446
- sensitive configurations**
 managing 398
- Sentinel** 241
- server certificate** 489
- serverless offerings**
 need for 210
- Service** 471

- service account 129, 384, 386
Service A pod 480
Service B pod 480
service-level agreements
 (SLAs) 8, 377, 456, 459, 460
service-level indicators (SLIs) 456-458, 463
service-level objectives
 (SLOs) 377, 456-459, 463
service mesh 22, 472-474
 TLS, enforcing within 491-497
services 211
sidecar pattern 152, 153
sidecar proxies 488
simple deployment model, CD 377
simple variables 297
single pane of glass (SPOG) 222
single source of truth (SSOT) 285
single-stage builds 105, 106
site reliability engineering
 (SRE) 455, 456, 467
Slack 368
Software-as-a-Service (SaaS) 8, 39, 416
software development 34
software development life cycle
 (SDLC) process 337
software reliability
 significance 453-455
Sonatype Nexus 63
source code 34
source code management 34
 tools 34
Spinnaker 379
SRE approach, to running distributed application in production
 automated remediation 463
 capacity planning 463
 chaos engineering 464
 continuous improvement 464
documentation and knowledge sharing 464
error budgets 463
monitoring and alerting 463
on-call and incident management 464
SLIs 463
SLOs 463
SSDs 194
SSH 40
SSH client 282
SSH key pair 40
sslip.io 228
staged rollouts 475
staging environment 46
Standard Operating Procedure (SOP) 446
startup probes 141
state files
 inspecting 266
stateful applications 26
 managing 192, 193
StatefulSet 405, 471
StatefulSet resources 193, 194, 414
stateless applications 26
storage driver
 configuring 66, 67
subnet 211, 280
subnet route table associations 212
Subversion 34
system administrators (SysAdmins) 455
systemd service 130

T

- target group 224
tasks 211, 213, 216
 scaling 217
 stopping 218
task scheduler 213
TCP 479

telemetry data collection 480
Terraform 22, 53, 239, 382
 backend configuration, creating 258-260
 dependencies 270
 external secrets, installing with 425-427
 graph 271
 installation link 242
 installing 242, 243
 required infrastructure creating
 with 324-331
 variable values, providing within 247, 248
terraform apply command 251, 252
terraform console command 270
Terraform Core 241, 242
terraform destroy command 252, 253
terraform directory 391
 app.tf 392
 argocd.tf 391, 392
 provider.tf 392
terraform fmt command 250
terraform init command 249
Terraform modules 253-256
terraform output command 267
terraform plan command 251
Terraform providers 241-243
Terraform state
 current state, viewing 268
 existing resources, importing into 269, 270
 managing 256, 257, 268
 resources, listing in current state 268
 resources, removing from state 269
terraform validate command 250
Terraform variables 246, 247
Terraform workflow 248, 249
Terraform workspaces 260-263
test suite 431
timeouts 475

TLS
 enforcing, within service mesh 49-497
TLS-enabled ingress gateways 489
tmpfs mounts 65
toil 374, 431, 456
token controllers 129
traditional DevOps
 versus modern DevOps 10, 11
Traefik 170
traffic encryption 476
traffic management
 features 497
 reference link 507
 with Istio 497-501
traffic mirroring 22, 504, 506
traffic tracing 476
Travis CI 53
triggers
 build, automating with 365, 367

U

Ubuntu 18.04 Bionic LTS 125
Ubuntu Virtual Machine (VM) 261

V

variable values
 providing, within Terraform 247, 248
versioned images
 containers, running from 68, 69
vfs 65
virtualized infrastructure 30
virtual machine 14, 29, 310
virtual machine, to container migration
 assessment 27, 28
VM scale set 324

VMWare 14
volumes 64, 128
 features 64
 mounting 65
vulnerability testing 446
vxlan 19

W

WebSockets 479
worker nodes 128

Z

zfs 65



www.packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

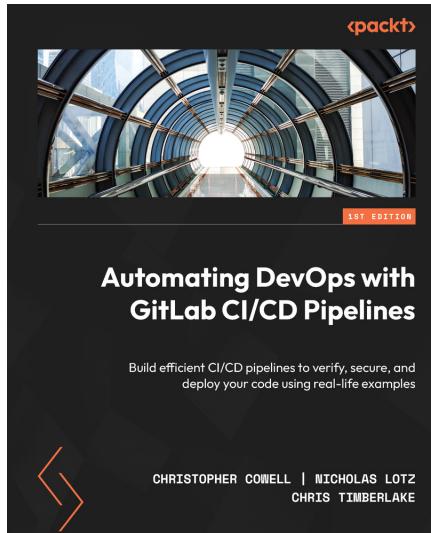


SAFe® for DevOps Practitioners

Robert Wen

ISBN: 9781803231426

- Understand the important elements of the CALMR approach
- Discover how to organize around value using value stream mapping
- Measure your value stream using value stream metrics
- Improve your value stream with continuous learning
- Use continuous exploration to design high-quality and secure features
- Prevent rework and build in quality using continuous integration
- Automate delivery with continuous deployment
- Measure successful outcomes with Release on Demand



Automating DevOps with GitLab CI/CD Pipelines

Christopher Cowell, Nicholas Lotz, Chris Timberlake

ISBN: 9781803233000

- Gain insights into the essentials of Git, GitLab, and DevOps
- Understand how to create, view, and run GitLab CI/CD pipelines
- Explore how to verify, secure, and deploy code with GitLab CI/CD pipelines
- Configure and use GitLab Runners to execute CI/CD pipelines
- Explore advanced GitLab CI/CD pipeline features like DAGs and conditional logic
- Follow best practices and troubleshooting methods of GitLab CI/CD pipelines
- Implement end-to-end software development lifecycle workflows using examples

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Modern DevOps Practices*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click [here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805121824>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly