Modern DevOps Practices

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

Gaurav Agarwal

BIRMINGHAM—MUMBAI

Table of Contents

2

Source Code Management with Git and GitOps 33

3

Containerization with Docker 61

4

Creating and Managing Container Images 91

Part 2: Container Orchestration and Serverless

5

Container Orchestration with Kubernetes 125

6

Managing Advanced Kubernetes Resources 161

7

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

Part 3: Managing Config and Infrastructure

8

Infrastructure as Code (IaC) with Terraform 239

9

Configuration Management with Ansible 275

10

Immutable Infrastructure with Packer 309

Part 4: Delivering Applications with GitOps

11

Continuous Integration with GitHub Actions and Jenkins 335

12

Continuous Deployment/Delivery with Argo CD 373

13

Securing and Testing Your CI/CD Pipeline 409

Part 5: Operating Applications in Production

14

Understanding Key Performance Indicators (KPIs) for Your Production Service 453

15

Implementing Traffic Management, Security, and Observability with Istio 467

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.

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:

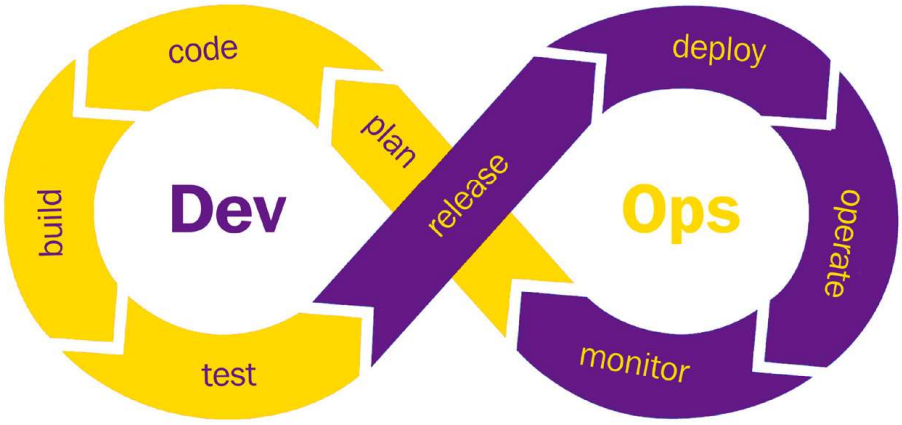

Figure 1.1 – DevOps infinity loop

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:

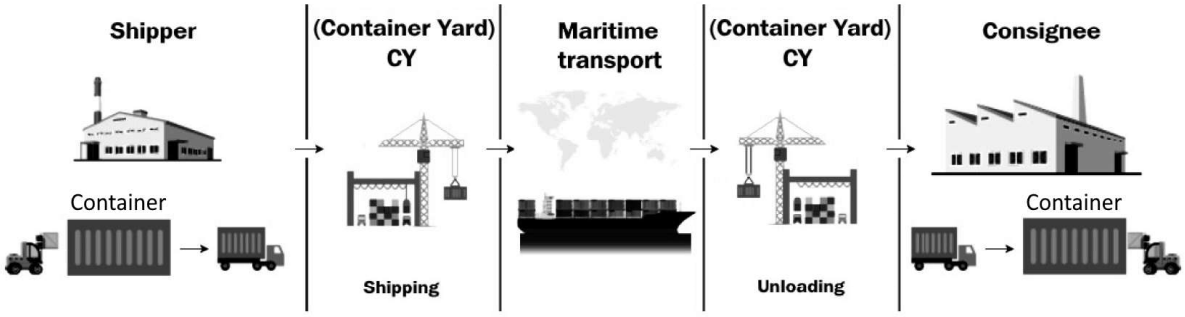

Figure 1.2 – Shipping container workflow

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:

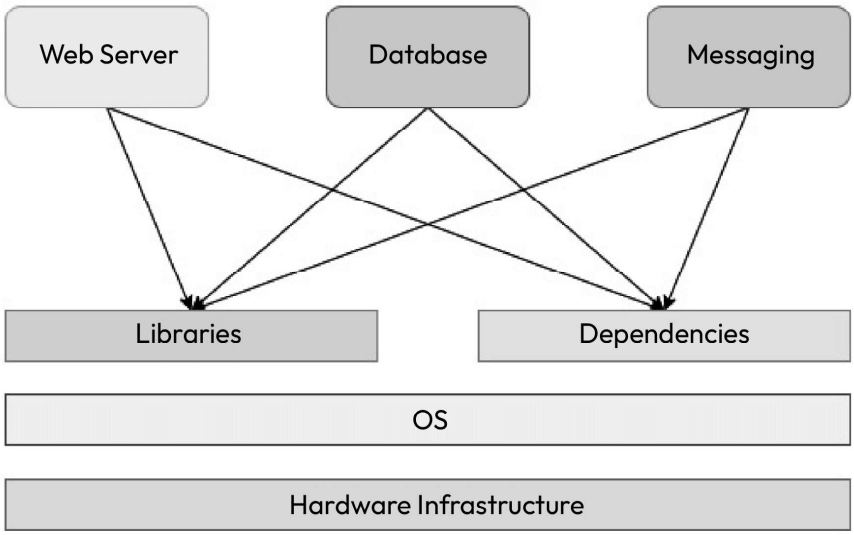

Figure 1.3 – Applications on a physical server

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:

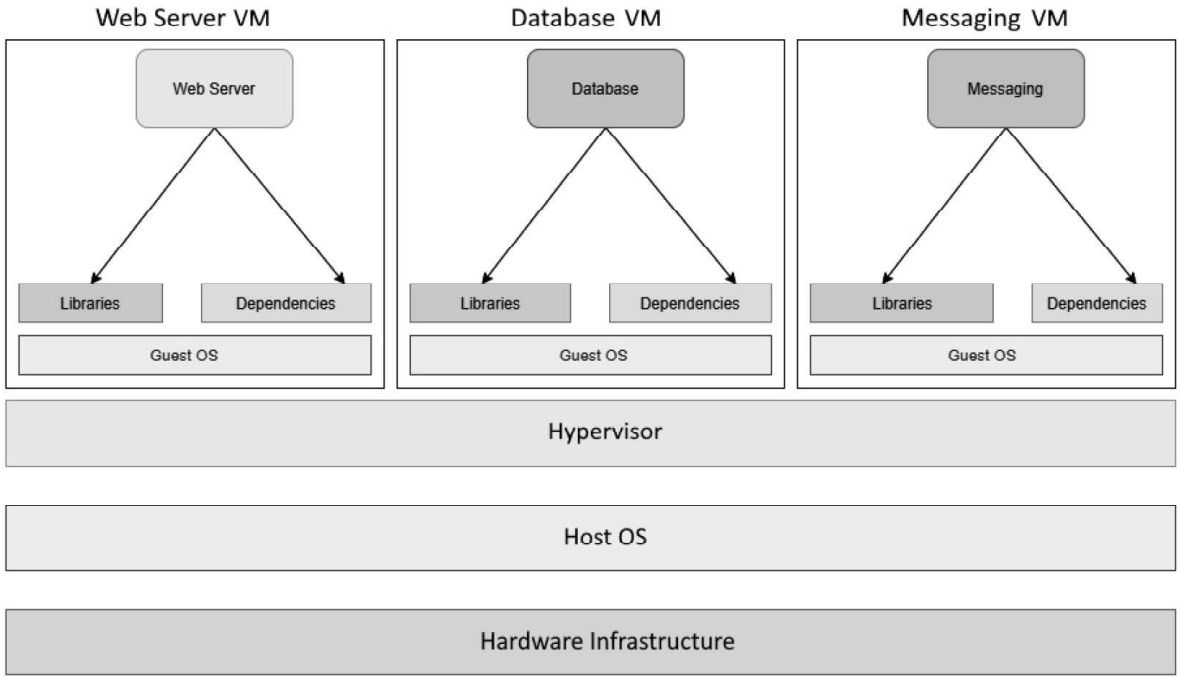

Figure 1.4 – Applications on virtual machines

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:

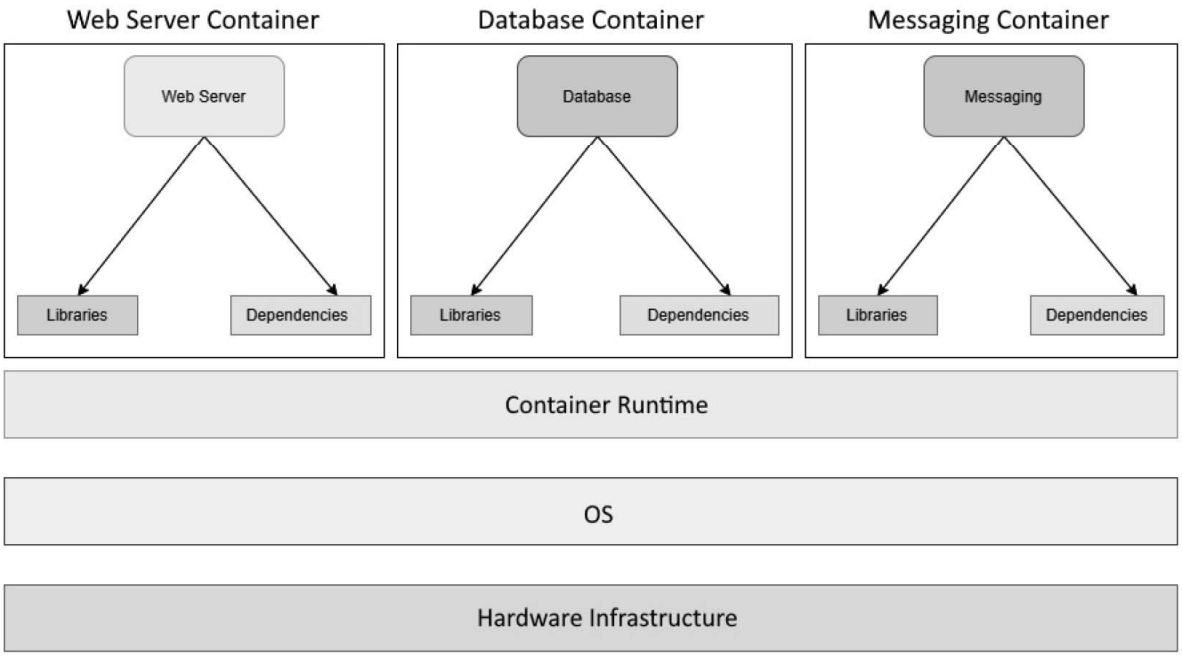

Figure 1.5 – Applications on containers

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:

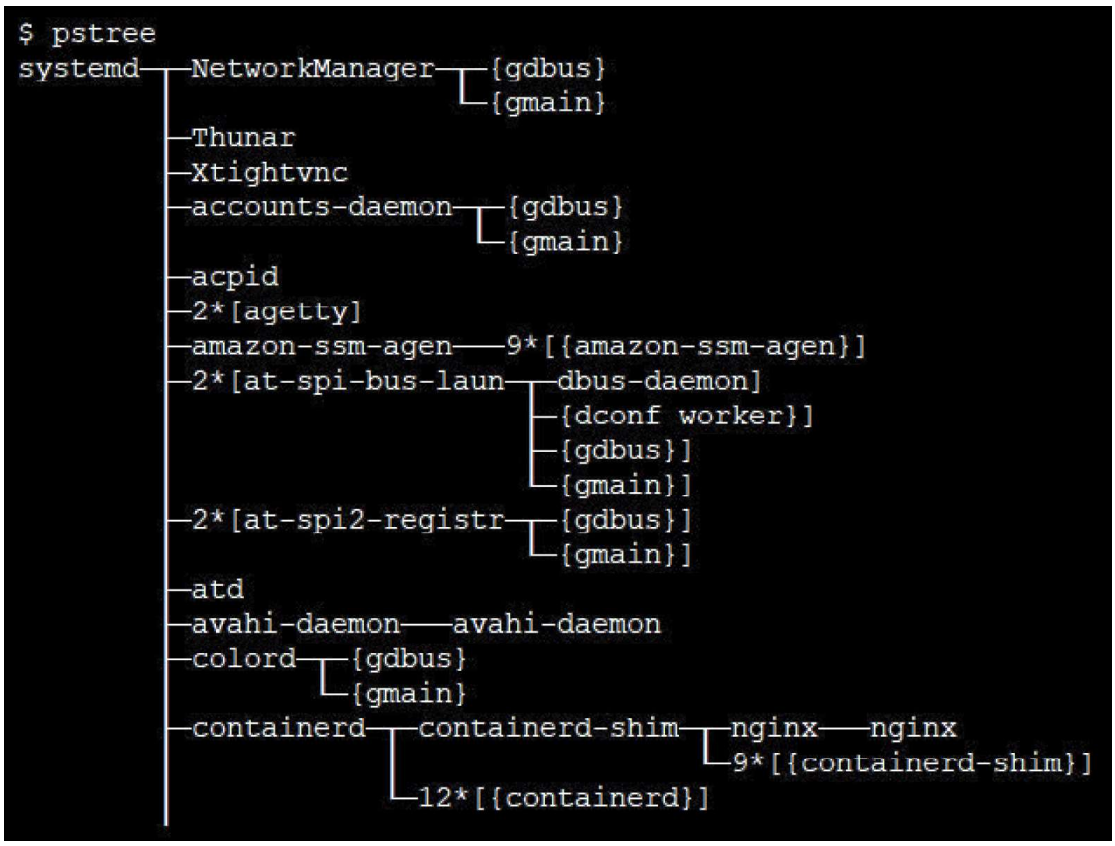

Figure 1.6 – OS processes

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.