

Kubernetes and Cloud Native Associate (KCNA) Exam Guide

**KCNA concepts featuring orchestration, architecture,
observability, and application delivery**



Sangram Rath

bpb



Kubernetes and Cloud Native Associate (KCNA) Exam Guide

KCNA concepts featuring orchestration, architecture,
observability, and application delivery



Kubernetes and Cloud Native Associate (KCNA)

Exam Guide

*KCNA concepts featuring orchestration,
architecture, observability, and application
delivery*

Sangram Rath



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

ISBN: 978-93-65892-116

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:



www.bpbonline.com

Dedicated to

The Readers

About the Author

Sangram Rath is an independent Cloud Architect and Technology Advisor specializing in virtualization, cloud computing, and cloud-native platforms with a focus on open-source technologies and a growing interest in security and AI. He also has an extensive training background, delivering sessions on these technologies to individuals and corporates, including fortune 500 ones.

He is the author of Hybrid Cloud Management using RedHat CloudForms (Packt, 2015), has reviewed courses such as Automation Solutions with Chef Automate (Packt, 2020), Google Cloud Platform All-In-One Guide (BPB, 2022), and has authored several proprietary courses on cloud technologies for training companies. He also ghostwrites technical articles and blogs.

He advocates open source, cloud, and cloud-native technologies through speaking opportunities and has spoken at various colleges and conferences such as Open Source Summit NA & Europe, Google DevFest, IIIT Bhubaneswar (TensorFlow UG), Open Source India, GlobalAzure, DigitalOcean to name a few.

Sangram has certifications from Microsoft, AWS, Google, Linux Foundation, and more across cloud, non-cloud, and cloud-native technologies.

About the Reviewers

- ❖ **Omar Alayli** is a Network and Security Specialist with Google Cloud, Qatar, focusing on Infrastructure Modernization, Networking, and Security. He holds several certifications in Networking and Security, including Google Professional Network Engineer (PCNE) and Google Professional Security Engineer (PCSE). Prior to joining Google, Omar was a Senior Cloud Architect with iSolution, one of Google Cloud's Premier Partners in the MENA region. Omar holds several industry specifications, including CISSP, CCSP, CCSK, Security+, PenTest+, CASP+, Cysa+, and KCNA. He also holds an MSc in Telecommunications and Software from the University of Surrey (UniS), UK, and an Executive MBA from ESCP Europe, Paris, France. His passion, life, and hobby is cybersecurity!
- ❖ **Samarth Shah** is a highly accomplished Tech Lead with almost 10 years of experience at industry-leading companies like Microsoft and Google. At work, he routinely solves hard problems in distributed systems and helps democratize cloud technologies for people around the globe. Currently, he spearheads initiatives within Google's BigQuery, focusing on petabyte-scale software solutions. Samarth has a proven track record of building massively scalable, distributed, and innovative products for both Azure and Google Cloud, generating millions in revenue for these organizations. His expertise lies in cloud infrastructure, databases, and compute, where he excels at designing and implementing high-impact solutions within the cloud computing domain.

Acknowledgement

I would like to express my gratitude to all those who contributed to the completion of this book.

I thank BPB Publications for bringing this book to fruition. Their assistance helped in navigating some of the complexities of the publishing process.

I would also like to acknowledge the reviewers, technical experts, and editors who provided feedback and contributed to the refinement of this manuscript. Their suggestions have helped enhance the quality of the book.

Last but not least, I want to express my gratitude to the readers who have shown interest in the book. Your support and encouragement is deeply appreciated.

Thank you to everyone who has played a part in making this book a reality.

Preface

Understanding the fundamental concepts is important in today's rapidly evolving world. This book, **Kubernetes and Cloud Native Associate (KCNA) Exam Guide**, covers some of these fundamental concepts that make up the cloud-native ecosystem. Consisting of ten chapters, this book covers all the exam domains of the KCNA exam and more. Additionally, each chapter includes practice questions to reinforce the learnings.

We start with an introduction to application architecture patterns, computing fundamentals, and cloud-native in [Chapter 1](#), creating the first step to understanding containers, Kubernetes and cloud-native. From there, we learn about containers and the need for container orchestration in [Chapter 2](#), creating the foundation for Kubernetes. [Chapter 3](#) provides a set of practical exercises on Docker and Containerd, reinforcing the theoretical knowledge gained in the earlier chapter.

[Chapter 4](#) focuses on Kubernetes basics such as architecture, resources, scheduling and API. In [Chapter 5](#), we take a step further to learn how security, networking and storage works in Kubernetes, providing the complete container orchestration lifecycle. We also learn about Service Mesh.

In [Chapter 6](#), we focus on the properties of a cloud-native architecture, cloud-native security and serverless. We also learn about the Cloud Native Community, how its governed and job roles pertaining to it.

[Chapters 7](#) and [8](#) cover cloud-native observability and Cloud Native Application Delivery respectively. Together, through these two chapters, we cover telemetry, monitoring and observability, observability tools, cost management, CI/CD, GitOps, and GitOps tools.

[Chapter 9](#) is again a hands-on chapter that reinforces the theoretical knowledge on Kubernetes through practical exercises.

Finally, [Chapter 10](#) covers information about the exam, which is essential to understand the KCNA exam logistics.

This book is designed to cater to students and professionals irrespective of their Kubernetes and cloud-native experience.

By focusing on the fundamentals and keeping in line with the exam objectives, this book aims to equip readers with a solid understanding of Kubernetes and cloud-native concepts. Whether you are a novice or an experienced learner, I hope this book will serve as a valuable resource in your exam and cloud native journey.

[Chapter 1: Stepping up to Kubernetes and Cloud-Native](#) - This chapter acts as a warmup session covering concepts such as monolithic and microservices architecture, virtualization, cloud computing and containers that are important to map the world of Kubernetes and cloud-native. It also introduces Kubernetes and its origins, cloud native and its landscape, and cloud-native certifications. It also provides certain insights into the adoption of Kubernetes and cloud-native technologies and their importance as a skill for career growth.

[Chapter 2: Understanding Containers and the Need for Container Orchestration](#) - The second chapter goes a bit deeper into containers, how they work and the need for container orchestration. It covers Docker and container concepts, such as images, registry, runtime, including how storage, networking and security works on standalone containers, highlighting the challenges and paving the way for an introduction to container orchestration. Following this, the chapter covers a typical container orchestraor architecture, workflow, its functions and advantages. The chapter also covers open standards that are important to make everything work, followed by an introduction to the Kubernetes as a container orchestrator.

Chapter 3: Hands-on Docker and Containerd - This chapter provides hands-on exercises covering Docker installation, creating a Dockerfile, and running containers. Exercises also cover image registry, Docker hub, working with Docker volumes, and networking. The second part of the chapter covers similar exercises for Containerd, which is the default runtime for Kubernetes.

Chapter 4: Kubernetes Basics - This chapter covers the first exam domain, Kubernetes Fundamentals, covering Kubernetes architecture and the components that make up the control plane and the worker nodes, Kubernetes resources, scheduling, and the Kubernetes API. It also covers different Kubernetes installation methods.

Chapter 5: Container Orchestration with Kubernetes - This chapter covers the second exam domain, container orchestration, and covers how to secure, network, and store data in Kubernetes clusters. It covers Kubernetes resources or API objects that provide these capabilities and Service Mesh, an important technology concept for running microservices in a cloud-native model.

Chapter 6: Cloud-Native Architecture - This chapter covers cloud-native architecture fundamentals and the tenets of one, such as scalability, loose-coupling, resiliency, autoscaling, security, etc. The chapter covers how to implement these in Kubernetes using first class resources or external tools. The chapter also covers Serverless. Finally, it revisits community and governance, open standards, and cloud-native roles, covering the third exam objective, cloud-native architecture.

Chapter 7: Cloud-Native Observability - This chapter explores what cloud-native observability is, covering topics such as telemetry, instrumentation, monitoring, metrics, logging, and events. It provides information on the native observability options in Kubernetes and then introduces cloud-native observability tools designed for it, such as Prometheus, Grafana, Jaeger and more. This chapter also covers cost

management, thus addressing the topics in the exam domain cloud-native observability.

Chapter 8: Cloud-Native Application Delivery - This chapter covers topics from the Cloud Native Application Delivery exam domain such as CI/CD, GitOps and GitOps tools like Flux and ArgoCD.

Chapter 9: Hands-on Kubernetes - This chapter teaches how to install Kubernetes using minikube, understand the environment, create and manage Kubernetes resources, basic operations, and troubleshoot through a series of hands-on exercises. This is not a requirement for the exam but greatly reinforces the learnings.

Chapter 10: About the Exam - This chapter provides information about the KCNA exam, such as the duration, number of questions, passing score, and other facts. It also includes information on how to prepare for taking the exam (before, during, and after), how to book the exam, the proctoring method, and more.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/5f4120>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Kubernetes-and-Cloud-Native-Associate-Exam-Guide>.

In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.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 BPB books and eBooks.

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 business@bpbonline.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 www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from?

Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Stepping up to Kubernetes and Cloud-Native

Introduction

Structure

Application architectures

Monolithic

Microservices

Monoliths or microservices

Computing fundamentals

Virtualization

Cloud computing

Containers

Virtualization versus containers

Future of applications and computing

Kubernetes and cloud-native

A brief history of Kubernetes

Cloud-native basics

Cloud Native Computing Foundation

Cloud-native landscape

State of Kubernetes and cloud-native

CNCF certifications

Career opportunities

Conclusion

Multiple choice questions

Answers

2. Understanding Containers and the Need for Container Orchestration

Introduction

Structure

Objectives

Containers

Docker

Container images

Container registry

Container runtime

Container creation workflow

Container storage

Container networking

Container security

Container orchestration

Container orchestration fundamentals

Common container orchestrators

Container orchestrator architecture

Container orchestration workflow

Functions of a container orchestrator

Benefits of container orchestration

Open standards

Open Container Initiative

Container Runtime Interface

Container Storage Interface

Container Network Interface

Service Mesh Interface

Introduction to Kubernetes

Runtime

Containerd runtime

CRI-O runtime

rkt container runtime
Storage
Networking
Security
Managed Kubernetes platforms
Multi-cloud/Hybrid Kubernetes
Lightweight Kubernetes variants

Conclusion
Multiple choice questions
Answers

3. Hands-on Docker and Containerd

Introduction
Structure
Objectives
Hands-on Docker

Setting up Docker
Prerequisites
Creating a Dockerfile
Creating and running a container
Additional container operations
Working with images and image registry
Persisting data with Docker volumes
Persisting data with bind mount
Networks in Docker
Creating multiple containers on the same network
Cleanup

Hands-on containerd

Prerequisites
Installing containerd
Working with container images

Creating and managing containers

Storage and networking using the ctr utility

The nerdctl utility

Conclusion

Multiple choice questions

Answers

4. Kubernetes Basics

Introduction

Structure

Objectives

Kubernetes architecture

Installing Kubernetes

The kubectl utility

Control plane

API server

Scheduler

Controllers

etcd store

Cloud controller manager

Worker node

kubelet

kube-proxy

Container runtime

Kubernetes resources

Namespaces

Pods

Types of containers in Pods

Deployments

StatefulSets

DaemonSets

Jobs and CronJobs

Resource requests and limits

Storage and networking resources

Creating resources

Manifests

kubectl examples

Scheduling in Kubernetes

Basic scheduling

Advanced scheduling

nodeSelector

Affinity and anti-affinity

nodeName

Taints and tolerations

Kubernetes API

Conclusion

Multiple choice questions

Answers

5. Container Orchestration with Kubernetes

Introduction

Structure

Objectives

Security

The 4 Cs of Cloud Native Security

Cluster security

Role-based access control

Service accounts

Network policies

ConfigMaps

Secrets

Networking

Service

ClusterIP

NodePort

Load balancer

Headless service

DNS

Ingress

Service mesh

Istio

Linkerd

Consul

Traefik Mesh

Storage

StorageClass

PersistentVolume

PersistentVolumeClaim

Conclusion

Multiple choice questions

Answers

6. Cloud-Native Architecture

Introduction

Structure

Objectives

Cloud-native architecture fundamentals

Cloud-native architecture

The Twelve-Factor App

Key tenets of a cloud-native architecture

Scalable

Loosely coupled

Resilient

Observable

Automated

Resiliency in Kubernetes

Deployments

ReplicaSets

StatefulSets

Probes

Storage resiliency

Network resiliency

Autoscaling in Kubernetes

Horizontal Pod Autoscaler

Vertical Pod Autoscaler

Cluster Autoscaler

Kubernetes Event-driven Autoscaling

Cloud-native security

4 Cs of Cloud Native Security

Zero trust

Defense-in-depth

Security in Kubernetes

Serverless

Benefits

Functions-as-a-Service

Notes of caution

Managed versus self-managed serverless

Open source serverless solutions in Kubernetes

Managed serverless platforms

Community and governance

Open standards

Cloud-native personas

Cloud architect

Cloud solutions architect

Cloud engineer
DevOps engineer
Site Reliability Engineer
Security engineer
DevSecOps engineer
Data engineer
Full stack developer

Conclusion

Multiple choice questions

Answers

7. Cloud-Native Observability

Introduction

Structure

Objectives

Telemetry, monitoring and observability

Telemetry

Instrumentation

Monitoring

Observability

Cloud-native observability

Native observability in Kubernetes

Metrics and monitoring

Logging

Events

Open-source observability tools

Prometheus

Prometheus metric types

Grafana

Jaeger

Elasticsearch, Logstash and Kibana

Fluentd

Thanos

OpenTelemetry

Cost management

FinOps

Conclusion

Multiple choice questions

Answers

8. Cloud-Native Application Delivery

Introduction

Structure

Objectives

Cloud-native application delivery

CI/CD

Continuous integration

Continuous delivery

Continuous deployment

GitOps

Infrastructure as Code

GitOps concepts

GitOps workflow

GitOps tools

Flux

ArgoCD

Conclusion

Multiple choice questions

Answers

9. Hands-on Kubernetes

Introduction

Structure

Objectives

Creating a Kubernetes cluster using minikube

Prerequisites

Install Docker

Installing minikube

Install kubectl

Creating a Kubernetes cluster using Play with Kubernetes

Understanding the environment

Creating resources in Kubernetes

Creating your first Pod

Deployments

Services

Basic operations and troubleshooting

Displaying resources

Working with namespaces

Describing resources

Viewing logs

Viewing events

Persistent storage in Kubernetes

Creating a Persistent Volume

Creating a Persistent Volume Claim

Updating the deployment to use the volume

Verifying data persistence

StatefulSets and DaemonSets

StatefulSets

DaemonSets

Conclusion

Multiple choice questions

Answers

10. About the Exam

[Introduction](#)

[Structure](#)

[Objectives](#)

[The Kubernetes and Cloud Native Associate exam](#)

Quick facts about the exam

[Booking the exam](#)

Purchasing the exam

Scheduling the exam

[Preparing for the exam](#)

[Before the exam](#)

Candidate handbook

System requirements

Identity requirements

Environment

[Exam day](#)

Pre-launch checks

Launching the exam

During the exam

[Results](#)

[Links](#)

[Conclusion](#)

[Index](#)

CHAPTER 1

Stepping up to Kubernetes and Cloud-Native

Introduction

This introductory chapter will take you through a warmup session, introducing and defining concepts that are important to map the world of Kubernetes and cloud-native as well as for the KCNA exam. It covers important topics such as monolithic and microservices architecture, virtualization, cloud computing, and containers. The chapter also briefly introduces Kubernetes, its origins, and cloud-native.

The chapter also includes information about the **Cloud Native Computing Foundation (CNCF)** and its role in promoting cloud-native technologies. You will also get a statistical view of the adoption of Kubernetes and cloud-native technologies and their importance as a skill for career growth. The chapter will also provide an overview of all the cloud-native certifications available from the foundation.

This chapter is for people who are either new or just getting started with technologies like cloud, containers, and microservices. In the subsequent chapters, we will dive straight into the KCNA domains.

Structure

This chapter covers the following topics along with quizzes:

- Application architectures
- Computing fundamentals
- Kubernetes and cloud-native
- CNCF certifications

Application architectures

Application architecture is a blueprint of design principles, patterns, and techniques that act as a guide when building and running applications.

Changes in business and consumer needs and digitization have shaped application design patterns promoting architectures that are more agile, resilient, fast and improve overall user experience. Hence, application architectures have evolved over the years from tightly coupled designs such as monolithic to loosely coupled designs such as microservices.

This section will introduce these architectures and also provide a difference between them as they are necessary to understand the importance of cloud-native designs discussed later in the book.

Monolithic

A monolithic application is usually characterized by a single code base that combines all functionalities of an application. The user interface, business logic, and data access logic are all tightly coupled as a single unit.

As an application grows and matures, so does the code base, and hence, it becomes complex to build, test, and manage over time. A small change to any of the functionalities results in the rebuilding of the entire application, and an issue with the change, no matter how small, carries the risk of a complete application down and the business impact thereof.

The following illustration shows a typical monolithic application architecture where all components are tightly coupled into a single code base:

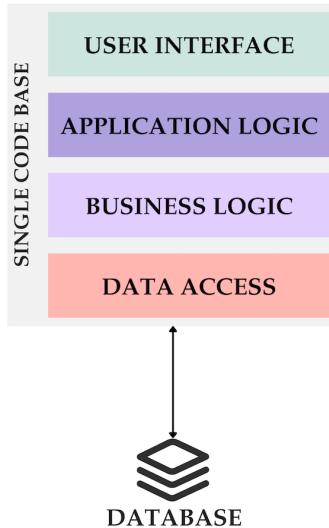


Figure 1.1: Monolithic application architecture

Microservices

Microservices architecture is an application development approach that breaks down a monolithic application into smaller, more manageable components, which enables various benefits such as agility, faster development, accelerated time-to-market, and more. Each component runs as a separate service, and they communicate over APIs. In a microservices architecture, each service performs a specific function, and scalability is limited to the demand of that function.

Each microservice is packaged independently along with its binaries, libraries, and other dependencies, mostly using containerization (covered later). This allows running multiple microservices together in a single VM or host, even though they may be running different programming languages and frameworks.

Microservices are developed around business needs, allowing the best technology selection for a use case. They mostly perform a single function and are designed for failure.

Running microservices in containers (managed by container orchestrators) is common these days.

The following illustration shows a microservices architecture where the application's components are broken down:

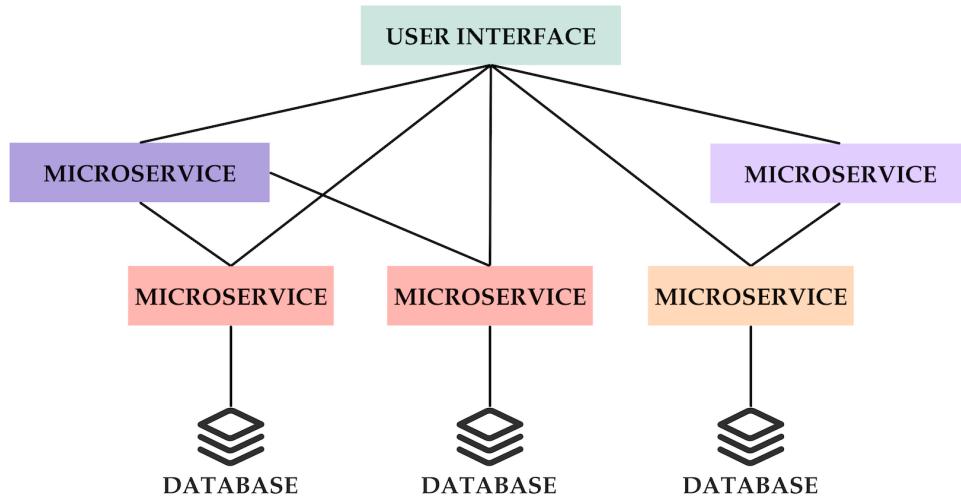


Figure 12: Microservices application architecture

Examples of applications that have migrated (refactored) from a monolithic architecture to microservices include GitHub, Uber, and Docker.

Monoliths or microservices

Let us try to map the differences between a monolithic and a microservices-based architecture through a use case.

The example application is a fintech application that allows a user to make payments in merchant stores by scanning a QR code. It has the following functions:

- User interface or front-end
- Authentication
- QR code validator
- RTP generator

- Checkout
- Payment processor
- Orders

The team is looking to implement a change, which is to add a new payment method to the checkout page. The following table lists the differences when implementing this change in a monolithic architecture vs. a microservices architecture:

	Monolithic	Microservices
Develop	Depending on the modularity of the code base, this could be complex	It is simpler to develop as changes are on a smaller code base
Build	Build entire code base (slower)	Build checkout service only (faster)
Test	Run all test cases for the entire application (slower)	Run test cases for checkout service only (faster)
Deploy	Redeploy the entire application, complete downtime	Redeploy checkout service only, with minimal to no downtime
Troubleshoot	It is difficult to identify the breaking change, especially if there were multiple code changes	It is easy to identify the breaking change and rollback

Table 1.1: Comparison between monolithic and microservices

A graphical illustration of the architectural difference is shown in *Figure 1.3:*

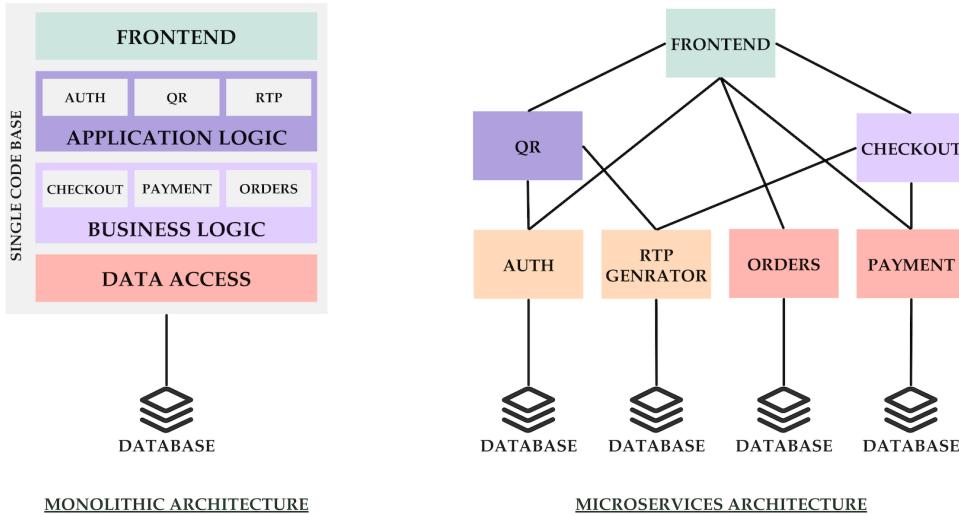


Figure 13: Microservices application architecture

A commonly asked question is which architecture is better, or sometimes the assumption is that microservices are always better than monolithic architecture. This is not always true. The choice of an application architecture depends on many parameters, such as the complexity of business logic, stage of development, complexity of the application, etc.

If an application's business logic is simple, that is, there are few functions, then a microservices architecture could be overkill. Similarly, it is seen that early state development of applications is always monolithic. Sometimes, it is better to keep it simple and monolithic.

Both architectures have advantages and disadvantages; hence, choosing the right architecture based on the needs is important. Modern applications are generally complex from the beginning and have a complex business logic; hence, a microservices architecture is the best fit.

For the KCNA exam, the focus will be on microservices.

Computing fundamentals

Computing has also evolved, keeping pace with the evolution of application architectures. In this section, we will look at the two pivotal

points in computing history that revolutionized how we run applications more efficiently.

Virtualization

Before virtualization, most applications ran directly on servers (read physical servers, bare metal hardware) that ran an operating system of choice. As the use of technology exploded, especially the internet and web-based applications, there was an increase in the number of applications being developed and deployed, hence emphasizing the need to optimize resource utilization and provide computing in a flexible and scalable manner. Enter virtualization. It enables the logical division of compute resources, thus running multiple operating systems (and workloads) on a single physical server. Virtualization was a pivotal point in the evolution of computing. It was driven by innovations such as **software defined infrastructure**, which eventually expanded to other areas such as networking, storage, etc. A software-defined approach to provisioning resources played a crucial role in the evolution of computing from hereon.

Virtualization decouples hardware from the operating system and provides a virtualized or emulated set of resources such as CPU, RAM, disk, and network by using software called a **hypervisor**, thus allowing multiple operating systems to run on the same hardware. The operating systems running on virtualized hardware are called guest VMs, and to the average OS user, it would not appear very different than bare metal hosting. Being able to run multiple operating systems on a single physical server meant we could run multiple applications on a single physical server. This provided increased resource utilization and efficiency, along with many other benefits.

The benefits of virtualization include:

- Reduced infrastructure management
- Reduced support personnel (due to consolidation)

- Faster provisioning than traditional hardware
- Faster provisioning meant reduced wait time to launch applications
- Lower cooling and power requirements
- Lower TCO and better ROI
- Hardware independence and portability between the same hypervisors

Some popular virtualization technologies include XEN, KVM, and the more famous VMware (ESXi).

The following figure illustrates the difference between traditional computing and virtualization:

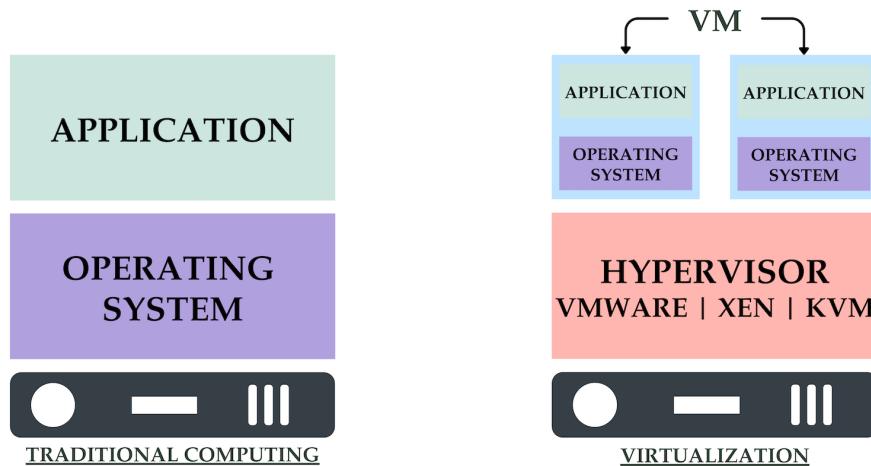


Figure 14: Microservices application architecture

Virtualization frees up unused resources (read servers) that could be repurposed for labs, development, etc.

Cloud computing

One of the important challenges with virtualization was the lack of self-service capability and on-demand provisioning. We still needed technical staff to provision the VMs, which were usually on request. Cloud

computing solved these challenges and many others by providing the capability to self-provision VMs and many other services (over time) through the use of APIs over a network.

Cloud computing is a game changer that brought in a paradigm shift in computing, especially in the way it is consumed. With the growth of e-commerce, social media, and wearable devices, computing power has been needed like never before. Cloud computing enables utility-based computing that is on-demand and pay-as-you-go. It provides a level of elasticity and scalability, which virtualization cannot. All this without having to invest upfront on hardware, especially helping small businesses and startups (and today's enterprises, too) adopt digital transformation. Software-defined computing methods such as virtualization and models such as Web 2.0 technologies have delivered this utility-based computing that we know today. This was a big boon, especially for developers. In fact, at Amazon, this was the driver for creating the AWS service.

Virtualization forms the basis of cloud computing. Hence, it continues to be used in the creation of cloud platforms, whether public or private. So, cloud computing was not necessarily an evolution in computing itself but was a pivotal change in how we consumed computing services.

While virtualization requires that you have an administrator who accesses the hypervisor admin portal or uses a CLI to create the resources for you, cloud computing is all about communication through APIs. The service, VM, for example, is exposed through an API and is accessible directly or through web-based interfaces/mobile devices, and anyone with a valid account and subscription can submit a request for resources on-demand. The API then talks to the hypervisor to create them. Public cloud platforms like Microsoft Azure, AWS, Google Cloud Platform are examples of such platforms.

Another key characteristic of cloud computing is multi-tenancy.

Cloud computing can be provided through one of the following deployment models:

- **Public cloud:** A public cloud is a deployment model accessible over the internet and is available to anyone. The users can be organizations, small businesses, startups, or individuals, and it is a true example of a multi-tenant service at scale. The services provided by public cloud vendors are used by hundreds of thousands of customers across the world, and some of the services are accessed millions of times per second. Popular examples of public cloud vendors are Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, and Oracle Cloud Infrastructure. Users pay as per use, otherwise called pay-as-you-go, which is one of the important selling points of public cloud. Most providers also offer some resources /services for free.
- **Private cloud:** This deployment model is usually restricted within an organization and accessed over a private network. The organization deploying or commissioning it maintains complete control over it. A cloud operating system is required to build private clouds, some examples of which are OpenStack, CloudStack, Azure Stack, and Rackspace. A private cloud allows organizations to configure it as per business needs security and allows for the highest levels of control. At the same time, setting up a private cloud involves upfront capital expenditure.
- **Hybrid cloud:** A hybrid cloud deployment is when an organization uses public and private cloud setups for their applications and workloads. An example could be where the database is hosted on a private cloud for security, compliance, and data sovereignty reasons, but the web application could be running in multiple public cloud locations. Hybrid clouds are also ideal for meeting short-term demands of resources.
- **Community cloud:** While not so popular and rarely spoken about, it is a model where multiple organizations, mostly with shared interests, share a cloud setup. The cloud setup may be internally managed or outsourced. Examples of this are government organizations sharing a single cloud, medical

organizations sharing a cloud platform for the development of a vaccine, etc.

Cloud computing defines three primary classifications of the service model. A service model defines the management responsibility of resources and, hence, the amount of control over them between the provider and the customer. These service models are defined as follows:

- **Infrastructure-as-a-Service (IaaS):** This delivery model closely resembles a VM and has components that make up a fully functional VM, such as CPU, memory, disk, and network, along with a pre-installed operating system. Popular examples include AWS EC2 instances, Azure Virtual Machines and Google Compute Engine instances. IaaS provides the greatest degree of control among all the cloud models but also has the greatest management responsibility. The customer manages OS (pre-installed by vendor), middleware, runtime, application, and data.
- **Platform-as-a-Service (PaaS):** A PaaS model includes middleware, runtime, and development tools apart from the infrastructure and OS. This makes it ideal for organizations focusing on application development and deployment only. PaaS platforms also do away with software license management. The cloud provider manages everything except the application and data. Azure Web Apps, AWS Beanstalk, Google App Engine are popular examples.
- **Software-as-a-Service (SaaS):** This service model involves the vendor managing everything, including developing the application. The customer consumes the application directly, usually through a subscription. This is ideal for organizations for whom IT is not the core business, such as a furniture manufacturing company that simply needs productivity apps for business operations. Microsoft 365 and Google Workspace are examples of SaaS.

Figure 1.5 shows a quick difference between these three models and the on-premises model:

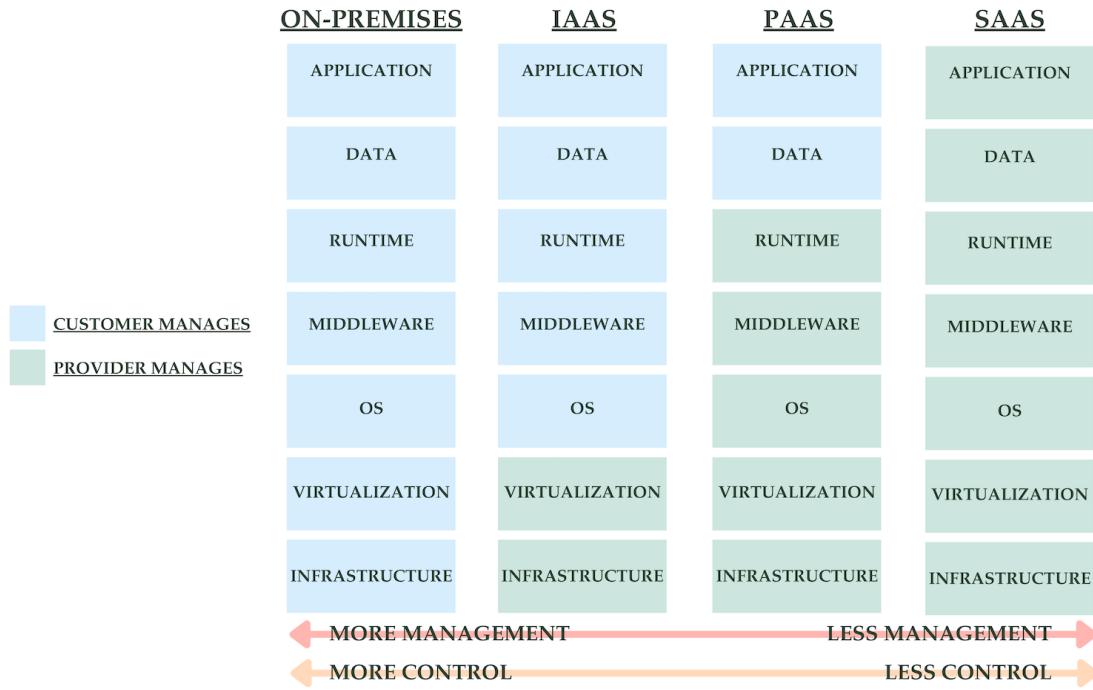


Figure 15: Resource management responsibilities

Note: The preceding image illustrates only the comparison of resource management responsibility between deployment models. This does not show the shared responsibility model, which will be covered later in the book.

Cloud computing as we know it today has evolved since the early 2000s from just VMs on-demand to hundreds of other services, such as object storage, firewalls, load balancers, managed databases, migration services, data-as-a-service applications, and many more. One of the emerging delivery models is the concept of everything-as-a-service, for example, Container-as-a-Service, Functions-as-a-Service, Kubernetes-as-a-Service, etc.

Note: This book covers virtualization and cloud computing topics as a warmup to containers, Kubernetes, and cloud-native. If you are new to these concepts and would like a more detailed explanation of them, BPB has some great books that cover this in more depth. Here are two of them:

- Cloud computing (<https://in.bpbonline.com/products/master-cloud-computing-concepts-cloud-computing-books>).

- Cloud computing simplified (<https://in.bpbonline.com/products/cloud-computing-simplified>)

Containers

Containers are the next evolution in the way we develop and run applications. Let us understand containers with an example.

Say our application is based on a microservices architecture with seven components, as mentioned in the example application architecture earlier. Let us look at some scenarios to understand the challenges we may face when running microservices in a traditional or virtualized environment and how containers can help:

- **Scenario 1:** There are two microservices using the same dependency, and one of them needs a dependency upgrade due to a feature change. However, this may impact the other microservices. If all services are running on a single server/VM, this can cause a breaking change.
- **Scenario 2:** One microservice has a bug that results in memory hogging. Overutilization of resources by one can result in resource shortage for others, causing a crash.
- **Scenario 3:** One of the microservices may need an older version of the runtime, while others may use a newer version. Running multiple versions of runtime on a single system can always cause management, security, and other issues.
- **Scenario 4:** Let us say we decide to run each microservice in its own VM, this would be a waste of resources as microservices may not use all allocated capacity.
- **Scenario 5:** The code may work on the developer's machine but not the production machine. This is called incompatibility between environments.

Containers solve these problems and many more. A container is a package containing the code and everything that is necessary to run the code, such as the runtime, libraries, other dependencies, configuration, etc. This packaging abstracts the underlying environment, making it portable to run anywhere. Containers leverage operating system virtualization and share the kernel, each within their isolation. An immediate benefit of this is lower resource utilization, such as memory and CPU.

A perfect analogy to containers is a shipping container, where you load goods (ideally similar goods or goods that belong as a unit), and they can be moved by any means of transport that can handle containers. A practical example of this is the movement of household stuff through a packer and mover due to relocation. The packer and mover loads all your household items into a single container (a unit) marked for you, and the container can be shipped from point A to point B without the need to unpack it in transit. The same container can be loaded onto a ship, then unloaded at the port and loaded onto a truck before it reaches your new destination.

The concept of containerization or isolation is not new and goes back many decades to 1979 when chroot system call was introduced in Unix V7; between this and the 2000s, UNIX had mechanisms termed as jails to isolate a runtime environment from other system processes and protected resources. From the 2000s, there have been many notable advancements in this area, such as Solaris Containers in 2004, Process Containers by Google in 2006, LXC or *Linux Containers* in 2008, Warden by CloudFoundry in 2011, and let me contain that for you (LMCTFY) by Google in 2013. Docker (launched in 2013) contributed to the democratization of containers as we know it today.

The benefits of containers over virtual machines include:

- **Open source:** Containers are based on open-source technology; hence, new features are available faster, and security issues are addressed faster.

- **Portability:** Move between multiple operating systems, hardware, and cloud platforms easily.
- **Consistency:** If a container runs on the developer's machine, it will also run on production.
- **Faster start time:** VMs have an OS; hence, there is a boot time that adds to the application ready time, whereas containers start almost instantly.
- **Lightweight:** While applications can also be packaged into VMs similar to containers, VM images are GBs in size as they contain the entire operating system, whereas container images are much smaller. This also means less overhead on resource requirements.
- **Microservices friendly:** Containers complement the microservices architecture in the best possible way.
- **No vendor lock-in:** The platform's open-source nature ensures that we can move the containers without vendor lock-in.
- **Faster development:** Combined with principles such as agile and DevOps, SDLC processes such as development, testing and deployment can be faster.

Business benefits include:

- **Faster time to market:** Containers (along with a microservices architecture) speed up development, testing, and deployment, allowing businesses to release apps and features faster and gain a competitive edge.
- **Efficiency and optimization:** Containers help achieve better optimization of resources and operational efficiency, especially when used through the DevOps model.
- **Better TCO and ROI than VMs:** Optimization of resources and efficiency in operations means better TCO and ROI than VMs.

While not very efficient, monolithic applications can also be containerized with minor modifications (sometimes no modifications).

The container world has many more concepts. A quick introduction to some of them are as follows. Some of these, such as *container orchestration*, will be covered in depth in the later chapters.

- **Container image:** Developers combine the code, runtime, and other dependencies into a packaged unit called a container image. When these image(s) are run, they are called containers. Container images comprise layers that are presented as a single virtual filesystem to the container during runtime. They are read-only and immutable.
- **Container registry:** Often, these images create a need to be shared with other developers or sometimes be available to everyone. A container registry helps you do that: push (store) and pull (access) the images. They provide secure and authorized ways to do so.
- **Container runtime:** The software that can run the container images created earlier on a host is called a container runtime (also called container engine). On a high level, container runtimes download the images from the container registry and run them for us with the provided configuration. They also help us manage the lifecycle stages of a container, such as creation, deletion, starting, stopping, etc. Some popular container runtimes are:
 - Docker
 - Podman
 - CRI-O
 - Containerd
 - Mirantis Container Runtime
- **Container orchestration:** As container adoption grew and microservices became the norm in newer application architectures, a challenge that was evident was managing them on a large scale. Assuming each microservice was running in a container and an application could have tens or hundreds of microservices while also scaling simultaneously, there are quickly a lot of containers to

manage. To add more complexity, some containers could function as a unit, and some may need high availability. To address these needs and more, tools and software were needed.

Enter container orchestration. A container orchestration software deploys, manages, and automates the lifecycle of containers (running microservices) at scale. They provide capabilities such as intelligent scheduling and provisioning of containers, resource management, self-healing, autoscaling, load balancing, routing, monitoring, and more. They add another layer of abstraction to containers.

Kubernetes is the most popular container orchestration software in the market. Other examples include Red Hat OpenShift, Docker Swarm, Apache Mesos, Rancher Labs, etc.

Installing, configuring, monitoring, and managing a highly available, scalable, and secure container orchestration platform in itself involves a lot of work and requires expertise. So, cloud vendors have been providing platforms such as Kubernetes as a managed service. Examples include Azure's AKS, AWS EKS, and GCP's GKE.

Container orchestration will be discussed in more detail in [*Chapter 2, Understanding Containers and the Need for Container Orchestration*](#).

Virtualization versus containers

Let us summarize the difference between virtualization and containers with an illustration:

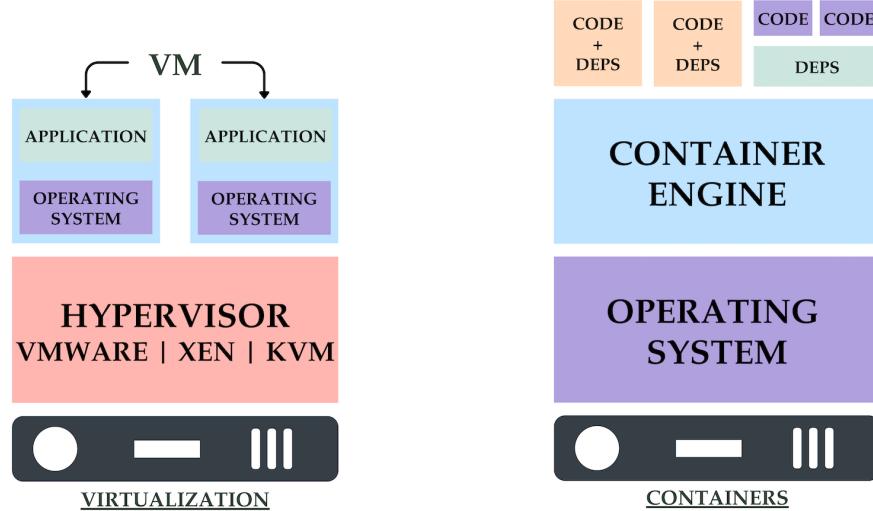


Figure 16: Virtualization versus containers

Future of applications and computing

The evolution can be looked at from two different approaches. One being how we consume them (delivery model) and the other being where we consume them. The computing evolution continues with technologies such as serverless, and we are no longer just consuming them remotely from a data center. The following are some of the advances in computing in the form of a quick introduction:

- **Serverless:** It is the next step in building and running applications, especially targeted at developers. One of the key benefits of a serverless architecture is zero infrastructure management. One does not even have to create container images. Just write code and let a (cloud) vendor manage all aspects of infrastructure, such as provisioning, scaling, storage, database, etc. The execution of the code is trigger-based; that is, an action or an event triggers the function to run. The action can be a click, an API call, or an event, such as a file upload.

The customer pays for computing only when the code runs, unlike VMs or container platforms, where you are charged for the running infrastructure. There are other costs associated, though,

like storage, API calls, databases, etc. This approach to computing is also called **Functions-as-a-Service (FaaS)**.

Benefits of serverless include lower computing costs (subjective), inbuilt scalability, ready-to-use backend-as-a-service, and faster development.

Examples of serverless solutions are AWS Lambda, Amazon DynamoDB, Azure Functions, Azure CosmosDB, Google Cloud Functions, Cloudflare Workers, etc.

Serverless application architecture and computing use cases include:

- Trigger or event-based invocations, such as a welcome email on the first sign-up
- Asynchronous processing tasks such as resizing an image after upload
- Running REST APIs
- **Edge computing:** This is already a big thing in computing today. It is a form of distributed computing where applications are deployed closer to data sources, that is, devices generating data. This is to facilitate faster processing of data for action, insights, and more.

Data, touted as the new oil, is being heavily used in businesses today for decision-making, especially in real-time. This has fueled the adoption of edge computing, among other benefits, such as lower latency and better bandwidth management. Edge computing involves moving infrastructure running time-sensitive applications to remote locations where data is generated.

Some use cases for edge computing include:

- Autonomous vehicles
- Wearable device
- Remote monitoring in industrial establishments like oil

- Content delivery (caching)
 - Telecommunications
 - Predictive maintenance
 - Agriculture
- **Fog computing:** At times, cloud computing may be too far away for processing data, edge computing may be too limited in resources to process the same data, or edge devices may be distributed across a very large area, making it difficult to implement edge computing. This is where fog computing can help. Fog computing infrastructure can have a larger capacity than edge computing and can be much closer to the data being produced.
- Smart cities are a good use case for such computing environments. Another example can be traffic video surveillance.

Kubernetes and cloud-native

Cloud-native architectures define most modern applications today and Kubernetes is at the forefront of it. Together, they revolutionize application development, deployment, and management.

In this section, we will look into the origins of Kubernetes and cloud-native and learn about the organization that is at the helm of advocating them.

A brief history of Kubernetes

Kubernetes is an open-source platform for managing containerized workloads at scale and is today more than just a container orchestrator. Its framework can help run scalable, resilient, and automated distributed systems at a scale. Kubernetes's open-source and pluggable nature makes it flexible, allowing users to bring in their tools. It is an ideal developer platform.

The Kubernetes project was created by a bunch of Google engineers and was based on Google's Borg cluster manager software. It was released to the world in 2014 as an open-source, community-driven project and later became the first project of the CNCF. Kubernetes is written in the Go programming language.

If you want to learn more about the origins and the Borg project, read the blog at: <https://cloud.google.com/blog/products/containers-kubernetes/from-google-to-the-world-the-kubernetes-origin-story>

The book will cover the technicalities of Kubernetes in detail in the later chapters, as it is the most important exam domain.

Cloud-native basics

It is obvious now that one must try to fit cloud-native into this evolution.

Cloud-native is a modern approach to building and deploying applications. It is a set of design guidelines, principles, and practices that help organizations build, deploy, and run applications in an agile, scalable, and resilient manner. They also emphasize automation and observability. Cloud-native tools and technologies augment and implement these principles and practices.

Applications are loosely coupled and designed for modern computing environments such as cloud, containers, and serverless from the ground up. Cloud-native applications are highly observable and use a greater degree of automation. Developers can experiment and make changes frequently.

A typical cloud-native architecture usually includes applications developed with a microservices architecture, packaged into containers, running on an immutable infrastructure with the microservices communicating with each other using API running on a service mesh. The entire lifecycle from development to delivery is continuously managed through a DevOps or DevSecOps pipeline involving continuous integration (CI) and

continuous deployment (CD) at the least. A true DevOps/DevSecOps pipeline includes more than just CI/CD.

Popular cloud-native stacks usually include the following components:

- Microservices
- Containers
- Container orchestrators (Kubernetes)
- Service mesh
- Observability
- DevOps and CI/CD

In a true cloud-native model, you can work with each preceding component without disrupting the entire system.

Businesses are changing, evolving, and are more product-aligned today. They needed products and features delivered yesterday. The need for speed and agility will drive businesses to digital transformation, and a cloud-native approach can help them achieve this faster.

This is just an introduction to the concept of cloud-native, and it is obvious to have questions. Cloud-native principles, their advantages and disadvantages are covered in [*Chapter 7, Cloud-Native Architecture*](#).

Cloud Native Computing Foundation

The CNCF is a community of organizations, developers, and end users who create and contribute to cloud-native technologies that make it possible to build and run modern applications in a cloud-native approach. Their mission is to make cloud-native ubiquitous.

CNCF was founded in 2015 by the Linux Foundation and a number of technology companies (22 of them), with Kubernetes as the seed project donated by Google. Since then, CNCF has seen rapid growth in the number of cloud-native projects, members, and contributors. At the time of writing this book, according to the stats at the CNCF website

(<https://www.cncf.io/>) there are 173 projects hosted by CNCF, 218K contributors and 190 countries involved.

CNCF hosts supports, directs, and oversees open-source vendor-neutral cloud-native projects like Kubernetes, Prometheus, Argo, Helm, and Istio. The projects are categorized into:

- **Sandbox:** The entry point for early-stage projects; these are experimental in nature and have not yet been used in production.
- **Incubating:** These are projects that are running in production in small numbers. They are generally stable.
- **Graduated:** Graduated projects are widely implemented in large environments, have a lot of contributors, and are more mature. The best example is Kubernetes.

Note: Visit <https://www.cncf.io/projects/> for a complete list of projects.

CNCF is a part of the larger Linux Foundation, a non-profit that acts as a hub for the open-source ecosystem across different technology sectors. As of this writing, it hosts 850 projects, and CNCF is just one of them. The Linux Foundation provides a learning and certification platform, provides training, does research, and many more activities, all governed by a leadership and advisory board. Learn more about all this and more at:

<https://www.linuxfoundation.org/>

The CNCF, along with the Linux Foundation, also hosts several events across the globe for the cloud-native community, members, and organizations like the popular *KubeCon + CloudNativeCon* and *Open Source Summit*. These events are a great place to learn about the latest and greatest developments in cloud-native technologies, their adoption across industries and use cases, and network with the community. Checkout the following page for upcoming events at:

<https://events.linuxfoundation.org/>

Cloud-native landscape

The cloud-native landscape is a directory of all cloud-native projects. Think of it as a map that can help you navigate through the overwhelming world of cloud-native. It contains both open-sourced and closed-sourced projects from across the world. The directory can be filtered based on the license type, organization, industry, and more.

The visual landscape can be accessed at:

<https://l.cncf.io>

The weblink also has a serverless landscape for technologies related to serverless architecture. There are PNG and PDF versions of the landscapes available as well. One can also download a CSV of all the data.

Apart from the landscape, the CNCF portal has a host of other informative resources, some of which are as follows:

- **Trail map:** If you are getting started and need a recommendation on building a cloud-native operation, this is for you.
- **DevStats:** An activity analysis on GitHub, such as contributions to Kubernetes and other CNCF projects.
- **Technology radar:** A must-read to keep yourself up-to-date about emerging technologies.

State of Kubernetes and cloud-native

The current state of cloud-native is confirmed to have crossed the chasm, meaning a larger market segment is now adopting it and has become mainstream. Similarly, technologies like Kubernetes have also crossed the chasm.

Various reports from leading organizations that provide tools and software for cloud-native workloads reveal that organizations are running more and more workloads on containers and Kubernetes, including production and mission-critical ones. This shows increasing confidence in

the cloud-native ecosystem. Let us look at some statistics that validate this.

Gartner, the US-based technological research and consulting firm, predicts that over 95% of new digital workloads will be on cloud-native platforms by 2025. That is not very far away:

<https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences>

The CNCF annual survey of 2022 reveals that containers are the new normal, and Kubernetes' use in production continues to increase. 64% of CNCF end users, these are companies that use Kubernetes internally and do not offer it as a service, use Kubernetes in production. Similarly, 49% of non-end user companies are using Kubernetes in production. Read more of the report at:

<https://www.cncf.io/reports/cncf-annual-survey-2022/>

The Kubernetes in the Wild Report 2023 by Dynatrace, an infrastructure observability platform provider powered by AI and automation, states that Kubernetes is emerging as the operating system of the cloud. The report finds that most Kubernetes clusters in the cloud are managed, such as AKS from Azure, EKS from AWS, and GKE from Google Cloud. It also reveals an interesting trend where Kubernetes clusters are increasingly running more auxiliary services such as service meshes, observability tools, security controls, etc., which means that cloud-native technologies' adoption is also on the rise. The report is available at:

<https://www.dynatrace.com/news/blog/kubernetes-in-the-wild-2023/>

Similarly, the 2022 Container Report by Datadog, which provides monitoring and security solutions, reveals that nearly half of organizations use Kubernetes to deploy and manage their containers. Managed Kubernetes services are very popular. The report is available at:

<https://www.datadoghq.com/container-report/>

Another report worth mentioning is The State of Kubernetes 2023 by VMware Tanzu which also finds that the benefits of Kubernetes go beyond just IT. The following is an excerpt from the report's summary:

Our key finding for 2023 is that Kubernetes is delivering measurable business benefits that extend beyond the boundaries of IT. Nine out of 10 (90%) respondents agree that cloud-native technology, including Kubernetes, is transforming the way their business operates. Almost two-thirds of stakeholders' report seeing indirect business benefits, such as developers are more productive (60%) and IT operators are more efficient (64%). A significant number recognize direct benefits—ones that directly affect the bottom line including the business is seeing growth in market share (25%), new revenue-driving customer experiences have been created (21%), and profit margins are increasing (20%).

The detailed report can be downloaded from the following link:

<https://tanzu.vmware.com/content/ebooks/stateofkubernetes-2023>

So, it is clear that Kubernetes and cloud-native adoption is on the rise, and there will be a demand for certified skilled professionals more than ever. Reports have also shown that an increase in the adoption of open-source technologies has revealed a shortage of skills.

CNCF certifications

Now that we know there is a skill shortage in this domain let us look at how you can acquire and validate these skills.

The CNCF, in collaboration with the Linux Foundation, provides courses and exams leading to certifications around Kubernetes and cloud-native through the Linux Foundation Education. The Linux Foundation also has courses and certifications for domains outside of Kubernetes, such as system administration (on Linux), web, and applications for Node.js. It has skill certifications across the IT career roadmap covering systems and infrastructure, network and security, and software development.

The following are some certifications from the *Linux Foundation Education* stable that cover the cloud, containers, and cloud-native domains. There are many more certifications than what is in the following figure. For a complete list visit:

<https://training.linuxfoundation.org/certification-catalog/>

Refer to the following figure for a high-level overview of currently available certifications across different domains:

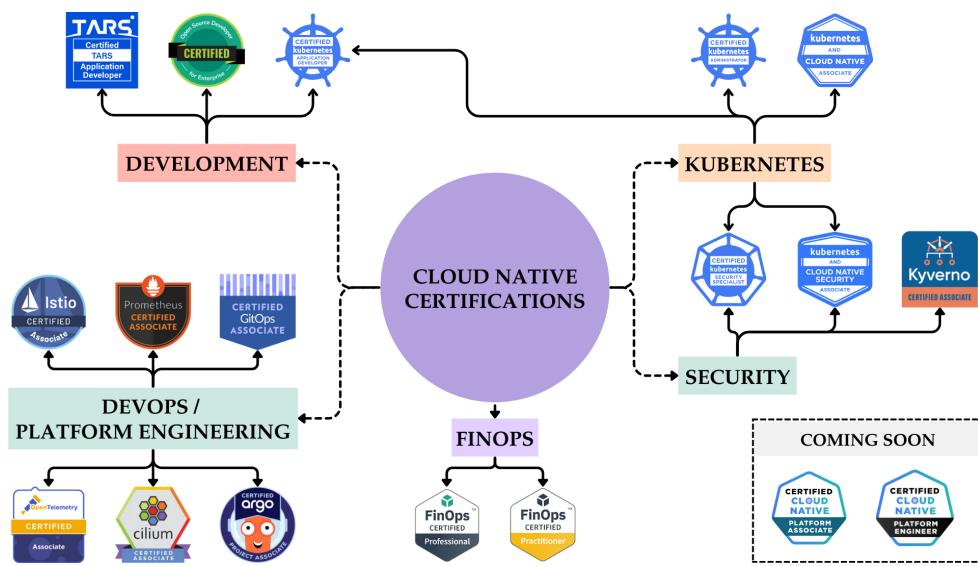


Figure 17: Cloud Native Certifications from the Linux Foundation

The official training and certification career roadmap graphics can be viewed at

<https://training.linuxfoundation.org/it-career-roadmap/>

Courses and certifications can be purchased from the Linux Foundation training website at

<https://training.linuxfoundation.org/full-catalog/>.

Most of the exams are performance-based and, hence, carry a lot of credibility in the industry. A certification based on one of these exams demonstrates a candidate's confidence and, more importantly, practical knowledge in the respective domain. Kubernetes-related certifications, especially the Certified Kubernetes Administrator (CKA), are among

the highly desired certifications for DevOps professionals. KCNA is an important steppingstone to acquire the next level of Kubernetes and cloud-native certifications.

The Linux Foundation Education website also provides a few courses for free to help candidates get started in the world of Kubernetes, Linux, and more. They can be viewed at <https://training.linuxfoundation.org/resources/>. A small list of these courses and their links are also included at the end of this book.

Career opportunities

Acquiring skills and certifications in technologies such as containers, Kubernetes, and other cloud-native projects like Prometheus can open up many career opportunities for an individual. Certifications such as the KCNA demonstrate an individual's theoretical knowledge of Kubernetes and cloud-native technologies, helping them stand out. A subsequent certification, such as a CKA, further demonstrates a strong hands-on command over Kubernetes.

The annual 2023 State of Tech Talent Report by Linux Foundation Research, which looked at the hiring managers' perspective, emphasizes the importance of upskilling and states that certifications are more important than university education and are necessary to verify skills. The report also reveals that a number of organizations are looking at hiring more staff in the cloud/container technology areas. Kubernetes, CI/CD, and DevOps are some of the skills organizations are prioritizing hiring on.

The report is available to read at:

<https://www.linuxfoundation.org/research/open-source-jobs-report-2023>

Popular job profiles that look for skills and certifications on Kubernetes and cloud-native technologies such as Helm, Istio, and Prometheus are:

- DevOps engineer

- Site reliability engineer
- DevOps architect
- Platform engineer

Even traditional infrastructure engineer, systems engineer, and systems administrator jobs prefer candidates with container and Kubernetes skillsets as the organizations hiring are either exploring or expecting the transition to these technologies sometime in the future.

Recently, there have also been job profiles that are more Kubernetes-focused with job openings for designations such as Kubernetes engineer, Kubernetes administrator, Kubernetes developer, and Kubernetes specialist.

Today, even data, AI, and ML technologies run on cloud-native platforms. Hence, acquiring these skill sets will cover you in those domains, too.

So, whether you are a fresher or an experienced professional looking to career transition to cloud-native, there is always a role for you. Experienced professionals can leverage their existing knowledge in scenarios such as migration and modernization as they would know both worlds.

Conclusion

Microservices architecture enables the development of complex applications by breaking them down into smaller, independently deployable services. This approach enhances scalability, improves fault isolation, facilitates faster development cycles, and allows for greater flexibility in technology choices. Containers are a preferred choice to deploy microservices, preferably in a cloud/cloud-native based environment.

A cloud-native architecture is a modern approach to building and deploying applications specifically on dynamic cloud environments. It encompasses a set of principles and technologies, including microservices,

containers, container orchestration, and DevOps practices, that enable organizations to leverage the full potential of cloud computing.

Kubernetes is one of the many projects at **Cloud Native Computing Foundation (CNCF)** and is one the important projects driving cloud-native adoption. The CNCF landscape is a great place to view and understand the scale of cloud-native projects. With containers being the new normal, Kubernetes is the preferred (open-source) platform for developing, deploying, and managing cloud-native architectures. Hence there is a huge demand for Kubernetes and cloud-native skilled professionals. The *Linux Foundation Education* has many courses and certifications to empower individuals with these skills.

In the next chapter, we will learn about containers and go over an introduction to Docker, image registry, container images, runtime etc. and understand the container creation workflow. We will then understand why we need a container orchestrator and its advantages. We will also cover an introduction to open standards such as CRI, CNI, CSI and SMI along with a quick introduction to Kubernetes.

Multiple choice questions

1. In a microservices architecture, you can mix and match programming languages and frameworks.
 - a. False
 - b. True
2. Which of the following computing models is an ideal fit for microservices?
 - a. Mainframes
 - b. Virtual machines
 - c. Containers
 - d. Kubernetes
3. What kind of virtualization are containers?

- a. Hardware virtualization
 - b. Operating system virtualization
 - c. Application virtualization
 - d. Data virtualization
4. Which of the following challenges of virtualization were solved by cloud computing?
- a. Availability of self-service capability
 - b. Upfront CapEx
 - c. Lack of elasticity
 - d. Ease of accessibility over a network
5. Which of the following features of containers results in less overhead on resource requirements?
- a. Portability
 - b. Smaller footprint
 - c. Faster startup
 - d. Open source
6. In which year was CNCF founded?
- a. 2012
 - b. 2014
 - c. 2013
 - d. 2015
7. What was the seed project of CNCF?
- a. Prometheus
 - b. Docker
 - c. Borg
 - d. Kubernetes

8. Given the following scenario, which cloud deployment model would you choose?

A boutique family-run business grows and sells Christmas trees. They want to jump on the e-commerce bandwagon, but given the nature of business, which is seasonal, they do not want to invest in IT infrastructure. They would like to be able to scale up during peak seasons of the year and scale down to minimal resources for the rest of the year.

- a. Private cloud
- b. Hybrid cloud
- c. Public cloud
- d. Community cloud

9. Which cloud service model provides the least management responsibilities?

- a. SaaS
- b. PaaS
- c. IaaS
- d. FaaS

10. Which of the following are traits of microservices over a monolithic architecture? (Choose all that apply)

- a. Independent scalability
- b. Higher resiliency
- c. Complex infrastructure
- d. Complex development processes

Answers

1.	b.
2.	c.
3.	b.

4.	b. c.
5.	b.
6.	d.
7.	d.
8.	c.
9.	a.
10.	a. b. c. d.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Understanding Containers and the Need for Container Orchestration

Introduction

The second chapter of this book focuses on containers and container orchestration. While the official exam blueprint mentions Kubernetes first in the list, understanding how containers work, the need for container orchestration, and other container infrastructure components, such as networking, storage, and security, is important before learning about Kubernetes as a container orchestrator.

While the main objective of this chapter is containers and container orchestration, this chapter also covers the following domains and objectives of the exam blue print:

- **Container orchestration:** Container orchestration fundamentals
- **Container orchestration:** Runtime
- **Cloud-native architecture:** Open standards

Structure

This chapter cover will cover the following topics:

- Containers
- Container orchestration
- Open standards
- Introduction to Kubernetes

Objectives

This chapter aims to provide an introduction to various container concepts, such as images, registry, the container runtime, storage, networking, and security in containers using Docker, the most popular container platform, as a base while highlighting the challenges of managing standalone containers leading up to the need for container orchestrators. The chapter covers container orchestration fundamentals, such as what a container orchestrator should do, a typical workflow, its benefits, and more. The chapter also provides an introduction to common open standards that are used in the container and cloud-native space, which are also part of the exam objectives. The final objective of this chapter is to briefly introduce Kubernetes and cover a few fundamentals around it. In the end, readers will be familiar with the general workings of a container and the need for container orchestration, along with an introduction to Kubernetes, the container orchestrator of focus for this book and the exam.

Note: This chapter will use Docker to explain various container concepts due to its popularity and simplicity. Occasionally, there will be references to Kubernetes and how certain concepts relate to it, but we will not deep dive into them in this chapter.

Containers

Containers are immutable, repeatable, and a standardized approach to running applications consistently across different environments. They decouple applications from the underlying infrastructure. They are great

for microservices-based application architectures, and their implementation can be seen in most modern applications.

In the first section of this chapter, we will see how containers work.

Occasionally, this chapter will refer to the example microservices pattern mentioned in [Chapter 1, Stepping up to Kubernetes and Cloud-Native](#) to explain some concepts. Here is the microservices architecture again for reference:

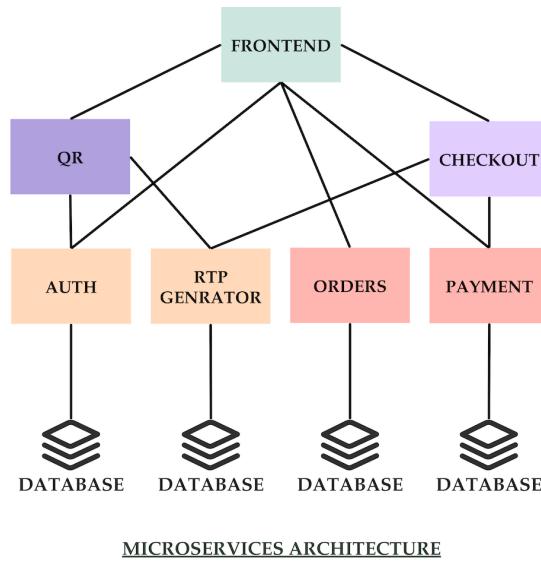


Figure 2.1: Microservices application architecture

Docker

Before we begin to understand the workflow of creating and running a container, let us quickly introduce Docker as it will be referred to often. As a product, Docker is a platform for developing, shipping, and running containers. It provides a suite of tools covering various aspects of running containers. The most popular of its tools is the Docker Engine.

Docker Engine is based on a client-server architecture. *Docker* (the command) is the client tool or CLI used for interacting with the software, and *dockerd* (Docker daemon) is the server daemon or process that provides the API, among other capabilities. Another important tool is the

Registry. Together, these three are the important components of the Docker Engine.

The following figure shows a typical Docker architecture and the workflow:

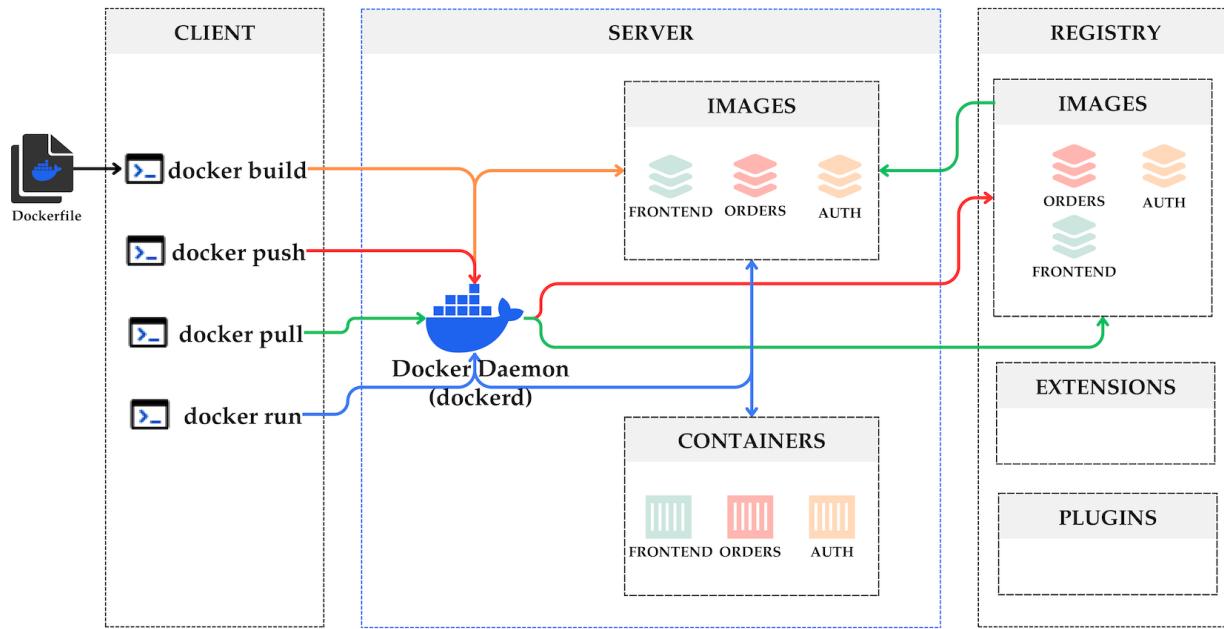


Figure 2.2: Docker architecture

We will describe some of them further as we progress in the chapter.

Container images

Container images are the building blocks of containers and containerized applications. It is the first step to running a container. A container image is a package that contains the application code, runtime, dependencies, configuration and anything else required for the application to run. You create a container from a container image. Think of it as a template.

Docker is popularly used to create container images. To create the image, a Dockerfile is written, specifying a base image and, subsequently, various commands (in different stages) to build the final image. At the very basic, these commands install necessary software, copy application code, configure required settings and start the application.

Consider the following example of a Dockerfile:

```
# Pull official base image
FROM node:16-alpine

# Set working directory
WORKDIR /app

# Add app
COPY . ./

# Install
RUN npm install

# Start app
CMD ["npm", "start"]
```

To build a container image using Docker, the **docker build** command is used. The command includes a tag and the location of the Dockerfile. An example command is given as follows:

```
docker build -t app-v1 .
```

Container images are based on a layered filesystem. Each step executed in the Dockerfile results in a filesystem that is a delta of the previous. This creates layers of filesystems on top of the base image. The technology used behind the scenes is the **copy-on-write** (CoW) strategy. Docker uses **unionfs** (union filesystem) to create a virtual filesystem from these layers. Storage drivers are used depending on choice, but they all support this layered filesystem approach. Some popular drivers are **aufs**, **overlay**, **overlay2**, **ZFS**, etc. Docker recommends the **OverlayFS** storage driver, even Kubernetes uses it. The layered filesystem is great in saving space and reducing the size of images, especially when multiple container

images share layers, such as base image and other layers with common software.

Figure 2.3 shows the layered approach of Docker images:

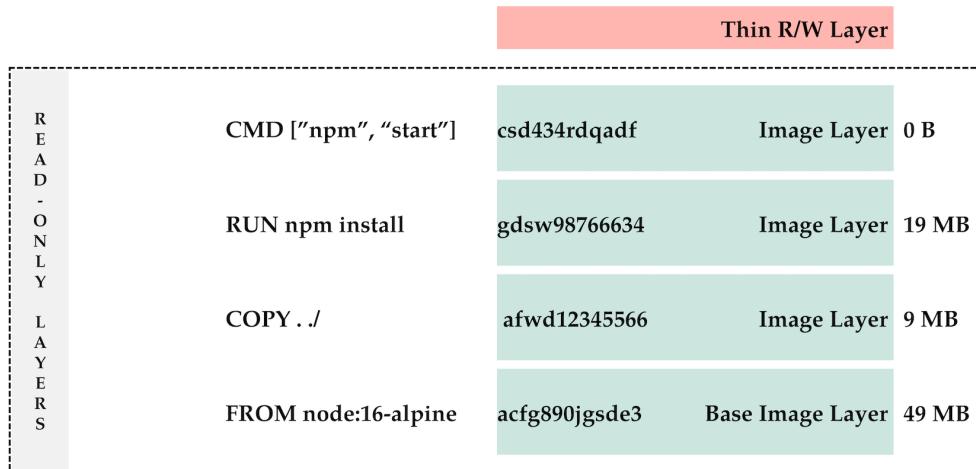


Figure 2.3: Container image layers

The final image is a lightweight, standalone unit ready to be deployed as a container. These images are **immutable**, can be versioned and deployed to environments such as dev, staging, production, etc. Any changes to the application in a container image are incorporated by building a new image and recreating the container.

Note: Container images can also be created from existing containers.

The final thin read-write layer shown in the preceding figure is where any modifications (writes) are stored. This layer is created when a container is spawned and deleted when the container is deleted. When a file needs to be modified, it is copied into this layer first. Hence called copy-on-write.

Container images are not only efficient in terms of resource utilization but also enable consistent and reliable application deployment, making them a fundamental component of modern container orchestration and cloud-native computing.

Container registry

Container images created in the previous step often need to be shared with other developers or sometimes made available to everyone. Hence, a way to store, distribute, and access them is needed. A container registry helps you do just that, acting as a storage repository for the images. Often centralized (and sometimes local, too), this is a storage location where developers can push images to and pull images from.

Popular container registries support versioning and tagging for history and to track changes. Modern container registries also address security by providing access control mechanisms, container image vulnerability scanning, and more. Container registries can be public or private.

A container repository contains all the versions of a particular container image, whereas a container registry contains multiple such images. The following figure shows a visual representation of this:

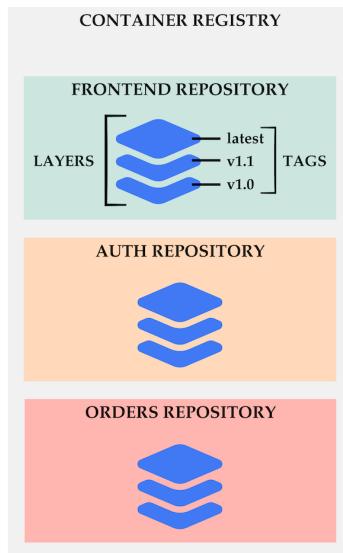


Figure 2.4: Container registry vs. container repository

A local *registry* is an example of a private registry and *Docker Hub Registry* (<https://hub.docker.com/>) is an example of a public registry.

Note: Docker's Registry was donated to the Cloud Native Computing Foundation (CNCF) and is now called Distribution, of the Open Container Initiative (OCI) distribution specification.

All the specifications such as storing and distributing are part of it.

Now that container images are created and stored in a container registry, we are ready to create and run containers. In the next section, we will see how to do that and learn how containers work.

Container runtime

At a very fundamental level, a container runtime creates and runs containers from the container images. A Linux based container runtime uses native features such as **cgroups** and **namespaces** to do this. Let us define them both:

- **cgroups** or control groups allow limiting, controlling, and monitoring resources such as CPU, memory, and disk I/O for a group of processes. *cgroups* also provide features such as prioritization. It was developed at Google and later merged into the Linux Kernel. The current version of *cgroups* is v2.
- **Namespaces** allow partitioning the kernel for isolation of processes from one another. The following is a list of namespaces used with containers:
 - **Process ID namespaces:** A process in a namespace can have the same Process ID (PID) as another process in another namespace.
 - **User namespaces:** A process may run as root inside the namespace but is less privileged outside.
 - **Network namespaces:** These are used to provide networking to containers by virtualizing the networking stack. Each network namespace is independent. It also supports physical interfaces.
 - **Inter-process communication namespaces:** Provide isolation by providing processes their own **inter-process communication (IPC)** resources, such as semaphores, shared memory, and message queues.

- **UNIX Time-sharing (UTS) namespaces:** Each process can have different hostnames and domain names.

Following are the different types of container runtimes that are used.

- **Low-level container runtimes:** A low-level container runtime performs tasks, such as setting up namespace, cgroups to create and run isolated processes called **containers**. It is uncommon and impractical to interact and use a low-level runtime directly.

Some popular examples of low-level runtimes are:

- **runc:** Written in go language, it is currently the industry standard low-level runtime for many high-level runtimes and engines such as Docker, Containerd.
- **crun:** This low-level runtime is written in C programming language, is fast and has a low-memory footprint.
- **runhcs:** A fork of *runc* written for Windows.

All of them implement the **Open Container Initiative (OCI)** specifications. We will cover the OCI standard later in this chapter.

- **High-level container runtimes:** Low-level container runtimes, such as *runC* can create a container in its simplest, most primitive form. However, in order to execute and manage the lifecycle of a container and provide higher capabilities, such as image management, storage mounting, networking we use high-level container runtimes such as **containerd**. High-level container runtimes have capabilities that primarily help with complete container lifecycle management. They provide capabilities, such as API, formats, packing, unpacking, image sharing etc. They are also called **container managers** or **container engines**.

Some popular examples of high-level runtimes or container engines are:

- Docker engine (dockerd)

- containerd
- Podman
- CRI-O
- Mirantis Container Runtime

To show the relationship between low-level and high-level container runtimes, consider this statement: **containerd** executes the containers, but **runc** creates them.

In most container architectures, a shim is used between the low-level runtime and the high-level runtime. For example, in containerd, **containerd-shim** is used. The shim enables daemonless containers and provides functionalities, such as allowing the runtime (*runc*) to exit after the container starts, keeping the STDIO open for containers in situations where the containerd or docker daemons die. Without this, the containers would exit.

Exam tip: Both low-level and high-level container runtimes create namespaced containers, i.e., they (all containers running in the same host) share the same kernel. Hence if a container gets exploited, it may result in a security issue at the host level as well, thus exposing all running services on the host and leading to privilege escalations.

- **Sandboxed container runtimes** use a proxy layer between the container and the OS kernel to provide additional isolation. This proxy monitors the instructions sent by the container to the kernel. **gVisor** container is an example of this.
 - **gVisor:** It is a Linux compatible open-source container runtime from Google and focuses on security at its core.
- **Virtualized container runtimes:** These runtimes use a small footprint virtual machine to deploy the container image. However, this makes their performance slow. Linux containers running on a

Windows host is a good example of this. Kata containers, an OpenStack project, fall under this category.

Exam Tip: Virtualized containers are the most secure container implementations followed by sandboxed containers and namespaced containers. Namespaced containers share the host kernel and, hence, are more prone to security exploits.

Container runtimes are installed on a supported operating system and the underlying infrastructure can be anything, such as bare metal, virtual machine, cloud instances, etc. A question often asked is whether containers and virtual machines are mutually exclusive. They are not. Containers can run inside virtual machines, and they complement each other.

It is emphasized again that containers are immutable, i.e., the contents of a container cannot be changed. Any changes made to a running container are lost when the container exits. To persist changes, we can either add the changes as another layer, create a new image, and re-create the container, which is impractical for many workloads, such as databases, or add persistent storage to it. Storage for containers is covered later in the chapter.

The most common (high-level) container runtime is *containerd*, which is also the default container runtime in Kubernetes and used by Docker as well. Container orchestrators such as OpenShift use *CRI-O*.

Container creation workflow

While at the container topic, let us quickly look into the steps involved in creating a container. We will use Docker as an example. A typical workflow when running a container using Docker involves the following steps:

1. Execute a container run command such as **docker run**. This step is possible due to the client tools such as Docker or Docker

Compose CLI that communicates with the Docker API. This invokes the Docker API.

2. At this point, the payload is passed on to `containerd` for further execution.
3. Containerd pulls the container image from the container registry (it first checks if the image is available locally).
4. Containerd creates a bundle based on the image; image layers are extracted to a COW filesystem and overlayed to create a merged filesystem.
5. Additionally, parameters and metadata are set, such as overriding `CMD`, `ENTRYPOINT`, SECCOMP rules, etc.
6. At this stage, the bundle is handed over to the low-level container runtime, which is runc via shim.

Note: Shim here enables daemonless containers and helps avoid a long running runc process among other things such as keeping the STDIO and FDs open, report status etc.

7. runc works with the kernel to create isolation for various internals, such as for process, network, filesystem, inter-process communication and UNIX time-sharing using namespaces.
8. runc also works with the kernel to create control groups (cgroups) to limit resources like CPU or memory limits to this container.
9. Finally, the container is started by runc and on a successful start. It hands over the control of the container to the shim and exits.

Note: When a container is removed, the namespace, allocated resources and the read-write layer are all deleted.

The following figure illustrates the container creation workflow:

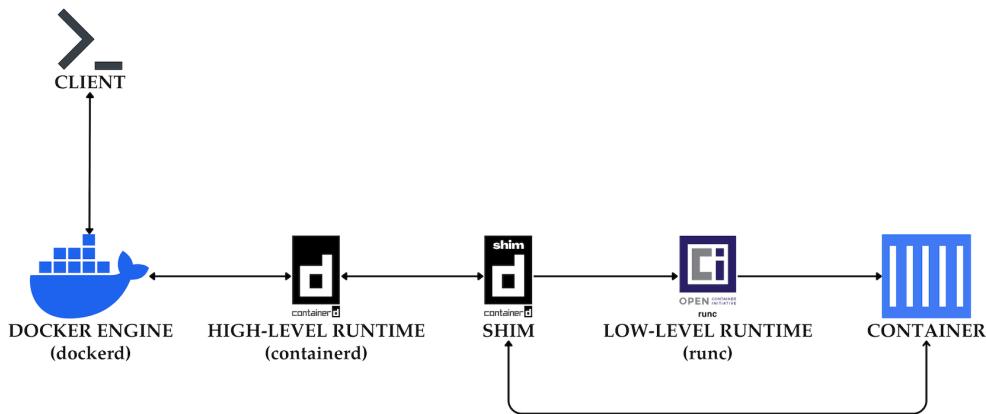


Figure 2.5: Container creation workflow

Note: The following explanations related to container storage, networking, and security are from the context of running stand-alone containers.

Explanations around container, networking, and security from the context of container orchestration are given later in the chapter.

Container storage

Containers are stateless by design. There is a thin read/write layer on a running container where changes are stored, but the changes are lost if the container fails or exits. In order to persist the data on the read/write layer, a new image must be created by including this layer. However, this solution is not practical for stateful workloads, such as databases or dynamic applications.

Hence, containers require persistent storage for stateful workloads. Container runtimes support storing data outside of the container lifecycle using various methods such as path mounts, data volumes, etc. This decoupling also enables bringing in a storage backend of choice, which is implemented using storage drivers. Now, if the container fails or exits, we

can create a new container and use the same volume or path to access the data.

Docker provides two mechanisms to persist data beyond the lifecycle of a container. They are **bind mounts** and **volumes**. The difference between the two is shown in the following table:

Volumes (Preferred option)	Bind mounts
Created and managed by Docker	Created and managed by user
Use Docker CLI to create volumes	Cannot use Docker CLI
Volumes are stored in a specific directory in the host filesystem managed by Docker. By default, it is /var/lib/docker/volumes	Bind mount paths can be anywhere on the host system
Cannot be modified by non-docker processes	Can be modified by non-docker processes
Secure than bind mounts	Security concerns due to the ability to change host filesystems by an infected container

Table 11: Difference between volumes and bind mounts

Container networking

Complex microservices (running in containers) often have to communicate with each other, with the host, with another host, and sometimes with the outside world. Container networking allows this communication to happen. At the very basic, containers require IP addresses, Domain Name System (DNS), hostnames and nameservers.

Container networking is implemented using the *network namespace*. Ports used by containers are unique within the namespace only and are mapped to a host port for access.

Traffic between containers is referred to as **east-west** traffic and the traffic between the microservices and external endpoints is called **north-south**.

The following are popular container networking types:

- **Bridged:** This is the default networking in most popular runtimes like Docker and Podman. It enables communication between containers on the same host.
- **Host:** This networking type shares the container host's network stack directly.
- **Overlay:** It is used for communication between containers across container hosts.
- **Macvlan:** This networking type assigns a MAC address to each container and connects it directly to the physical network.
- **IPvlan:** Containers using this network type can share the same physical network interface but have distinct IP addresses.
- **None:** Container is not connected to any other container or the host. It sets a loopback device only.

For example, Docker provides various networking capabilities, such as IP addressing, DNS, hostnames along with support for nameservers and IPv6 addressing. It also provides a pluggable networking model using drivers which allows users to bring in a networking stack of choice. It supports the preceding networking types and *bridged* is the default type used.

Container security

While containers use isolation as a fundamental property, it is not enough. An out of the box simple implementation of containers has a

high risk of being exploited.

Let us take one such scenario to understand this a little more. Containers on the same host share the kernel, which becomes an attack surface. A compromised container can be used as an attack vector to gain access to the kernel and hence to other containers as well. Using untrusted container images or running untrusted code inside a container can lead to a compromised container. A threat actor may use these compromised containers to gain access to the host and execute their malicious plans, such as creating more containers, using your resources, accessingin, and modify sensitive data etc.

Containers are the most targeted platforms today and security should not be an afterthought.

When running stand-alone containers, you must follow certain best practices to secure the containers. Additionally, use security features provided by container runtimes engines and if necessary, use third-party security tools based on the level of security required. The following is a list (not a complete list) of some of the common best practices:

- Run only signed images by leveraging *Docker Content Trust* signature verification feature. In simple terms, you can pull images to run from a container repository that is signed with a specified key.
- By default, containers in Docker (and many other container runtimes) run in root mode. Prefer running them in rootless mode. Use root mode for a container only if really necessary and with security controls in place.
- Use capabilities in containers where applicable. This helps in providing a specific capability without providing root access.
- By default, containers connected to the same container network (for example *bridged*) can talk to each other. Similarly, exposing a container outside of the host makes a potential risk too, avoid exposing all containers and use a reverse proxy. The onus is on

the developer/administrator to configure networking in a secure manner.

- Allow trusted users to manage and control the docker daemon.
- Use Linux security features such as SELinux, seccomp or other external tools.

Note: The REST API endpoint uses a UNIX socket instead of a TCP socket and this provides additional security.

Docker (as a product) provides many security features such as roles, access management, single sign-on (SSO), multi-factor authentication (MFA), and more. However, the scope of this section is to cover security within Docker Engine only.

Implementing security for stand-alone containers and doing this for every container for multiple container hosts can be a daunting task with the possibility of leaving loopholes. We will see later in the book how implementing them through container orchestration can be relatively easier.

Container orchestration

Now that we understand how a container works, let us look at the next problem statement. We saw that a lot goes on behind the scenes to execute, create and manage the lifecycle of a single container. The effort is amplified when there are multiple containers, persistent storage, networking and a need to securely run these containers. So, while containers make microservices-based application development, publishing, and deployment easy and more efficient, they become harder to manage and operate as their numbers grow.

Additionally, there are other real-world challenges that appear when executing, running, and managing containers at scale. Some of them are highlighted as follows:

- **High availability:** A container host failure will result in complete downtime since the containers will also go down with the host.

Containers can be created on a new host, but this becomes a manual task, including provisioning a new container host.

- **Scalability:** Stand-alone containers cannot auto-scale. One can always create additional containers manually to scale, but managing the storage, networking, load-balancing, and the time required is not practical in a real-world scenario.
- **Operational inefficiency:** When you have many containers and container hosts, scheduling which container is deployed on which host becomes a manual task. This would also require deep observability, which container runtimes cannot provide. Without visibility, containers hosts may either be overutilized or underutilized. Add additional manual tasks like storage management, networking configuration, and applying security policies, the time and effort spent in managing stand-alone containers is simply inefficient.
- **Agility:** A microservices architecture brings agility to application development. However, if we have to manually upgrade a container every time there is an application update and have to do this for containers at scale, the application is no longer agile. Again, the time and effort spent in one such operational task is huge in cloud-native terms.
- **Cost:** Inefficiency in resource utilization, operations, and the lack of agility, scalability, and high availability all add to increased costs in managing containerized applications.

To summarize, managing containers at scale becomes an inefficient, costly, and repetitive task.

Container orchestration solves these challenges and provides many cloud-native capabilities. In this section, we will look into the fundamentals of container orchestration and learn about popular container orchestrators.

Container orchestration fundamentals

Container orchestration is fundamentally about automation. A container orchestrator software (today) automates the packaging, execution, deployment, scaling, management, and monitoring of containers at scale. It can also automate the lifecycle of a container's associated infrastructure, such as storage, networking, and security. You are no longer required to manage each container host and the stand-alone containers running inside it individually.

Simply put, you can run multi-container applications on multiple machines.

Container orchestration is made possible by using a declarative approach to defining the behavior of an application. You define the composition of the application, how it should be deployed, scaled, configured for access, networked and secured, and let the container orchestrator manage the provisioning and lifecycle of infrastructure components needed for running the application. This is a level of abstraction above containers and the reason the declarative configuration can work anywhere, i.e., on-premises, on cloud, on edge, or a mix of them.

Container orchestration helps you shift the focus to running the application instead of running the containers.

Many of these started as simple tools for container orchestration but are full-fledged platforms today. Container orchestrators such as Kubernetes are game changers. Some call them cloud-native operating systems or container operating systems. Some other popular container orchestrators apart from Kubernetes are OpenShift and Docker Swarm. Most of these are also available as open source; one can self-host them and customize them according to requirements.

Common container orchestrators

Here is a quick introduction to the popular container orchestrators:

- **Kubernetes:** Also called K8s, its origin goes back to Google's Borg container management system. It is the most widely used

container orchestration platform.

- **OpenShift:** This comes from the house of Red Hat and is the preferred platform in enterprises. The enterprise product is called **OpenShift Container Platform (OCP)** and the upstream community-driven open-source variant is called *OKD*. It uses the **CRI-O** runtime.
- **Docker Swarm:** Coming from Docker, this product line provides advanced feature sets for managing a cluster of Docker hosts, including scaling, networking, and security. It is technically called **Docker Swarm mode** and is part of the docker engine.
- **Marathon:** While not many may know this product, it is a container orchestration platform for *Apache Mesos* and *DC/OS*, which you may have heard.

Container orchestrator architecture

Before we can declare how multiple containers should be orchestrated to run our application, we must create a cluster with all the hosts where containers will run (managed hosts) and some additional managing hosts.

A typical container orchestrator architecture consists of two types of nodes (hosts), which are explained as follows:

- **Control nodes:** They are also called **control plane nodes**, **master nodes**, or **manager nodes**. These nodes run various components, such as APIs, data stores, etc., that manage the worker nodes, schedule containers, and more.
- **Compute nodes:** They are also simply called **nodes** or **worker nodes**. These are the nodes where containers run.

Once a cluster is ready with a container orchestration tool like Docker Swarm or Kubernetes, we provide the declarative configurations file to it. The orchestration tool uses the information in it to schedule deployment of containers into the cluster by choosing the best host for them. The orchestration tool also manages the lifecycle of the container post-

container deployment based on predetermined specifications. Container orchestration tools work in any environment that runs containers.

We will revisit these concepts again in [Chapter 4, Kubernetes Basics](#).

Container orchestration workflow

The following figure is an illustration of a typical container orchestration workflow:

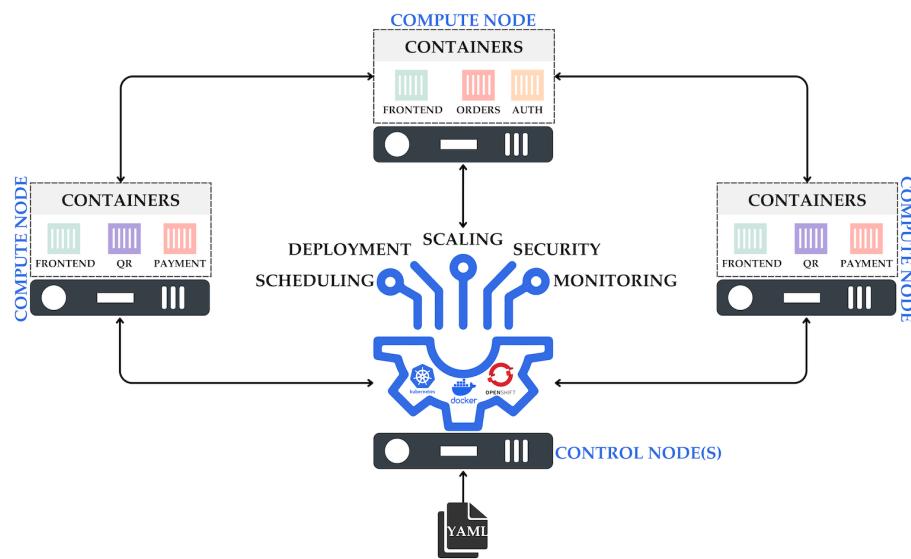


Figure 2.6: Container orchestration workflow

Functions of a container orchestrator

While different container orchestrators may bring in additional functions or features based on various business requirements, following are some of the core functions every container orchestrator performs:

- **Scheduling:** One of the primary functions of an orchestration platform is to identify a host based on (container) resource requirements and other parameters, such as constraints, and coordinate the deployment of containers.
- **Scaling:** Add or remove containers based on demand and limits.
- **Desired state:** Ensure that containers are running as declared in the configuration file.

- **Self-healing and monitoring:** Recreate failed containers, maintain a minimum number of containers, monitor container resource consumption, etc.
- **Networking:** This includes capabilities such as load balancing, networking between containers, service discovery, etc.

Benefits of container orchestration

Apart from the obvious benefits described earlier, the following are some additional benefits to container orchestration:

- **Cost savings:** Better resource utilization and dynamic scaling result in cost savings. Additionally, container orchestration requires fewer man-hours in management, which means additional cost savings.
- **Observability:** Container orchestrators come with some degree of monitoring and observability and additionally support many open-source tools. These provide deep visibility into the entire containerized infrastructure and applications, which bring in other benefits, such as cost optimization, better governance and security.
- **Consistency:** A declarative approach to application deployment results in a consistent deployment anywhere; this also avoids configuration drift.
- **Flexible:** Container orchestration provides greater flexibility when running microservices. You can mix and match tools and technologies for an optimal stack that works for your application. Decoupled storage and networking provide additional flexibility.
- **Faster time to market:** Container orchestration brings in cloud-native features and agility, thus enabling businesses to bring new application features, capabilities and updates to customers faster.
- **Distributed:** Container orchestration allows you to build a platform that is distributed across geographies, clouds, datacenters, providing the possibility of an always online application. Application is decoupled from the infrastructure.

- Enhanced reliability: Achieve enhanced reliability through observability and monitoring, implement self-healing of failed containers and improved fault tolerance.

Open standards

Before we introduce Kubernetes, the container orchestration platform, let us be aware of the various open standards that provide specifications that Kubernetes uses.

There is more to consider when using a container orchestration platform. We have seen how complex and diverse a container orchestration platform can be. Additionally, with the popularity of containers and Kubernetes, numerous runtimes, storage, networking and security tools are now available. This multitude of options can become overwhelming and introduce insecurities around interoperability, compatibility, and vendor lock-in that can impact the adoption of cloud-native architectures for applications. Businesses may hesitate to invest huge amounts only to find that a tool or platform has become old/decommissioned or is no longer compatible with a platform.

Standards are required to ensure interoperability, avoid vendor lock-in, etc. These standards provide specifications for interfaces (think API) that allow customers to bring in a runtime, storage, and network of their choice. These specifications work together to govern the interaction of container engines, hosts, orchestrators, and the supporting networking and storage.

When you think of a standard, the best analogy is the SATA interface for hard drives or the USB-C interface for mobile chargers. A common standardized interface means you can swap charging cables.

Open Container Initiative

The Open Container Initiative (OCI) is a Linux Foundation project that focuses on creating open standards for container formats and

runtimes. Quoting from their official website at <https://opencontainers.org/>:

Established in June 2015 by Docker and other leaders in the container industry, the OCI currently contains three specifications: the Runtime Specification (`runtime-spec`), the Image Specification (`image-spec`) and the Distribution Specification (`distribution-spec`). The Runtime Specification outlines how to run a filesystem bundle that is unpacked on disk. At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime.

OCI standardizes low-level runtime and provides specifications to do so. These specifications are as follows:

- **image-spec**: The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. The spec consists of an image manifest, image index, multiple filesystem layers and configuration.
- **distribution-spec**: This specification defines an API protocol to facilitate and standardize the distribution of content.
- **runtime-spec**: The runtime specification aims to specify the configuration, execution environment, and lifecycle of a container. The configuration is specified in a **config.json** file.

The OCI standard follows best practices and, most importantly, avoids vendor lock-in. A high-level summary is that containers should not be bound to clients or orchestration stacks, must not be tightly associated with a commercial vendor or project, and must be portable across infrastructures.

In simple words, a container image created using OCI specifications can run on any container runtime that implements the OCI specs.

Container Runtime Interface

The **Container Runtime Interface (CRI)** is more specific to Kubernetes. It is an API that allows Kubernetes to use any container runtime that is CRI compliant. *containerd* and *CRI-O* are examples of runtimes that conform with the CRI. In simple words, you can swap the container engine.

Not all container runtimes are the same, especially the higher-level container runtimes. Each container runtime may have features different from others. As the number of container runtimes increased and Kubernetes grew popular, there was a need to be able to support all of these. In the past, Kubernetes integrated directly with container runtimes to support them. Hence, a pluggable interface (think API) called the **Container Runtime Interface** was developed and introduced in 2016 that enabled interoperability, allowing users to bring in a runtime of their choice.

The CRI defines the protocol for communication between *kubelet* (this runs on every Kubernetes worker node) and the container runtime. This protocol is based on gRPC and contains two services, the *ImageService* and the *RuntimeService*, which are explained as follows:

- The *ImageService* is used for pulling images from the repository, inspecting the images, and removing the images.
- The *RuntimeService* is used for managing the lifecycle of pods and containers including interacting with them.

Container Storage Interface

The **Container Storage Interface (CSI)** is also an API specification that enables a pluggable storage architecture and allows Kubernetes (or any CSI compliant container orchestrator) to use any storage provider that provides a CSI compatible plugin.

The need for the CSI standard was similar to the CRI. Prior to CSI, the code for the volume plugin that supported various backends was part of the core Kubernetes binaries. With time and the adoption of Kubernetes,

it became difficult to add new volume support faster, among other security and development challenges. CSI enables interoperability and avoids vendor lock-in when it comes to storage.

When implementing a storage backend, the storage vendor provides a CSI driver that can be deployed on Kubernetes, enabling containers to use them. The CSI standards are very minimal and provide great flexibility in developing the CSI drivers. Both block and file systems are supported.

The core services of the CSI protocol include the *IdentityServices*, *ControllerServices*, and *NodeServices*.

Container Network Interface

Similar to CSI, Container Network Interface (CNI) enables a pluggable network architecture in container orchestrators like Kubernetes, enabling one to bring in a networking solution (through the use of a plugin) of choice.

CNI provides specifications and libraries for writing plugins, which can be used by the container runtime to configure network interfaces. At the time of this writing, according to the CNI page (<https://www.cni.dev/>), both Linux and Windows containers can use CNI.

CNI is also a CNCF project, and many container runtimes such as *containerd*, *CRI-O*, *rkt*, etc., and container orchestrators such as *Kubernetes*, *OpenShift*, *Apache Mesos*, and *Amazon ECS* use this specification. Some popular networking plugins that implement CNI are:

- Calico
- Flannel
- Weave

Service Mesh Interface

A service mesh is an infrastructure layer that primarily provides communication capabilities for the services running in a container

orchestration platform. These days, it is common to use a service mesh for complex microservices-based architectures where an increasing number of network endpoints surface. A service mesh helps in managing, monitoring, and securing these network endpoints. It provides capabilities, such as traffic policy, telemetry, and management. While most CNI plugins work on Layer 3 (some support Layer 2 as well), a service mesh works on Layer 7 of the networking model.

Service Mesh Interface (SMI) is also a Cloud Native Computing Foundation (CNCF) project and was created to standardize and provide open specifications covering common service mesh capabilities. SMI is API based and more specific to Kubernetes. The end goal is a portable, interoperable, and agnostic service mesh implementation.

The most common service mesh technologies one may have heard of are *Istio*, *Linkerd*, *HashiCorp Consul*.

Introduction to Kubernetes

We have already referenced Kubernetes multiple times in this chapter and are aware that it is a container orchestration platform. It is a CNCF *graduated* project that builds upon over a decade of Google's experience running containers at scale and is a highly extensible and versatile container management platform. Kubernetes has been widely adopted and implemented in production across various industries including large enterprises and is no longer only used for non-production workloads or by startups and SMEs. Kubernetes has crossed the chasm, i.e., it has entered mainstream.

It is declarative, which means you write your desired configuration in a file (typically YAML/YML) and provide it to Kubernetes. It stores this configuration in a data store, repeatedly compares the desired state with the current state, and always ensures the desired state is maintained. To compare this to an analogy, think of a thermostat used to maintain the desired temperature of a room, where it automatically turns on and off to do so.

Note: Kubernetes does support imperative commands/API.

Kubernetes provides many enterprise level and production ready features such as:

- Horizontal scaling
- Self-healing
- Automated rollouts and rollbacks
- Service discovery, load balancing, and networking
- Extensibility
- Automatic bin packing
- Secret and configuration management
- Network policies
- IPv4/Ipv6 dual-stack, and more

This chapter briefly introduces Kubernetes and addresses that it can solve stand-alone container management challenges around runtime, storage, networking, security etc. We will explore its architecture and other technical details in [Chapter 4, Kubernetes Basics](#).

Runtime

Kubernetes needs a container runtime to spin containers, and it should be installed on every node where containers will run. The default runtime is *containerd*, but Kubernetes implements the CRI, which means the container runtime layer is pluggable and can be replaced with any runtime that is compliant. Containerd is responsible for downloading and running the container image.

Exam tip: Kubernetes needs a container runtime that is CRI compliant such as containerd, CRI-O etc. CRI-O is also a popular runtime used with Kubernetes.

This discussion is incomplete without a quick history of Docker and Kubernetes. Prior to the CRI standard, Docker was the default and the only container runtime supported in Kubernetes for some time (K8s introduced support *rkt container engine* too). Docker engine does not implement CRI. After it was introduced, Kubernetes community created dockershim so Kubernetes could use Docker as if it was a CRI compatible runtime. However, dockershim was getting difficult to maintain due to the additional effort and increasing many new features were incompatible. It was officially removed from Kubernetes from release v1.24. However, *containerd* is also from the Docker stable, is an independent CNCF project, and implements the CRI standard.

Containerd runtime

Before we move ahead, let us give a quick introduction to *Containerd*, which is the default container runtime in Kubernetes.

According to the official definition from <https://containerd.io/>:

Containerd is an industry-standard container runtime with an emphasis on simplicity, robustness and portability

Containerd was born out of Docker and later donated to exist independent of it. It is a CNCF graduated project. It implements the CRI, which is required for it to work with Kubernetes. Containerd allows you to bring in any low-level runtime of choice such as runc, kata, gVisor, etc.

The following feature highlights of containerd make it the most used (high-level) container runtime:

- It manages the complete lifecycle of a container from images to execution including storage and networking.
- Its modular and extensible design provides users with better control over container execution and supervision.
- It is known to provide better container performance.

- It uses robust communication protocols such as *gRPC* and *protobuf* for fast and reliable container operations.
- It implements *seccomp*, *AppArmor* and provides certain default profiles that provide better security and vulnerability management.

Note: Containerd is also the runtime in Docker.

CRI-O runtime

This is another container runtime designed specifically for Kubernetes. Hence, it implements the CRI and conforms to OCI specs. It is also a CNCF graduated project.

CRI-O was created as a joint effort by Intel, IBM, Red Hat, SUSE, and Hyper. It was created when Docker was still used in Kubernetes, and the idea was to have a runtime that can be used directly by Kubernetes. CRI-O supports runc, kata, or any OCI runtime.

Container orchestrators such as *OpenShift*, *Oracle Linux Cloud-native Environment*, and *openSUSE Kubic* use CRI-O as the default runtime.

rkt container runtime

The *rkt* container runtime was developed at CoreOS and later acquired by Red Hat. It implemented a separate container format called the *App Container spec* or *appc*, which is different than those created using Docker but was able to use them.

rkt used a pod-based approach similar to Kubernetes, was pluggable, supported CNI, implemented security features like SELinux, and integrated with Kubernetes.

It was accepted by CNCF and incubated the project. However, it is no longer an active project.

Storage

Kubernetes supports a pluggable storage architecture based on the CSI specifications, which means you can bring in a storage backend of your choice. Kubernetes publishes a list of supported drivers, and due to its open-source nature, it supports the most popular storage platforms, such as NetApp, Amazon EBS, Azure Disk, and GCE Disks. It provides complete storage lifecycle management capabilities.

Networking

Kubernetes uses a networking model that tries to mimic a traditional virtual machine or physical host networking. This is achieved by software-defined networking, abstraction, and a plugin-based networking implementation. Container runtimes implement the network model using plugins that use the CNI specifications. This makes the networking model pluggable and allows users to bring in a networking capability of choice or change it later without lock-in.

Kubernetes provides standard networking capabilities that allow accessing the applications from the outside along with networking policies to control communication, service mesh capability and more. We will cover them in detail in *Chapter 5, Container Orchestration with Kubernetes*.

Security

Container orchestration platforms like Kubernetes follow the 4 C's (code, container, cluster and cloud) of cloud-native security, which suggests a layered approach to security.

Kubernetes provides various security features for implementing security policies, role-based access control (RBAC), secrets, and configuration management, etc. It also publishes various guides and best practices. Its extensibility allows users to bring in security tools that address a security need. There are hundreds of such security tools available today, specifically for Kubernetes. Automation makes consistent implementation of security policies across the platform easier, which was seen as a challenge with stand-alone containers. For example, a service mesh can be

used to secure communication between services, or a declarative network policy can be applied.

Here, we are simply generalizing these features and will cover some of them in later chapters.

Managed Kubernetes platforms

The design of Kubernetes and its current maturity state make it complex. Designing, deploying, and maintaining a Kubernetes platform for a large setup has its own challenges, especially given the number of features and capabilities it supports and its pluggable architecture. For many organizations/startups, this takes away the focus from application development and can also be a costly affair.

This is where managed Kubernetes platforms come to the rescue and have been popular as well. Cloud providers such as Microsoft, Amazon, Google, Oracle, and others provide managed Kubernetes services, where they manage the installation and subsequent operations of the platform, leaving the users to simply use it. This is ideal for developers.

A few examples of such services are Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS) and Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE).

Other container orchestrators, such as *OpenShift* are also available as a managed solution by Red Hat in collaboration with the cloud providers.

Multi-cloud/Hybrid Kubernetes

An emerging trend is to run Kubernetes clusters across multiple cloud platforms and/or hybrid environments. Services, such as *Google Cloud Anthos* and *Azure Arc* support these architectures for Kubernetes.

Lightweight Kubernetes variants

Due to its popularity users are looking at running Kubernetes everywhere. Not every installation involves multiple applications, services, complex networking and storage requirements, hence not requiring a multi-node complex setup. This is where lightweight Kubernetes distributions are useful.

The following are some of the popular lightweight Kubernetes distributions and their use cases:

- **Minikube:** This was developed by the Kubernetes project and can run on Windows, Linux and MacOS. It uses virtualization in the background to create nodes and creates a single-node cluster by default but supports a multi-node configuration as well. It is ideal for testing or POC scenarios only and is not recommended for production.
- **K3s:** This distribution was developed at Rancher (SUSE), an Enterprise Kubernetes platform. It is designed for use in a single node setup and was built for IoT/Edge scenarios in a production environment. For example, it can be run on Raspberry Pi. It can also run on Windows, Linux, and MacOS.
- **MicroK8s:** It is from Canonical (that provides Ubuntu) and is also a production grade Kubernetes distribution. It uses vanilla Kubernetes and provides HA by default. Ideal for development, edge/IoT, it can also run on Windows, Linux, and MacOS. It provides the maximum features among the three.

Some other new entrants include:

- Red Hat Device Edge
- AKS Lite from Azure

Conclusion

Containers package software applications and their dependencies into isolated units, ensuring consistent execution across different environments. This portability, along with improved resource utilization, faster

deployment cycles, and enhanced scalability, makes containers a cornerstone of modern cloud-native architectures.

A container orchestrator automates the deployment, scaling, and management of containers (containerized applications), addressing the challenges of managing multiple containers at scale.

Open standards such as OCI, CNI, CRI, and CSI enable portability and interoperability and avoid vendor lock-in between different container orchestrators. Kubernetes is one such container orchestrator with wide adoption across industries.

In the next chapter, we will gain some practical experience on using Docker and the containerd runtime before moving on to Kubernetes.

Multiple choice questions

1. Which of the following Linux features provides isolation in containers? Select all that apply.
 - a. Namespaces
 - b. cGroups
 - c. SELinux
 - d. iptables
 - e. Both a and b
2. Which of the following would you say are advantages of running applications in containers? Select all that apply
 - a. Smaller footprint
 - b. Better dependency management
 - c. Efficient resource utilization
 - d. Faster bootup
 - e. All of the above

3. Which of the following is required to build a container image?
 - a. .env
 - b. docker-compose.yaml
 - c. docker.sock
 - d. Dockerfile
4. Which of the following would you use to publish and share container images?
 - a. Container registry
 - b. Docker Hub
 - c. Docker Runtime
 - d. Container orchestrator
5. Containers on the same host share the underlying Kernel.
True or false
 - a. True
 - b. False
6. Which of the following container runtime types provide the least security?
 - a. Virtualized container runtimes
 - b. Proxy container runtimes
 - c. Namespaced container runtimes
 - d. Paravirtualized container runtimes
7. Which of the following present in a container orchestrator compute node is responsible for running containers?
 - a. Container orchestrator client
 - b. Container runtime
 - c. Container orchestrator daemon

- d. Orchestrator runtime
8. Which of the following is not an open standard hosted by CNCF?
- a. Container Runtime Interface
 - b. Container Storage Interface
 - c. Container Networking Interface
 - d. Application Programming Interface
9. Which of the following specs does OCI provide to standardize low-level container runtimes:
- a. Image and runtime
 - b. Distribution and runtime
 - c. Image, distribution and runtime
 - d. Runtime only
10. Which of the following open standards makes the container runtime pluggable, allowing one to bring in a runtime of choice?
- a. Open Container Initiative
 - b. Container Storage Interface
 - c. Container Runtime Interface
 - d. Container Networking Interface
11. Which of the following are benefits of using volumes in containers?
- a. Volumes provide automatic encryption of data
 - b. Volumes persist data beyond container lifecycle
 - c. Volumes allow sharing of data between containers
 - d. Volumes provide automatic backup capabilities
 - e. Both b and c

12. Which of the following is the default networking type in Docker containers?
- a. Bridged
 - b. Host
 - c. None
 - d. Overlay
13. Which of the following are security concerns when running containers? Select all that apply.
- a. Containers running as root
 - b. Using unsigned images as base images
 - c. Running all containers on the same container network
 - d. Shared kernel design of containers
 - e. All of the above
14. Select the core functions that a container orchestrator provides. Select all that apply.
- a. Scheduling
 - b. Self-healing
 - c. Scaling
 - d. State management
 - e. All of the above
15. Which of the following is the default container runtime in Kubernetes?
- a. CRI-O
 - b. Containerd
 - c. Docker
 - d. gVisor

16. It is mandatory to use runc as the low-level runtime in Containerd. True or false.
- False
 - True
17. When using container orchestrators, one does not have to think about security as they are secure by default. True or false.
- True
 - False
18. Which of the following is a networking plugin that implements CNI?
- Flannel
 - Overlay
 - Linkerd
 - OpenShift
19. Which of the following can help with advanced observability, traffic policy shaping and securing network endpoints?
- A container runtime such as Containerd
 - A CNI plugin such as Weave
 - A linux security feature such as SELinux
 - A service mesh such as Istio
20. Which of the following is a lightweight Kubernetes distribution purpose built for IoT/Edge scenarios?
- MicroK8s
 - K3s
 - Minikube

d. AKS Lite

Answers

1	e.
2	e.
3	d.
4	a.
5	a.
6	c.
7	b.
8	d.
9	c.
10	c.
11	e.
12	a.
13	e.
14	e.
15	b.
16	a.
17	b.
18	a.
19	d.
20	b.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Hands-on Docker and Containerd

Introduction

The third chapter of this book guides the reader through a practical demonstration of running containers using *Docker*. This chapter will also cover hands-on exercises on interacting with containers using *containerd*. A practical experience of *containerd* may be helpful in troubleshooting of Kubernetes installations.

Practical knowledge of *Docker* or *containerd* is not necessary for the KCNA exam, hence this chapter can be skipped if you are in a time crunch and have hands-on experience managing containers using Docker already.

Note: Do not copy and paste the commands from the book (especially when using the e-book format). This may break the command, for example, by adding additional spaces or not running a multi-line command as a single command. Typing the commands is a better way to learn, which also helps with retention.

Structure

This chapter has two major sections, Docker and *containerd*. The following exercises are covered along with a quiz at the end:

- Hands-on Docker
- Hands-on *containerd*

Objectives

This chapter aims to provide a hands-on experience of using docker commands to build, run, and manage containers. It provides step-by-step examples on how to build container images using Dockerfile, push them to a public registry, create a container from the image and manage them later. This chapter will also provide a similar step-by-step approach to creating and managing containers directly with containerd using the ctr and the nerdctl CLI tools. By the end, readers will be familiar with container management using command line tools for both Docker and containerd.

Hands-on Docker

Docker has been the de facto standard when building, publishing and managing container images and for running standalone containers (including multiple containers in an unorchestrated manner). Docker commands continue to be used for image management even in Kubernetes environments.

In this section, we will gain practical experience running docker commands through a series of exercises demonstrating how to containerize and run an application using standalone containers.

Setting up Docker

Depending on the operating system, there are various ways to install Docker. The most popular approach, especially during development and/or learning, is to use the **Docker Desktop** offering, which is available for Linux, Windows, and Mac.

However, for this chapter, we will use an online sandbox/playground called **Play with Docker** that is available at no cost and is easier to get started with. It is accessible at <https://labs.play-with-docker.com/>.

If you wish to install Docker on a machine of your choice for the exercises, use one of the following methods to do so:

1. To install Docker Desktop on your machine:
 - a. Mac: <https://docs.docker.com/desktop/install/mac-install/>
 - b. Windows: <https://docs.docker.com/desktop/install/windows-install/>

- c. Linux: <https://docs.docker.com/desktop/install/linux-install/>
2. To install Docker Engine only on one of the supported Linux distros, start here at <https://docs.docker.com/engine/install/>.

For this book, we will not discuss the installation steps of each method. Instead, use the links provided previously and follow the instructions.

Prerequisites

As mentioned earlier, this chapter will use *Play with Docker* for the exercises. One of the important pre-requisites to use it is to have an account with Docker Hub (<https://hub.docker.com/>). Follow the given instructions to create an account if you do not have one already:

1. Visit <https://hub.docker.com> and select one of the sign-up options. You can use an existing *Google* or *GitHub* account to sign up or also sign up using the email-based approach. The following figure shows the landing page of Docker Hub and the sign-up options available as of this writing:

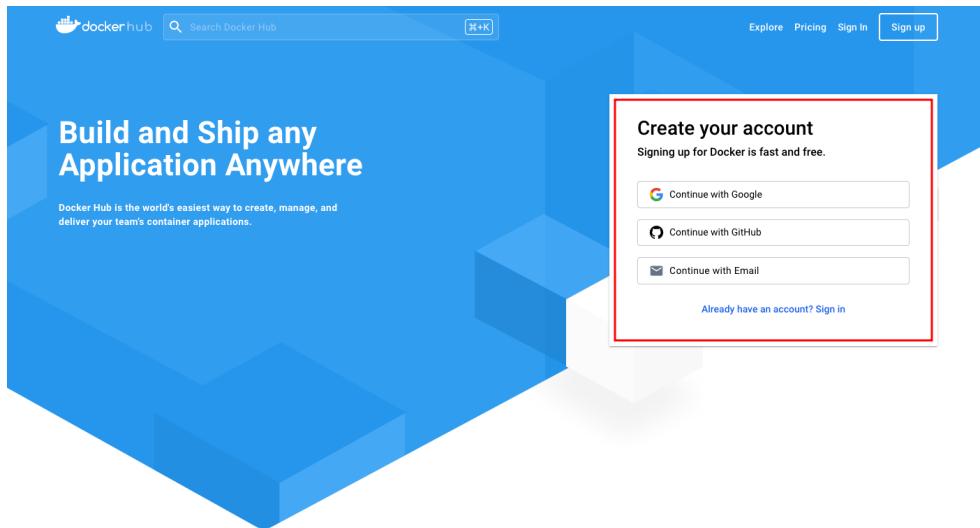
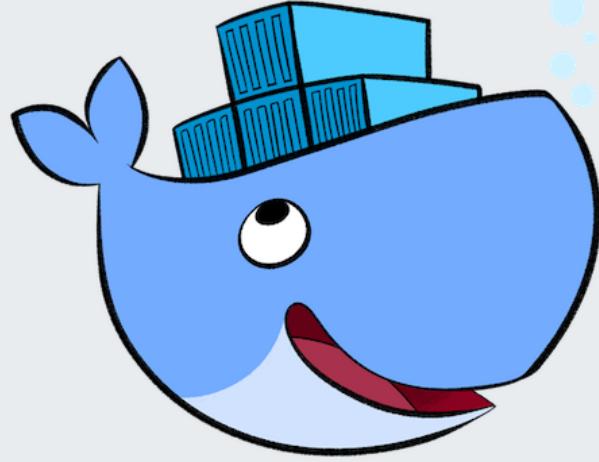


Figure 3.1: Landing page of <https://hub.docker.com/>

2. Next, sign in to Play with Docker. To do so, navigate to <https://labs.play-with-docker.com/>, click Login | Docker, and complete the login process. This is shown in the following figure:



Play with Docker

A simple, interactive and fun playground to learn Docker

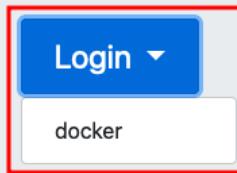


Figure 3.2: Logging in to Play with Docker using Docker Hub

On successful login, the **Start** button is activated. Click **Start** to begin. This activates a session for 4 hours.

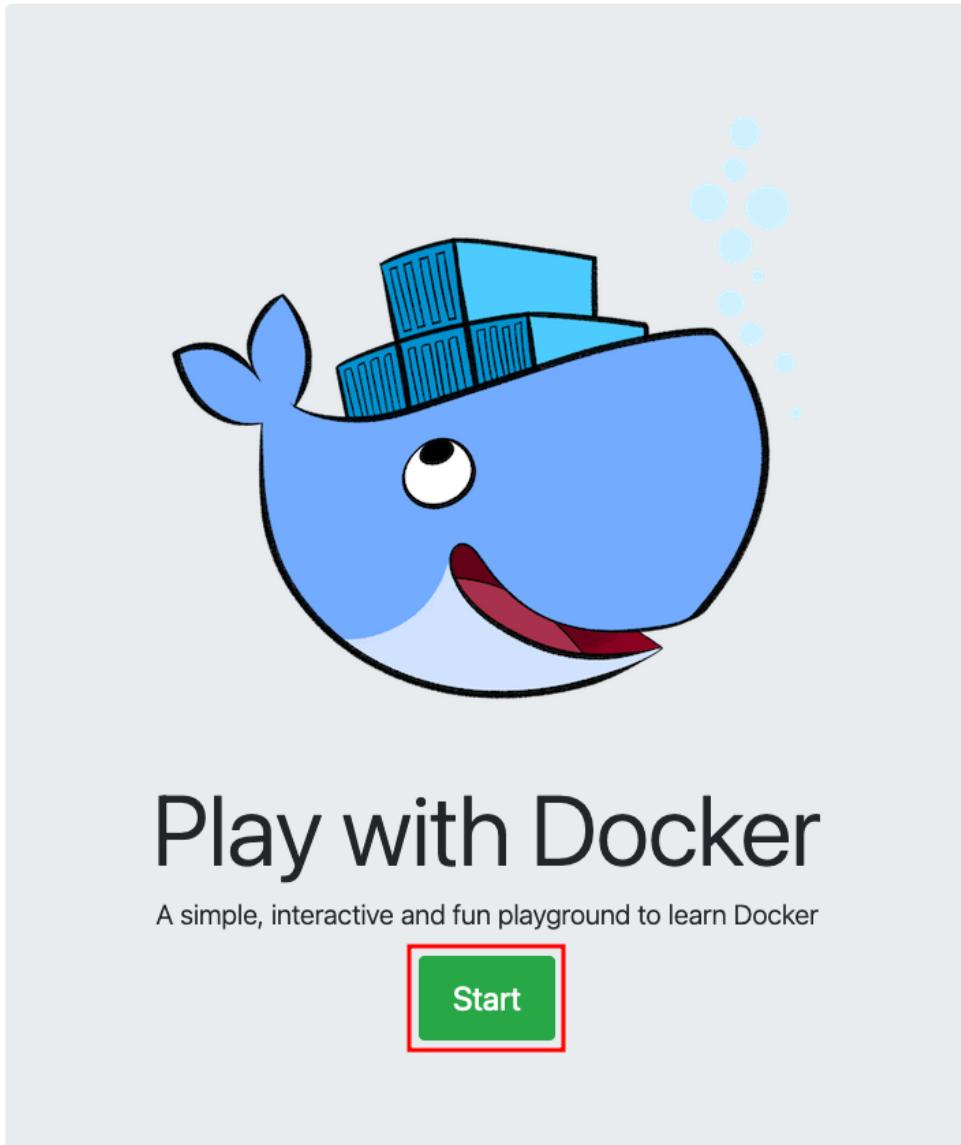


Figure 3.3: A successful login displaying the Start button

3. Initially, the session has no Docker instances. Click + ADD NEW INSTANCE to create a new instance, which is shown as follows:

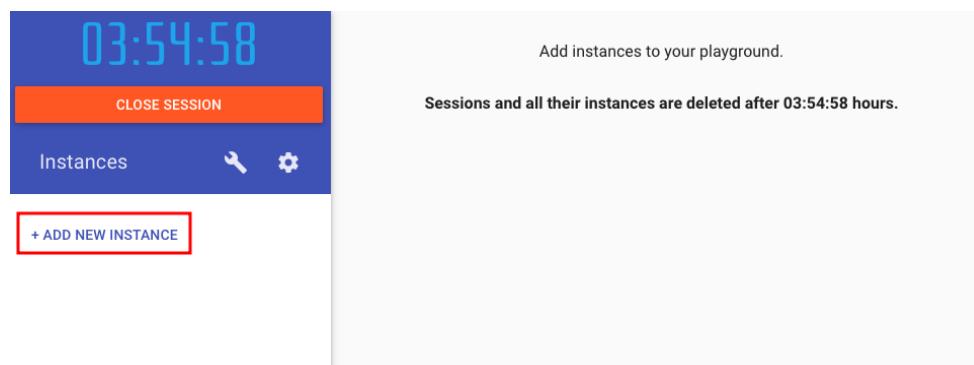


Figure 3.4: Adding a new Docker instance to the session

4. This should activate a docker instance, as shown in the following figure:

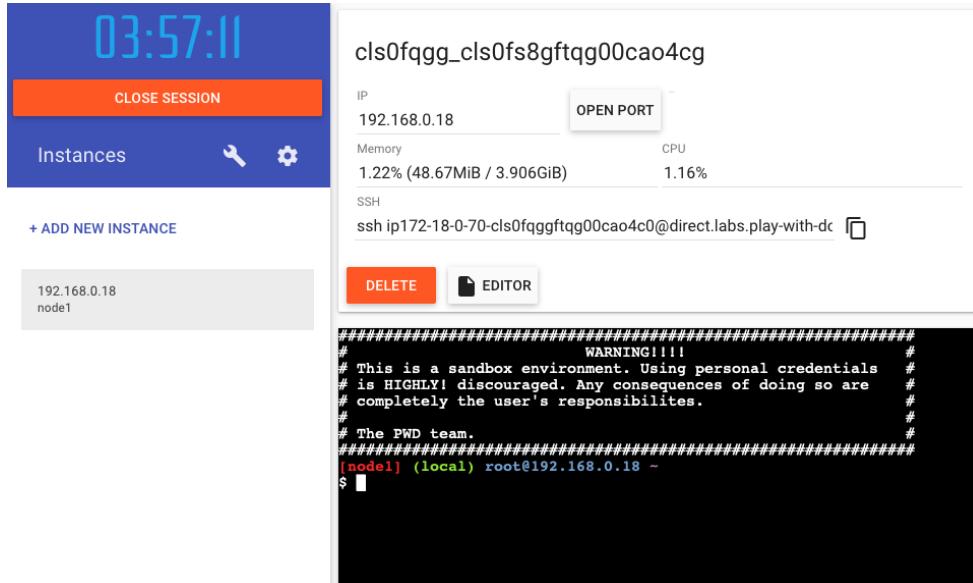


Figure 3.5: Successful creation of a Docker instance

The sample code used here is based on *Node.js* and is an adaptation of the *getting started app by Docker*. Knowledge of Node.js is not necessary for this hands-on.

Creating a Dockerfile

As discussed in the previous chapter, before we can create and run a container, we need to create a container image. To do so, a *Dockerfile* is created, which contains the necessary steps for building the image:

1. Start by cloning the following repository, which contains the sample app:

```
git clone https://github.com/sangramrath/2170.git
```

2. Change to **CH03** directory and review the contents (the sample application code) if necessary:

```
cd 2170/CH03
```

3. To create the Dockerfile, start the **vi** editor:

```
vi Dockerfile
```

4. Next, paste/type the following code. The comment section before each step describes what it does. When done, save and exit using: **wq** followed by **Enter**:

Note: the completed Dockerfile is also available in the solutions inside the root of the repository and can be copied to this folder

```
# Pull a base image
FROM node:18-alpine

# Set a working directory
WORKDIR /app

# Copy the app code into the working directory
COPY . .

# Install dependencies
RUN yarn install --production

# Start app
CMD ["node", "src/index.js"]

# Define the port on which the app is listening
EXPOSE 3000
```

5. We are now ready to build the image. The **docker build** command is used for this.

Note: There is a dot '.' at the end of the command.

docker build -t kcnaprep:1.0 .

The command will do the following:

- a. All the image layers that make up the base image **node:18-alpine** will be downloaded.
- b. Additional layers will be created for each of the image build steps mentioned in the *Dockerfile*.

- c. The **-t** flag in the command tags the container image with the name provided.
 - d. The '.' at the end of the command specifies the **docker build** command to look for the Dockerfile in the current directory.
6. Verify that the image was created by running the **docker images** command:

docker images

The following figure shows an example of the output.

[node1] (local) root@192.168.0.8 ~/2170/CH03				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
kcnaprep	1.0	ceef1466534d	15 seconds ago	221MB

Figure 3.6: Output of the docker images command

Creating and running a container

In the following steps, we will start a container from the image created earlier and verify we can access the application running in it.

1. To create and run a container, we can use a single command, **docker run**.

```
docker run --name kcnaprep-app -d -p 3000:3000 kcnaprep:1.0
```

The following command can be broken down into the following parts:

- The **docker run** command creates and runs the new container with the provided parameters.
- **--name** assigns a name to the container.
- **-d** runs the container in the background. It means run in detached mode. One can also use **--detach**.
- **-p** enables a container to be accessed from outside. It stands for publish and publishes a container's port to the host.
- **kcnaprep:1.0** is the image from which the container is created. The docker command looks for the image locally and if it is not present tries to download it from the docker hub by default. In this particular example, the image is available locally.

- Verify that the container is running using the **docker ps** command:

```
docker ps
```

The following figure shows an example of the output:

```
[node1] (local) root@192.168.0.8 ~/2170/CH03
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
695dcb9bfa60 kcnaprep:1.0 "docker-entrypoint.s..." 7 seconds ago Up 6 seconds 0.0.0.0:3000->3000/tcp kcnaprep-app
```

Figure 3.7: Output of the docker ps command

- Access the application by navigating to the unique URL for your instance. Click the port number, i.e., **3000**, which should now be visible at the top of the page. This is illustrated in the following figure:

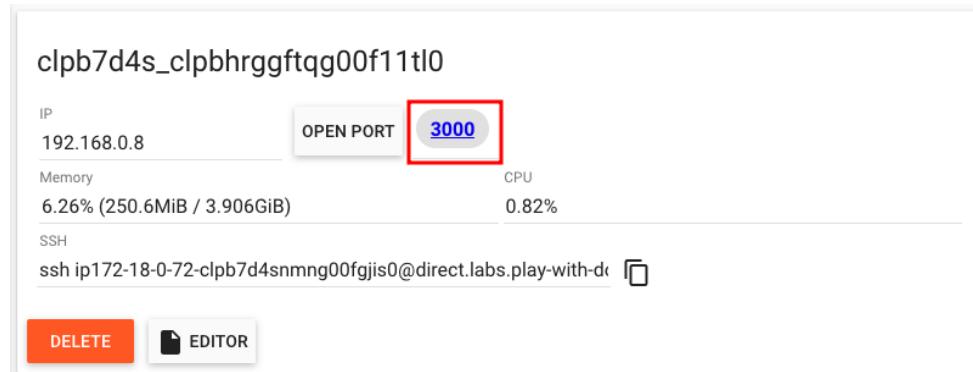


Figure 3.8: Accessing the sample application after running the container

If the preceding port number is not automatically visible, click **OPEN PORT** button, type **3000**, and click **OK**.

- This should load the app on the browser. Add a few links by pasting the link one at a time and clicking **Add Link**. The following image shows an example of it:

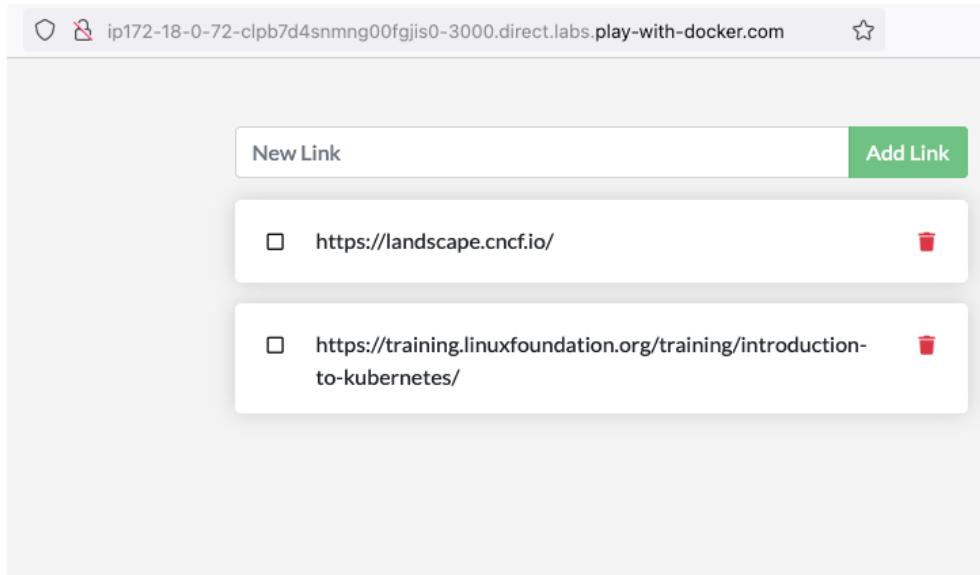


Figure 3.9: Screenshot displaying a working application

Additional container operations

In this section of the Docker exercises, you will learn about docker commands for common operations such as stop, start, remove, etc.

1. To stop a container, use the following **docker stop** command:

```
docker stop kcnaprep-app
```

2. To start a container, we use the **docker start** command.

```
docker start kcnaprep-app
```

3. To remove a container, use the **docker rm** command. The container must be stopped before removing. To remove forcefully, use the **-f** switch.

```
docker stop kcnaprep-app
```

```
docker rm kcnaprep-app
```

Or,

```
docker rm kcnaprep-app -f
```

4. As a final step, create a new container again using the same **docker run** command as earlier. Notice how the links added earlier are missing. This is because containers are stateless, and any changes made, or data

written to the read/write layer are lost when a container is deleted. We will address this problem in the *Persisting Data* section.

Working with images and image registry

In this section, you will learn how to push the local image to a public image registry, i.e., Docker Hub. You will also learn how to download images from the image registry.

Before continuing, ensure you have an account with Docker Hub:

1. Navigate to <https://hub.docker.com/>, and sign-in if needed. Make a note of your username (this will be required later). Next, click **Create repository**, as shown in the following figure:

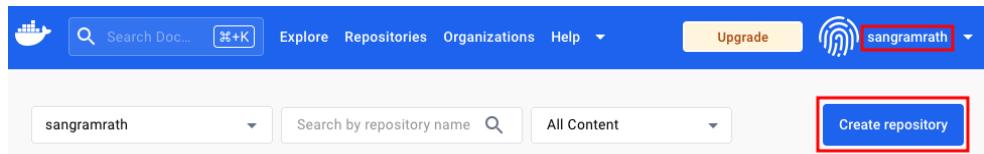


Figure 3.10: Retrieving the Docker Hub username

2. Enter a name for the repository, use **kcnaprep**. Click **Create** to create the repository, which is shown as follows:

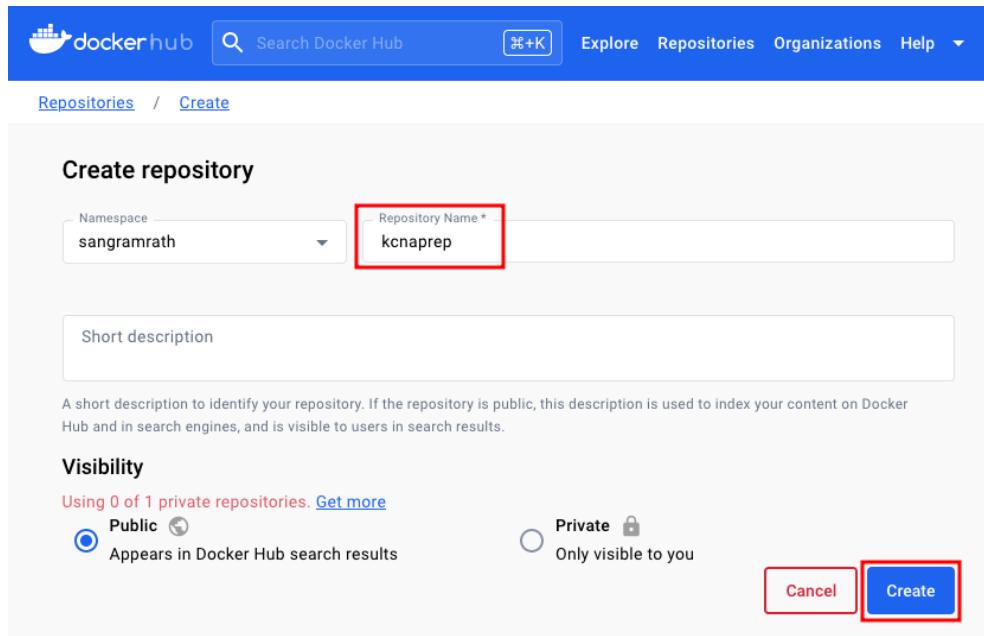


Figure 3.11: Creating a repository in Docker Hub

You are now ready to push images to it.

3. Head back to the Docker instance session at <https://labs.play-with-docker.com/>. The following steps are a continuation from the earlier section, and hence, expect that the container image created earlier will be available locally.
4. Before we can push images, we must tag them with the repository created with Docker Hub. To do so, we use the **docker tag** command. Run the following command to tag the local image replacing the **USERNAME** with your actual username:

```
docker tag kcnaprep:1.0 USERNAME/kcnaprep:1.0
```

5. Run **docker images** to verify that the local repository has an image with the new tag:

```
docker images
```

The output is as follows:

[node1] (local) root@192.168.0.8 ~/2170/CH03				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
kcnaprep	1.0	ceef1466534d	30 minutes ago	221MB
sangramrath/kcnaprep	1.0	ceef1466534d	30 minutes ago	221MB

Figure 3.12: Output of the docker images command

6. Next, login to Docker Hub from the command line. Execute the **docker login** command followed by your username and password:

```
docker login
```

7. To push the image, use the **docker push** command followed by the image tagged with the Docker Hub repository:

```
docker push USERNAME/kcnaprep:1.0
```

8. Go to your Docker Hub repository at:

<https://hub.docker.com/repository/docker/USERNAME/kcnaprep/general> and verify that you can see the image. Replace **USERNAME** with your actual username. The following figure shows an example of the pushed image in the docker hub repository:

The screenshot shows a Docker Hub repository page for 'sangramrath / kcnaprep'. The 'General' tab is selected. At the top, there's a note to 'Add a short description for this repository' with an 'Update' button. Below that, the repository name 'sangramrath / kcnaprep' is displayed. A 'Description' section indicates it has no description. A 'Docker commands' section shows the command 'docker push sangramrath/kcnaprep:tagname'. The 'Tags' section lists one tag, '1.0', which is highlighted with a red border. The 'Automated Builds' section is present but inactive.

Figure 3.13: A successfully pushed image along with its tag

9. We will now test pulling this image and running a container. To do this we will simulate a fresh environment. In the <https://labs.play-with-docker.com/> portal, add a new instance by clicking + ADD NEW INSTANCE.
 10. Pull the docker image from Docker Hub using the **docker pull** command:
- docker pull USERNAME/kcnaprep:1.0**
11. Run a container with the **docker run** command. Notice the image name has changed:
- docker run --name kcnaprep-app -d -p 3000:3000 USERNAME/kcnaprep:1.0**

Note: The list of links will be empty because it is a new container deployment and we are not persisting the data outside of the container.

You can delete this instance in *Play with Docker* by clicking **DELETE**. Ensure you are deleting the new instance created to test.

Persisting data with Docker volumes

So far, the links that you add are written to an SQLite DB that is saved inside the container in a folder called **/links**. If the container were to be stopped, removed, and then recreated, the list of links would be empty. Similarly, if there was a host crash all the data would be lost. In this section, we will look at options to persist the data in a container.

As discussed in the previous chapter, Docker provides two options to persist data. We will quickly look at both:

1. To create a volume, we use the following **docker volume** command:

```
docker volume create kcnaprep-db
```

2. Start a new container with a volume mount. The **docker run** command will include the **--mount** flag with certain parameters. The source is the volume name, and the target is a folder inside the container:

```
docker run --name kcnaprep-app -d -p 3000:3000 \
--mount type=volume,src=kcnaprep-db,target=/app/data k
cnaprep:1.0
```

3. Next, access the web app and add a few links.
4. To verify data persistence, stop and remove the container using commands discussed earlier. Then recreate a new container with the existing volume; a new container will have a different container ID. You can simply run the same command in *Step 2*. Following is the sequence of commands for reference:

```
docker stop kcnaprep-app
```

```
docker rm kcnaprep-app
```

```
docker run --name kcnaprep-app -d -p 3000:3000 \
--mount type=volume,src=kcnaprep-db,target=/app/data k
cnaprep:1.0
```

5. Launch the web app and verify that the links created in *Step 3* are present.
6. Inspect the volume using the **docker volume inspect** command:

```
docker volume inspect kcnaprep-db
```

7. Stop and remove the container before continuing:

```
docker stop kcnaprep-app
```

```
docker rm kcnaprep-app
```

8. Remove the volume as well, which is shown as follows:

```
docker volume delete kcnaprep-db
```

Persisting data with bind mount

We can also achieve data persistence using **bind mounts**. The following steps show the alternative method:

1. Start a container with the following command. The **--mount** flag in the command uses the type **bind** which specifies that we are using the bind mount approach. The rest of the syntax for **--mount** remains unchanged:

```
docker run --name kcnaprep-app -d -p 3000:3000 \
--mount type=bind,src=./data,target=/app/data kcnapre
p:1.0
```

2. Next, access the web app and add a few links.
3. To verify persistence, remove the container and recreate a new one with the same bind mount. You can also simply rerun the command in *Step 1*:

```
docker rm kcnaprep-app -f
docker run --name kcnaprep-app -d -p 3000:3000 \
--mount type=bind,src=./data,target=/app/data kcnapre
p:1.0
```

4. Access web app again and verify that the data persists, i.e., the same links added earlier are present.
5. Remove the container before continuing.

```
docker rm kcnaprep-app -f
```

Networks in Docker

Let us learn how to create networks in Docker in the following steps:

1. Start by creating a network using the following **docker network** command:

```
docker network create kcnaprep-net
```

2. Verify the network is created and also view the default networks shipped with docker installation using the following command:

```
docker network ls
```

Creating multiple containers on the same network

In the following steps, we will create containers attached to the network created earlier:

1. Create and start a MySQL container attached to the **kcnaprep-net** network:

```
docker run --name kcnaprep-db -d -p \
--network kcnaprep-net --network-alias mysql \
-v kcnaprep-mysql-data:/var/lib/mysql \
-e MYSQL_ROOT_PASSWORD=Pa55w.rd -e MYSQL_DATABASE=links \
mysql:8.0
```

2. Create and start the web app container attached to the same network:

```
docker run --name kcnaprep-app -d -p 3000:3000 \
--network kcnaprep-net \
-e MYSQL_HOST=mysql -e MYSQL_USER=root \
-e MYSQL_PASSWORD=Pa55w.rd -e MYSQL_DB=links \
kcnaprep:1.0
```

3. Verify that two containers are running and are connected to the same network:

```
docker ps --filter network=kcnaprep-net
```

With this, you have successfully deployed a two-tier app using containers implementing networking and persistent storage.

Cleanup

In this final exercise, you will delete all resources created so far including containers, volumes and networks created in the previous section. Additionally,

you will also learn how to remove the images you created at the beginning of this exercise.

Note: Do not delete the image from Docker Hub.

Refer to the following steps to begin removing the resources:

1. Remove previously created containers. Either stop and then remove them or use the **-f** flag to remove them by force. Both examples are again demonstrated as follows:

```
docker stop kcnaprep-db
```

```
docker rm kcnaprep-db
```

```
docker rm kcnaprep-app -f
```

2. List any remaining containers and delete them as well. If needed, refer to the previous exercises for the command to list containers.
3. Delete the container image built initially and stored locally:

```
docker rmi kcnaprep:1.0
```

4. Delete the image tagged for pushing to Docker Hub. Replace **USERNAME** with your actual username. Also, check if there are any other images and delete them as well:

```
docker rmi USERNAME/kcnaprep:1.0
```

5. Delete the volume created in the previous step. Additionally, check if there are any more volumes and delete them as well. Ensure the containers using them are deleted first:

```
docker volume delete kcnaprep-mysql-data
```

```
# Use the following command to list docker volumes
```

```
docker volumes
```

6. Delete the network:

```
docker network delete kcnaprep-net
```

Hands-on containerd

In this section, we will perform a few hands-on exercises on the containerd runtime. While one will most likely not install and use containerd as a stand-

alone container runtime like they would with Docker, being familiar with **ctr** and other commands can help when troubleshooting Kubernetes installations.

Containerd comes bundled with a debugging utility called the **ctr**. We will learn to interact with containerd using it. We will also learn how to interact with containerd using another utility called **nerdctl**.

Prerequisites

In order to follow the exercises, you will need a modern Ubuntu system with at least 1 vCPU and 1 GB RAM.

Ubuntu 20.04 was used in this example.

Installing containerd

There are multiple ways to install containerd such as from binaries, from source or from packages. Containerd can be installed on various CPU architectures and their respective binaries are available in its GitHub repo. For more information on this visit <https://github.com/containerd/containerd/releases>.

We will use the package-based method to install containerd. The **containerd.io** packages are made available by Docker in DEB and RPM formats. This is simpler and installs all other required components such as **runc** and CNI plugins.

In the following steps, you will install containerd using apt and verify that the **ctr** command works. Installing **containerd** also installs the **ctr** utility:

1. At the terminal, run the following command to install **containerd**:

```
sudo apt update && sudo apt install containerd -y
```

2. Next, verify that **containerd** was installed, is running, and the default CLI works by running the following commands:

```
sudo systemctl status containerd.service  
# Check containerd version  
sudo ctr --version
```

Working with container images

In the following steps, you will download the image pushed earlier to Docker Hub. This image was created and pushed in the Docker exercises:

1. To pull an image from docker hub, we can use the **ctr images pull** command. Run the following command to do so. Replace **USERNAME** with your actual username from the Docker Hub:

```
sudo ctr images pull docker.io/USERNAME/kcnaprep:1.0
```

2. List all images available locally:

```
sudo ctr images ls
```

Creating and managing containers

You are now ready to run a container using the image downloaded earlier. In this step, you will learn how to **use** the **ctr** command to create the container, verify that the container is running, and how to connect to the container. You will also learn how to delete the container:

1. To create and start a container from the image, we can use the **ctr run** command. This command is actually an alias for two commands, **ctr container create** and **ctr task start**. In containerd the **ctr container create** command is used to create the container, but this does not start the container. Then, the **ctr task start** command is used to start the container, similar to **docker start**. It is also important to note that in containerd, providing a container name (it is called an ID) is necessary. For this command, use **ctr** to download the image from your repository at Docker Hub. Replace **USERNAME** with your actual username from the Docker Hub:

```
sudo ctr run -d docker.io/USERNAME/kcnaprep:1.0 kcnaprep-app
```

2. Verify that the container is created using the following command:

```
sudo ctr container ls
```

3. Note that the **container ls** command does not tell you if the container is running. To verify if a container is running, use the **ctr task ls** command:

```
sudo ctr task ls
```

4. To connect to the container, use the **ctr task exec** command. It requires the **--exec-id** flag followed by an ID (or name) for it:

```
sudo ctr task exec -t --exec-id bash1 kcnaprep-app bash
```

5. To delete/remove a container using the **ctr** command, one must kill and remove the task first before removing the container. Attempting to remove the container directly will result in an error:

```
# Kill the task
```

```
sudo ctr task kill kcnaprep-app
```

```
# Remove the task
```

```
sudo ctr task rm kcnaprep-app
```

Alternatively, you can run the **ctr task rm** command with the **--force** flag to do both in one command:

```
sudo ctr task rm kcnaprep-app --force
```

Note: The **rm** command can be replaced by **remove**, **delete** or **del**.

6. Next, remove the container with the following command:

```
sudo ctr container rm kcnaprep-app
```

Storage and networking using the **ctr** utility

The **ctr** command does not support working with volumes or networks like the **docker** command. From the previous chapter, we know that **containerd** is a high-level runtime that is CRI compatible which means its primary focus is image and container management only.

It is also not possible to publish (expose) ports using **ctr** command. However, there is a utility called **nerdctl**, that one can use that supports more commands. We will quickly look into it next before ending this chapter.

The **nerdctl** utility

The debugging utility, **ctr**, is incompatible with Docker CLI and hence not a very friendly utility to experiment with all features of **containerd**, including experimenting with new and cutting-edge features of **containerd**.

The **nerdctl** utility is a non-core sub-project of the containerd project. It is a CLI that works similarly to Docker and supports most of the docker command equivalents. One can simply replace the **docker** command with **nerdctl** and they would work. Apart from having a similar UI/UX as Docker CLI, one of the important highlights is that it supports rootless node, which allows creating and running containers that do not have root privileges.

It is not the core objective of this chapter to explore all commands of **nerdctl**. We will run only a few of them demonstrating their capabilities. This part of the exercise is deliberately kept short.

Note: It is assumed that you are using the same VM/instance that was used in the earlier **ctr** exercise. If not, ensure that containerd is installed before continuing further.

In order to experience most of the command capabilities of **nerdctl**, the following requirements must be in place:

- Containerd, the runtime.
- CNI Plugins (for networks such as bridge network).
- BuiltKit for building images.
- RootlessKit and slirp4netns for supporting rootless mode.

The following installation process takes care of all the networking requirements for this exercise. Given the scope of this exercise, we are not exploring rootless mode and hence the **nerdctl** commands use **sudo** to run in privileged mode;

1. Start by running the following commands in sequence to download, extract and copy the binaries for use. The comments prior to the command describe what it does. The latest release as of this writing is **v1.7.21**, you can visit the GitHub repo in the link for the latest release:

```
# Download the nerdctl release
wget https://github.com/containerd/nerdctl/releases/download/v1.7.2/nerdctl-full-1.7.2-linux-amd64.tar.gz

# Extract and copy the binaries
```

```
sudo tar Cxzvvf /usr/local nerdctl-full-1.7.2-linux-amd64.tar.gz
```

```
# Verify nerdctl works
```

```
sudo nerdctl version
```

2. List all images available locally and optionally, pull an image from docker hub. Replace **USERNAME** with your actual username from the Docker Hub:

```
sudo nerdctl images
```

```
# Run the following command only if a local image is missing
```

```
sudo nerdctl pull docker.io/USERNAME/kcnapprep:1.0
```

3. Next, create a volume for data persistence.

```
sudo nerdctl volume create kcnapprep-db
```

4. Create and run a new container with persistent volume, published on a port using the **nerdctl run** command:

```
sudo nerdctl run --name kcnapprep-app -d -p 3000:3000 \
-v kcnapprep-db USERNAME/kcnapprep:1.0
```

5. Verify that the container is running:

```
sudo nerdctl ps
```

6. Access the app locally or through the host's port to verify connectivity.
7. Connect to the container using **nerdctl exec**. Then, type **exit** to exit out of the container:

```
sudo nerdctl exec -it kcnapprep-app /bin/sh
```

8. Finally, remove the container, image, and volume with the respective commands mentioned as follows:

```
# Stop and remove the container
```

```
sudo nerdctl stop kcnapprep-app
```

```
sudo nerdctl rm kcnapprep-app
```

```
# Remove the image
```

```
sudo nerdctl rmi USERNAME/kcnaprep-app:1.0
```

```
# Remove the volume
```

```
sudo nerdctl volume remove kcnaprep-db
```

As noticed previously, the commands are the equivalent of Docker commands.

For a complete list of command and for further learning, visit:

<https://github.com/containerd/nerdctl/blob/main/docs/command-reference.md>.

Conclusion

In this chapter, we discussed how to use Docker and containerd to create and manage stand-alone containers. Docker can be installed either using the Docker Desktop approach or the Docker Engine approach. The Docker Desktop method is preferred as it is packaged as a virtual machine and provides a graphical user interface along with many additional features and extensions making it a complete platform developers can use from development to deployment. Additionally, Docker Desktop can be installed on all major operating system platforms such as Windows, MacOS and Linux whereas the installation of Docker Engine is supported on Linux based platforms only.

A Dockerfile is required to build container images (or also called Docker images) using the **docker build** command which is then published to a container registry with the **docker push** command. To download the image, you will use the **docker pull** command.

You can then use **docker run** command to start and run a container from one of the images. Containers are private by default, but they can be exposed outside using the **-p <host-port>:<container-port>** flag. Containers are stateless by design, hence to persist data we use volumes. You will use the **docker volume** command to create and manage volumes and mount them to a container using the **--mount** flag in **docker run** command. To create and manage custom networks you will use the **docker network** command.

Containerd is the default container runtime of Kubernetes and we use the **ctr** command to manage containerd resources. The **ctr image pull** command is

used to download a container image. The **ctr run** command is used to create and run a container. It does not support working with volumes and networks like Docker. Finally, the **nerdctl** command is a great alternative to **ctr** as it has a similar user experience as Docker.

A good understanding of Docker and containerd commands provides a strong foundation for learning advanced Kubernetes, its CRI and in troubleshooting scenarios.

We are now ready to jump to Kubernetes. In the next chapter, we will learn about the architecture of Kubernetes and various common objects that are used to deploy applications such as deployments, pods, services and more.

Multiple choice questions

1. Which command would you run to pull a container image in Docker?
 - a. docker build
 - b. docker pull
 - c. docker create
 - d. docker image
2. What does the '.' at the end of the docker build command signify? Example command: docker build -t kcnaprep:1.0.
 - a. It signifies that it's the end of the command
 - b. It signifies a tagging format for the container image
 - c. It signifies that the built image should be stored in the current directory
 - d. It signifies that the Dockerfile can be found in the current directory
3. What does -p do in a docker run command?
 - a. It prints the container ID.
 - b. It runs the container in the background.
 - c. It publishes a container's port(s) to the host.
 - d. It publishes all exposed ports to random ports

4. Which of the following are the correct commands to run a container in the background?
- docker run --name nginx -d nginx
 - docker run --name nginx nginx
 - docker run --name nginx --detach nginx
 - Both a and c
5. Which of the following is the preferred method to securely persist data when running containers?
- Bind mounts
 - Inside the container
 - Volumes
 - In the image itself
6. Which of the following command would you run to create a volume with using ctr in containerd?
- sudo ctr volume create volumel
 - sudo ctr task volume create volumel
 - sudo ctr task create volumel
 - None of the above
7. Which of the following utility for containerd supports creating containers in rootless mode?
- ctr
 - nerdctl
 - containerd
 - rootlessctl
8. Which of the following flags allows you to forcefully remove a container?
- f
 - d
 - p

- d. -i
9. You want to quickly run nginx in a container for demonstration purposes. Which of the following commands would you run to pull an image if needed and then create, start and run the nginx container? Select all that apply.
- a. docker image pull nginx
 - b. docker create --name nginxdemo nginx
 - c. docker run --name nginxdemo nginx
 - d. docker attach nginxdemo
10. Which flag would you use in the docker run command to connect a container to a specific network?
- a. --net
 - b. --network
 - c. docker network connect
 - d. docker network create

Answers

1	b.
2	d.
3	c.
4	d.
5	c.
6	d.
7	b.
8	a.
9	c.
10	b.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord.\(bpbonline.com](https://discord(bpbonline.com)



CHAPTER 4

Kubernetes Basics

Introduction

This chapter covers the *Kubernetes Fundamentals* domain of the exam. It is a high-weightage section covering 46% of the exam and expects understanding around the following areas:

- Kubernetes resources
- Kubernetes architecture
- Kubernetes API
- Containers
- Scheduling

In this chapter, we will learn about the Kubernetes architecture and the components that make up a cluster. We will cover important Kubernetes resources and how to create and manage them. The chapter will also throw light on topics such as Kubernetes API and how scheduling works.

This chapter will also quickly touch upon the installation options available for Kubernetes but will not cover this in detail since it is out of the exam's scope. We will, however, practice installing a non-production

cluster and how to create resources in *Chapter 9, Hands-on Kubernetes*. A practical knowledge of Kubernetes installation is not necessary for the KCNA exam.

Structure

This chapter covers the following topics:

- Kubernetes architecture
- Control plane
- Worker nodes
- Kubernetes resources
- Creating resources
- Scheduling in Kubernetes
- Kubernetes API

Objectives

The aim of this chapter is to provide theoretical knowledge of Kubernetes, its architecture, and its workings. You will also learn about fundamental Kubernetes resources such as *Pods*, *Deployments*, *StatefulSets*, and *DaemonSets*, to name a few, along with an insight into when to use which. You will then learn how scheduling works in Kubernetes, covering topics such as *Affinity*, *Anti-affinity*, *nodeSelector*, *Taints and Tolerations*, etc.

The chapter will finally introduce Kubernetes API, its use, what API objects are, along with related definitions before discussing how to deconstruct an example Kubernetes API endpoint.

Kubernetes architecture

At this point in the book, you understand the advantages of microservices and running them as containers. You know how to run them using Docker, and through that process, you learned about certain

challenges that stand-alone containers bring or cannot solve and the need for container orchestration. You have also been introduced to Kubernetes, how it can be a container orchestrator, and more. Let us further explore the workings of running containers in Kubernetes.

Before we can begin deploying containerized applications to Kubernetes, we need a Kubernetes cluster.

When you install Kubernetes, you create a cluster. A production Kubernetes cluster typically consists of multiple nodes designated as **control plane** nodes and **worker** nodes. During the installation process, these nodes are converted into selected roles. These nodes run various Kubernetes components that provide different features and capabilities.

Note: The official Kubernetes documentation uses the word **node** to mean worker nodes only. For a better clarification, this chapter and the book uses the word **control plane** nodes and **worker** nodes.

Multiple control nodes are used for high availability, redundancy, and performance. The control plane nodes make up the **control plane** of the cluster. The number of worker nodes depends on the (containerized) workload requirement, for example, the number of Pods that are expected to run and the resource requirements of each Pod and the containers within them. You can always add additional worker nodes very easily.

The following illustration shows a visual representation of a typical Kubernetes architecture with three control plane nodes and three worker nodes along with the components that run on them:

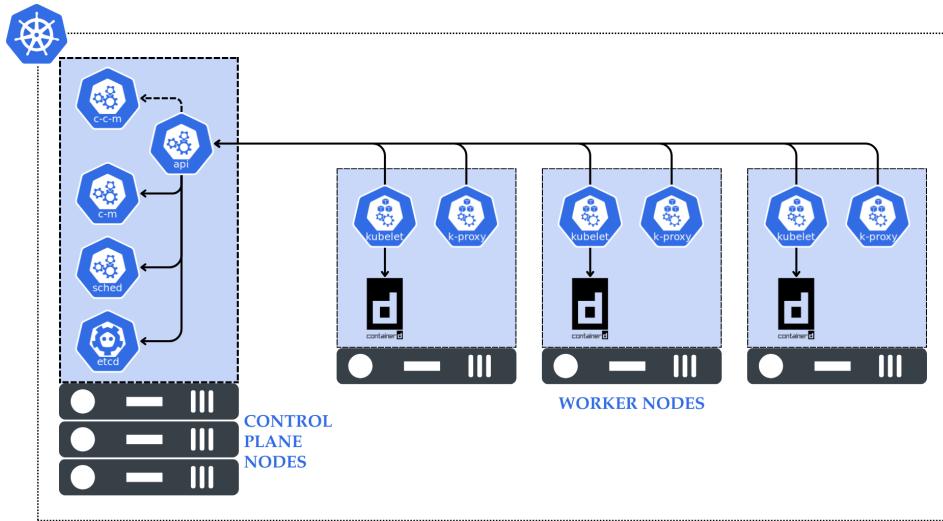


Figure 4.1: Illustration of a typical Kubernetes architecture

The following table further lists and summarizes the components that run on each node type. We will discuss these components individually later in this chapter when discussing the control plane and worker node:

Node type	Component	Function
Control plane nodes	API server	The central management interface that handles all communication with the cluster through RESTful APIs.
	Scheduler (sched)	It assigns Pods to appropriate nodes based on resource requirements, constraints, and scheduling policies.
	Etcd	The key-value store that serves as the cluster's backing store for all configuration data.

	Controller manager (CM)	The component that manages controllers which regulate the state of the cluster.
	Cloud controller manager (CCM)	It helps integrates cloud provider-specific functionality into the cluster control plane
Worker nodes	Kubelet	It is an agent running on each worker node, responsible for running containers in Pods as per the cluster's configuration.
	Kube-proxy (k-proxy)	This component manages networking rules and provides basic load balancing within the cluster.

Table 4.1: Summary of key components in a Kubernetes cluster

Installing Kubernetes

The first step in running your containerized applications in Kubernetes is to install Kubernetes or obtain a managed Kubernetes installation. When you install Kubernetes on a set of nodes, you create a **cluster**. Nodes (or hosts) can be bare-metal servers, virtual machines, cloud VMs, etc.

Various tools can be used to install Kubernetes depending on the scope, i.e., learning, dev and test, production, etc.

For test drive, development, and learning environments such as installing on a desktop OS, the following tools can be used:

- Minikube
- Kind

Both tools will require that a container engine is installed along with **kubectl** as a prerequisite.

Some popular tools used for production Kubernetes Deployments are:

- **Kubeadm**: While popular and more powerful, this installation process involves more prerequisite work, such as ensuring container runtime, **kubectl** and **kubelet** are installed before running **kubeadm**. Also, there is substantial post-initialization work related to networking, storage, etc.
- **Kops**: An easier option compared to **kubeadm** for setting up a production Kubernetes cluster. It supports cloud infrastructure provisioning. It also requires that **kubectl** is installed separately.
- **Kubespray**: Probably the easiest among the three, this is an ansible based playbook along with other bells and whistles to help you set up a Kubernetes cluster.

In short, the amount of work involved in setting up and subsequently managing a production Kubernetes cluster is quite high; hence, managed Kubernetes offerings such as **Azure Kubernetes Service (AKS)**, **Google Kubernetes Engine (GKE)**, etc. are gaining popularity. At the same time, knowledge of Kubernetes installation and administration is a sought-after skill set.

Understanding the detailed installation processes is beyond the scope of the exam and this book. The CKA is a more appropriate exam/certification for this.

The **kubectl** utility

The next puzzle in the Kubernetes architecture is a client or tool to communicate with the cluster. The **kubectl** CLI utility is the piece of this puzzle. It is a command line tool that allows you to communicate and

interact with the Kubernetes cluster and its resources. It uses the Kubernetes API to communicate with the control plane for tasks such as creating and managing resources/objects, managing and troubleshooting the cluster, etc. For example, you can describe a resource, view logs, view events from the cluster, and more. When you run a **kubectl** command, it is translated to an API call and presented to the Kubernetes API for processing.

To successfully authenticate and communicate with a cluster, **kubectl** requires a configuration file containing information about the cluster, such as the API server endpoint, one or more users, namespaces, and, most importantly, the authentication mechanisms. This file is named **config**, and the default location is **\$HOME/.kube**. One can provide a separate location by setting the **KUBECONFIG** environment variable or adding **--kubeconfig** to the **kubectl** command. The config file can contain information about multiple Kubernetes clusters. This file is commonly called **kubeconfig**.

Kubectl can be installed on Windows, Linux, or Mac. *Chapter 9, Hands-on Kubernetes*, has practical exercises on installing and working with **kubectl**.

Note: The kubectl version used to manage a cluster can be either the same as the Kubernetes cluster version, one minor version older, or one minor version newer.

A **kubectl** command has the following format:

kubectl [command] [TYPE] [NAME] [flags]

For example:

kubectl describe pod frontend -n development

We will look at some frequently used **kubectl** commands later in the chapter in the *Creating resources* section.

Control plane

The control node(s) make up the control plane and runs the following components:

- API server
- Scheduler
- Controllers
- etcd data store

The *etcd* data store can be deployed separately, and it is recommended for large-scale deployments. Control plane nodes can also function as worker nodes.

API server

In Kubernetes, you must use APIs for communication, resource creation, and cluster management, among other operations. To use APIs, they must be exposed. The API server is the component that exposes the Kubernetes HTTP API and serves as the frontend of a Kubernetes cluster. Every communication with the Kubernetes cluster goes through the API server. This component enables clients such as **kubectl** and **kubeadm**, end users, and external components to communicate with the cluster. Different parts of the Kubernetes cluster also use the API server to communicate with one another. Using the API one can query the state of objects and create/modify objects.

The API server process is called **kubeapi-server**.

Note: API server or kubeapi-server is the only component that talks directly to etcd.

For example, when you run a **kubectl** command, it is received by the API server, checked for authentication and authorization, and on successful validation, stored in **etcd**.

You can also interact directly with the Kubernetes API using RESTful calls. This is particularly helpful for developers writing applications.

Kubernetes provides official client libraries for most popular programming languages.

Scheduler

The **kube-scheduler** is a crucial component within the Kubernetes cluster and is responsible for making intelligent decisions regarding the placement of new Pods onto nodes. There are certain key concepts concerning the scheduler, such as node affinity, anti-affinity, and taints and tolerations that ensure efficient placement of Pods. They are explained as follows:

- Node affinity allows administrators to influence the Pod placement on a node.
- Anti-affinity, on the other hand, prevents Pods from co-locating on the same node.
- Taints and tolerations provide a mechanism for nodes to repel or accept Pods.

We will cover affinity, anti-affinity, taints and tolerations in slightly more detail in the *Scheduling in Kubernetes* section of this chapter.

One can also customize the scheduling behavior through plugins. The process is identified by **kube-scheduler**.

Controllers

Controllers are control loops that watch the current state of the cluster and constantly make changes to try to achieve the desired state. An analogy for control loop in Kubernetes is a thermostat in your room that tries to bring the room temperature to your desired temperature.

Controllers interact with the API server to manage state. The process is identified by *kube-controller-manager*. Although this is a single process, multiple controllers at play in the background. The most common of these controllers is the *deployment controller*, which tracks and manages Pod objects. Some other examples of controllers in Kubernetes are:

- **Job controller:** A job controller ensures that enough Pods are running to complete a task.
- **StatefulSet controller:** This controller manages the desired state of Pods that run stateful applications such as databases.
- **DaemonSet controller:** A DaemonSet controller ensures that each node has a copy of the Pod.

The controllers discussed above are called **built-in controllers** and are part of kube-controller-manager. Controllers running outside of it that provide extended functionalities can also be created.

etcd store

etcd is a persistent key-value based data store that acts as a single source of truth for all the cluster data. It provides the necessary consistency and high availability for proper functioning of the cluster.

On large scale deployments, the etcd cluster is installed on a dedicated set of nodes. It is also recommended to have a backup strategy for the etcd store.

Cloud controller manager

It is common to run Kubernetes on public cloud providers, introducing scenarios where Kubernetes must communicate with cloud provider APIs. The CCM is a control plane component that helps you link a Kubernetes cluster with the provider it is running on.

For example, when you create a *LoadBalancer* service type, Kubernetes needs to talk to the cloud provider API to create the load balancer and the external IP resources. This information must then be reflected in the cluster. This is done by one of the controllers inside the CCM, which is called the **service controller**.

There are two more controllers in CCM, Node controller and Route controller.

Worker node

The worker node(s) are where the Pods run. Each worker node runs at least the following components:

- Kubelet
- Kube-proxy
- Container runtime, such as containerd

kubelet

Kubelet is an agent that runs on every node and is an important component facilitating the communication between the control plane and the node. It manages the node and the deployment of Pods on those nodes, among a few other responsibilities, such as:

- Registering the node
- Monitoring, managing, and other operations on the node, such as networking
- Pod execution
- Coordinating with the container runtime to run and manage the state of containers.

In the workflow, the `kubeapi-server` provides the `kubelet` with a `PodSpec`, and the `kubelet` ensures that the containers described in the `PodSpec` are running and healthy. It interacts with the container runtime, i.e., `containerd`, to start and stop containers.

kube-proxy

`kube-proxy` is a network proxy running on every node that is responsible for forwarding data to the correct endpoints for *services*. It does this by maintaining a set of network rules that allow communication to the Pods, originating from inside or outside the cluster.

Container runtime

A container runtime is a must in worker nodes. When a Pod spec is sent to kubelet for processing, it will communicate with the container runtime for container creation as well as other operations. The **Container Runtime Interface (CRI)** plays an important role here as it makes the runtime layer of Kubernetes pluggable. This enables you to use any CRI compatible runtime such as containerd, CRI-O, etc.

Note: The direct integration of Docker as a container runtime has been deprecated since Kubernetes v1.24.

The container runtime in a worker node manages the entire lifecycle of the container, which involves working with the kernel to create an isolated environment for the container, for example, creating the network namespace (not to be confused with the Kubernetes namespace).

Kubernetes resources

In this section, we will begin by defining a few terminologies that will be used throughout this chapter and the rest of the course to follow. We will then learn about some of the core Kubernetes resources before moving on to the architecture and other details. This will help us understand the architectural concepts better.

A **resource** in Kubernetes is a single instance of a resource type. It is also called an **object**. The words resource and object are often used interchangeably. For example, a *Pod* is a resource or object of the resource type *Pods*.

Note: While there is a more complex answer to this that addresses how Kubernetes APIs work and how they relate to resources and objects, it is not required for this scope. We will cover this later in this chapter under Kubernetes API.

Once your cluster is up, you can run the **kubectl api-resources** command to view a list of all available API resources.

Namespaces

Namespaces are a type of resource in Kubernetes that allows you to divide the cluster into isolated groups. They can have quotas and a separate set of users. For example, diving a cluster based on the application environment or department. Namespaces can also be used when you want to have the same name for two resources, for example, the same name for two deployments. Namespaces are also referred to as **virtual clusters**.

The initial namespaces that are created by default are:

- **default**: This is where resources are created by default.
- **kube-node-lease**: This namespace contains the node leases which are used by *kubelet* to send heartbeats to the control plane.
- **kube-public**: A namespace used for cluster resources that must be public throughout the entire cluster. All clients, including unauthenticated ones, can read it.
- **kube-system**: All Kubernetes objects are created in this namespace.

The following conditions apply to resources in namespaces:

- Within a namespace, the names of resources must be unique
- Namespaces apply to resources that are namespace scoped. If no namespace is provided for such resources, they are created in the *default* namespace.

By default, if no namespace is specified, a resource is created in the default namespace. You can create namespaces as needed and additionally set a namespace of your choice as the default namespace.

Pods

Pods are one of the most fundamental resources used to deploy/run containers that contain the application workload. They are the smallest unit of containerized applications that can be managed by Kubernetes.

A Pod wraps one or more containers into this unit, is atomic in nature, and hence scheduled as a unit, meaning if it contains multiple containers, they are scheduled together. Each Pod is isolated from other Pods by default. Pods share the network namespace, volumes, cgroups, and Linux namespaces. Containers within a Pod share the Pod's IP address and can communicate with each other by default using standard inter-process communication methods.

In Kubernetes, Pods are the basic building blocks of a microservices application.

A Pod can be one of the following phases:

- Pending
- Running
- Succeeded
- Failed
- Unknown

Pods cannot be restarted; only the containers within the Pod can. This is the work of the kubelet. By design, Pods are ephemeral. They are created, deleted, and recreated as needed, and hence are not expected to be durable and reliable.

The following is an example of a manifest file in YAML used to declare the configuration of a Pod:

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: kcnaprep  
spec:
```

```
containers:
- name: kcnaprep
  image: USERNAME/kcnaprep:1.0
  ports:
  - containerPort: 80
```

Types of containers in Pods

Here is a quick overview of the types of containers you may see in a Pod:

- **Regular containers:** These are containers that run the app or the code. They are also called **app containers**.
- **Init containers:** These are special containers that run before the primary app container to prepare, such as to run scripts, copy seeding data, run a utility, etc.
- **Sidecar containers:** These types of containers always run along with other containers in a Pod providing additional functionality. A popular example is sidecar containers for logging and monitoring.
- **Ephemeral containers:** These types of containers are temporarily run inside a Pod for purposes such as troubleshooting or debugging.

Deployments

A deployment in Kubernetes is a resource that provides declarative updates for Pods and ReplicaSets. You declare the desired state of an application running in Pods using a deployment configuration file, often referred to as a YAML file. A controller that manages deployments, called the **deployment controller**, will ensure that the desired state is maintained.

Deployments provide capabilities such as:

- Maintaining a specified number of replicas of a Pod using ReplicaSet
- Recreating Pods or self-heal
- Scaling the number of Pods
- Rollout and rollback updates to Pods or the deployment itself

Following is an example of a desired state declared using a *Deployment* kind or object:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
```

```

- name: frontend

  image: USERNAME/frontend:1.0.0

  ports:
    - containerPort: 8080

```

The following figure illustrates a deployment with two replicas:

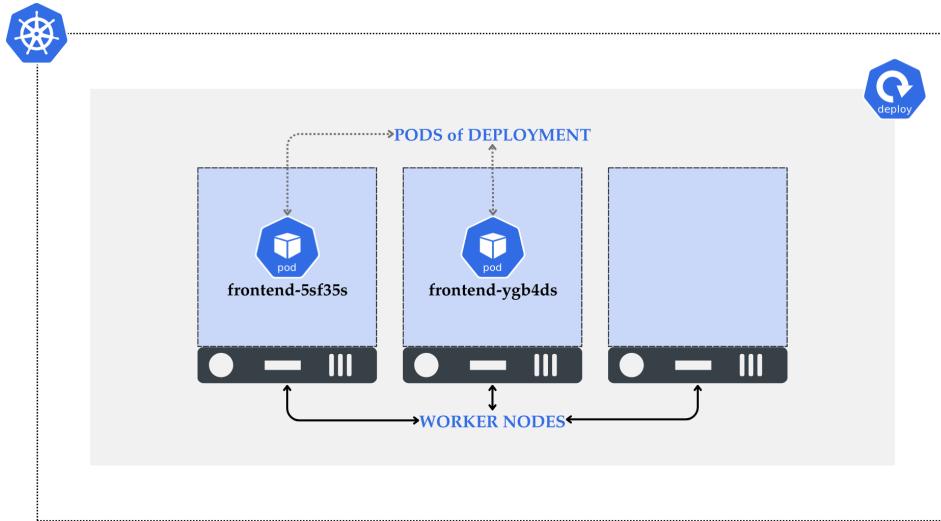


Figure 4.2: A deployment with two replicas

StatefulSets

Deployments are used to manage stateless applications. For stateful applications, Kubernetes provides an object called **StatefulSets**. Here, statefulness means the state of the Pod network, storage, etc. A database replica, such as MySQL replica, is an example of a stateful workload that requires stable network, storage, and ordering during scheduling and rescheduling.

Both deployments and StatefulSets manage Pods based on identical container spec.

StatefulSets ensure the following:

- Guaranteed ordering by sequentially creating, updating, and deleting Pods based on their **ordinal index**, which avoids split-

brain scenarios.

- Pod uniqueness by assigning each Pod a stable, unique identity in a predictable name format, for example **mysql-pod-0**, **mysql-pod-1** and so on.
- Mapping to the correct volume.

StatefulSets are ideal for use cases that require:

- Stable and unique network identifiers.
- Stable and persistent storage.
- Ordered graceful deployment and scaling.
- Ordered automated rolling updates.

An example StatefulSet YAML file for a MySQL database is given as follows:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: auth-db
spec:
  replicas: 2
  serviceName: auth-db
  selector:
    matchLabels:
      app: auth-db
  template:
    metadata:
```

```
labels:
  app: auth-db

spec:
  containers:
    - name: mysql
      image: mysql:5.7
      ports:
        - name: tcp
          protocol: TCP
          containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: strongpassword
      volumeMounts:
        - name: auth-db-pvc
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: auth-db-pvc
      spec:
        storageClassName: "local-path"
        accessModes:
          - ReadWriteOnce
      resources:
```

requests:

storage: 1Gi

The following figure shows a StatefulSet without persistent storage (it is uncommon but very valid) with unique Pod names:

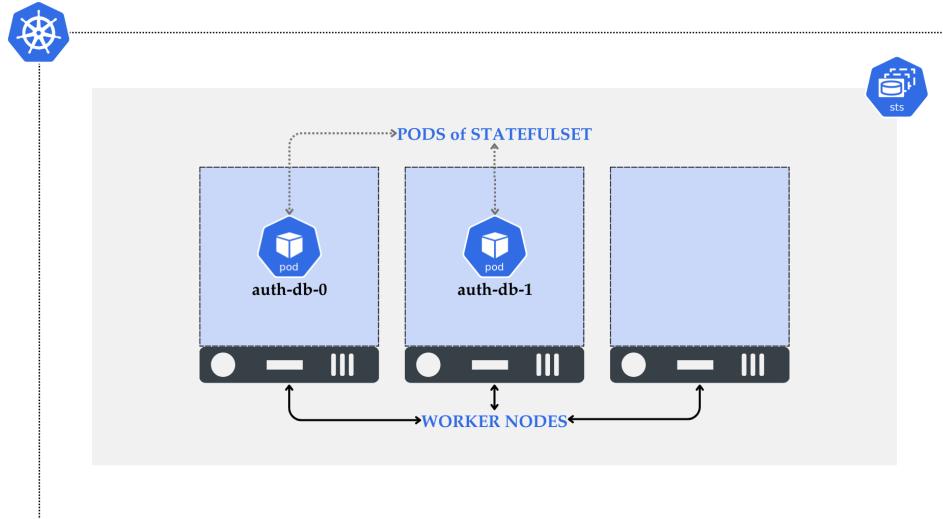


Figure 4.3: A StatefulSet with two replicas without persistent storage or PVC

DaemonSets

A DaemonSet is an API object that runs a copy of the Pod in all applicable nodes. The controller that manages it is called a **DaemonSet controller**.

Following is a visual representation of a DaemonSet:

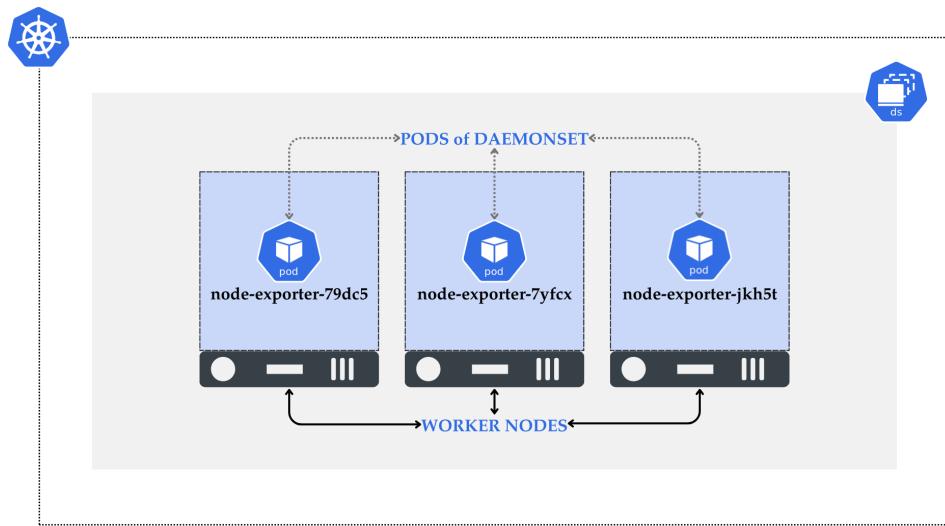


Figure 4.4: A DaemonSet

Consider the following example YAML of a DaemonSet:

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: prometheus
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
    metadata:
      labels:
        app: node-exporter

```

```
spec:  
  containers:  
    - name: node-exporter  
      image: prom/node-exporter:v1.8.0  
      ports:  
        - containerPort: 9100
```

Jobs and CronJobs

Apart from running applications, systems may also need to run tasks such as backups, database migrations, batch processing, cleanup files or disk space, garbage collection, etc. These tasks could be a one time or a periodically scheduled task. To run such tasks, Kubernetes provides two resource types, **Jobs** and **CronJobs**.

The following YAML file is an example of a *Job*:

```
---  
apiVersion: batch/v1  
kind: Job  
metadata:  
  name:  
spec:  
  template:  
    spec:  
      containers:  
        - name: pi  
          image: perl:5.34.0
```

```
        command: ["perl", "-Mbignum=bpi", "-wle", "p
rint bpi(2000)"]
    restartPolicy: Never
  backoffLimit: 4
```

Here is an example of a simple *CronJob* manifest that runs every day at 3 AM to back the **kcnaprep** database:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: kcnaprep-db-backup
spec:
  schedule: "0 3 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: kcnaprep-db-backup
              image: USERNAME/kcnaprep-db-backup:v1
              env:
                [portion of code removed]
              imagePullPolicy: IfNotPresent
            restartPolicy: OnFailure
```

Resource requests and limits

While discussing Pods, let us understand resource **requests** and **limits**, which are explained as follows:

- Requests are the guaranteed resources that a container gets.
- Limits are the maximum resources that a container can get.

The following YAML shows an example of a Pod containing multiple containers with resource requests and limits specified:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: kcnaprep
spec:
  containers:
    - name: kcnaprep-ui
      image: USERNAME/kcnaprep-ui:v1
      resources:
        requests:
          memory: "1Gi"
          cpu: "500m"
        limits:
          memory: "2Gi"
          cpu: "1"
    - name: kcnaprep-app
      image: USERNAME/kcnaprep-app:v1
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
        limits:
```

```
memory: "1Gi"  
cpu: "500m"
```

It is a recommended best practice to set requests and limits for containers. These also play an important role when scheduling Pods onto nodes.

Storage and networking resources

Kubernetes also provides APIs for storage and networking resources such as *Services*, *PersistentVolumes*, and *PersistentVolumeClaims*. We will cover them in [Chapter 5, Container Orchestration with Kubernetes](#).

Creating resources

Resources can be created in an imperative or declarative manner. A declarative manner is usually preferred and requires a manifest.

Manifests

To deploy a resource or API object in a Kubernetes cluster, you describe its desired state in a configuration file, otherwise called a **Manifest** file. These can be written in either YAML or JSON.

A single manifest file may contain the configuration of multiple resources. They are separated using '---' (three hyphens).

The following is part of an example of a manifest file for Pod, which we also saw earlier in the chapter:

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: kcnaprep  
spec:  
  containers:
```

```
- name: kcnaprep  
  image: USERNAME/kcnaprep:1.0
```

The following fields are key components of a manifest and must exist in a YAML file:

- **apiVersion**: API version of the resource or object being created.
- **kind**: Type of resource such as *Pod*, *Service*, *Deployment*, etc.
- **metadata**: Specify a name, label, annotation, etc.
- **spec**: This important component is where you define the desired state, otherwise called the **object specification**.

Manifest files are deployed using the **kubectl** tool, generally using the **kubectl apply** command, which is a declarative way of creating resources. They can also be deployed using the **kubectl create** command, which is an imperative approach.

kubectl examples

The kubectl CLI discussed earlier in this chapter can be used for both imperative and declarative commands. We will see some examples below but practice them in much detail in [Chapter 9, Hands-on Kubernetes](#).

You can create or manage common Kubernetes objects using verb-driven commands such as **kubectl run**, **kubectl create**, **kubectl expose**, etc. Here are some examples of **kubectl** imperative commands:

Create and run a simple Pod with a specific image:

```
kubectl run nginx --image=nginx
```

To create a deployment:

```
kubectl create deployment nginx --image=nginx
```

To expose a deployment:

```
kubectl expose deployment nginx --type=ClusterIP --port=80 --name=nginx-svc
```

API objects in Kubernetes are represented in JSON format, and when working with Kubernetes resources or objects, you may want to extract specific data from this JSON. Kubectl supports **JSONPath template**, allowing JSONPath expressions to filter and display customized outputs.

Consider the following example of using JSONPath query language to filter:

```
kubectl get pods -o=jsonpath='{.items[0].metadata.name}'
```

Note: JSONPath regular expressions are not supported but you can use a tool like jq for this.

Refer to the following reference documentation for a list of commands:

<https://kubernetes.io/docs/reference/kubectl/>

Scheduling in Kubernetes

Scheduling in Kubernetes involves matching a suitable node that satisfies the conditions for running a Pod. This is done by the **kube-scheduler** service that runs in the control plane.

Without getting into a lot of details, as they are not necessary for this exam, the node selection is based on a two-step **filtering and scoring** approach. In the filtering step, the scheduler identifies all suitable nodes, and in the scoring step, it assigns a score to these nodes based on active scoring rules, ranks them, and selects one with the highest ranking.

Basic scheduling

At a fundamental level, basic scheduling considers the resource *requests* parameter of the containers to match a suitable schedulable node that can run the Pod.

Note: Resource requests and limits were covered earlier in this chapter under Kubernetes resources.

Advanced scheduling

To control the assignment of a Pod to a particular node using any of the following methods, you must start by labeling the node(s).

A node can be labeled using the **kubectl label** command. Following are some examples of working with labels:

- To label a node:

```
kubectl label nodes worker0 role=database
```

Note that here `role=database` is a random key=value pair of your choice.

- To view labels:

```
kubectl get nodes --show-labels
```

nodeSelector

Using the *nodeSelector* field in the Pod spec, one can constrain Pod creation to a specific node. It is simple and works using the *label* and *selector* concept. It requires the nodes to be labeled.

Following is an example of a Pod with a *nodeSelector* property that will select a node labeled with **role=database**, which is **worker0** in this case:

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
  name: mysql
```

```
spec:
```

```
  containers:
```

```
- name: mysql
  image: mysql
  imagePullPolicy: IfNotPresent
nodeSelector:
  role: database
```

Affinity and anti-affinity

Affinity allows better control over the node selection logic, such as placing Pods on particular nodes or nodes with particular Pods.

Anti-affinity ensures certain Pods are not placed on certain nodes or placed along with other Pods.

nodeName

The *nodeName* field can be used in a Pod spec to specify a particular node where it should be created. It is a static way of creating Pods and is ideally not recommended for typical use cases.

Taints and tolerations

Taints are properties applied to nodes that prohibit Pods from being scheduled on them unless they have a tolerance for it. Tolerations are applied to Pods. This is also a matching process where taints are matched to tolerations.

Taints are applied to a node using the **kubectl taint** command. Following is an example of tainting a node:

```
kubectl taint node worker-0 workload=db:NoSchedule
```

Kubernetes has a few built-in taints used in certain conditions, such as when a new node is added and it is in *NotReady* status or where control plane nodes are *unschedulable*.

Here is a final summary of node affinity, anti-affinity, and taints and tolerations:

- Node affinity allows administrators to influence the Pod placement by specifying node affinity rules based on node labels.
- Anti-affinity, on the other hand, helps prevent certain Pods from co-locating on the same node, enhancing fault tolerance.
- Taints and tolerations provide a mechanism for nodes to repel or accept Pods based on specified attributes.

Kubernetes API

The Kubernetes API allows you to interact with the cluster programmatically, especially with objects such as Pods, Deployments, Services, etc. These are also called **API objects**, and they represent the current state of a system.

Let us cover some definitions before going further:

- **Objects**: These are entities that represent the state of a system. For example, Pods, Deployments, ReplicaSet, etc.
- **Resources**: An instance of the object is a resource.
- **Resource type**: It is the API endpoint used to query an object. All resource types are versioned, and some, mostly newer resource types, are grouped. Examples of resource types are Nodes, Pods, etc. This can be viewed using the command **kubectl api-resources**.
- **Kind**: This defines the schema of a resource type.

Kubernetes APIs are grouped for convenience and extensibility. The legacy group is at **/api/v1**, and there is only one version of it. All the newer named groups are at **/apis/** path. For example, **/apis/apps/v1** is the apps API group for Deployments, ReplicaSets, etc.

Let us consider this API URL for example:

<http://localhost:8080/api/v1/namespaces/default/pods/kcnaprep>

Here:

- **/api** is where the Kubernetes API legacy endpoint lives
- **/v1** is the version of the API
- **/namespaces** are used to specify the namespace where the object is
- **/default** is the name of the namespace
- **/pods** is the resource type, object of kind Pod
- **/kcnaprep** is the name of the object, i.e., Pod

You can interact with the API using kubectl, client libraries, or direct HTTP requests. You can also write custom clients or applications that can interact with the cluster through the API. Kubernetes provides an official Go client, among other programming languages, for interacting with APIs through code.

The API is versioned, with each version introducing new features, improvements, and sometimes deprecating older functionality. It follows RESTful principles with resources represented as objects in JSON or YAML format and accessible via URLs. Kubernetes API versions also indicate their stability and support level. There are three levels:

- **Alpha:** Versions that contain the word alpha are experimental versions with new features or unstable APIs. They may contain bugs, be incompatible, or introduce breaking changes. These are only for testing and, in most cases, do not have support. An example of such a version could be v1alpha1.
- **Beta:** Beta versions contain the word beta in the version. These versions have features that are well-tested and safe to enable but in non-production environments. Beta versions have a nine-month life span from introduction to deprecation and from deprecation to removal. You must also expect incompatible or breaking changes from one beta version to another or from the beta version to the stable. v1beta1 is an example of such a version.

- **Stable:** Stable versions are denoted as v1, v2, etc. These versions remain for all future Kubernetes releases.

When you run a **kubectl** command, it is converted into an API request and presented to the API server, which exposes these APIs.

To view a list of all available resource types (API) in a cluster, use the following command:

kubectl api-resources

The following figure shows the output of the command:

NAME	SHORTNAMES	APIVERSION	NAMESPACEDEP	KIND
bindings		v1	true	Binding
componentstatuses	cs	v1	false	ComponentStatus
configmaps	cm	v1	true	ConfigMap
endpoints	ep	v1	true	Endpoints
events	ev	v1	true	Event
limitranges	limits	v1	true	LimitRange
namespaces	ns	v1	false	Namespace
nodes	no	v1	false	Node
persistentvolumeclaims	pvc	v1	true	PersistentVolumeClaim
persistentvolumes	pv	v1	false	PersistentVolume
pods	po	v1	true	Pod
podtemplates		v1	true	PodTemplate
replicationcontrollers	rc	v1	true	ReplicationController
resourcequotas	quota	v1	true	ResourceQuota
secrets		v1	true	Secret
serviceaccounts	sa	v1	true	ServiceAccount
services	svc	v1	true	Service
mutatingwebhookconfigurations		admissionregistration.k8s.io/v1	false	MutatingWebhookConfiguration
validatingwebhookconfigurations		admissionregistration.k8s.io/v1	false	ValidatingWebhookConfiguration
customresourcedefinitions	crd, crds	apiextensions.k8s.io/v1	false	CustomResourceDefinition
apiservices		apiregistration.k8s.io/v1	false	APIService
controllerrevisions		apps/v1	true	ControllerRevision
daemonsets	ds	apps/v1	true	DaemonSet
deployments	deploy	apps/v1	true	Deployment
replicasetss	rs	apps/v1	true	ReplicaSet
statefulsets	sts	apps/v1	true	StatefulSet
tokenreviews		authentication.k8s.io/v1	false	TokenReview
localsubjectaccessreviews		authorization.k8s.io/v1	true	LocalSubjectAccessReview
selfsubjectaccessreviews		authorization.k8s.io/v1	false	SelfSubjectAccessReview
selfsubjectrulesreviews		authorization.k8s.io/v1	false	SelfSubjectRulesReview
subjectaccesreviews		authorization.k8s.io/v1	false	SubjectAccessReview
horizontalpodautoscalers	hpa	autoscaling/v2	true	HorizontalPodAutoscaler
cronjobs	cj	batch/v1	true	CronJob
jobs		batch/v1	true	Job
certificatesigningrequests	csr	certificates.k8s.io/v1	false	CertificateSigningRequest
leases		coordination.k8s.io/v1	true	Lease

Figure 4.5: Output of kubectl api-resources

The **kubectl api-resources** command lists all available resources you can deploy in a Kubernetes cluster. You can also create your own custom resource through a **CustomResourceDefinition** (CRD).

Conclusion

A Kubernetes cluster typically consist of control plane nodes and worker nodes. The control plane nodes run the Kubernetes system components

and your workloads or applications run on worker nodes. You interact with the cluster using REST APIs or kubectl.

To logically partition a cluster, you can use namespaces. *Pods* are the basic unit of deployment in Kubernetes but in a real-world scenario higher construct such as *Deployments*, *DaemonSets* and *StatefulSets* are used. Use a *StatefulSet* to run stateful workloads such as databases. These resources are created using a manifest file written in YAML which is the declarative way. Resources can also be created using ad-hoc **kubectl** commands which would be the imperative approach and is suitable for quick tests, troubleshooting etc.

Resource scheduling is done by the *Scheduler* in control plane node which assigns Pods to nodes based on resource requirements (like CPU and memory) defined using *limits* and *requests*, constraints (node labels, *affinity* rules), and current cluster resource availability, ensuring optimal workload distribution. Additionally, you can use *taints* and *tolerations* to further control scheduling.

In the next chapter, we will learn about networking, storage, and security options in Kubernetes, including an introduction to service mesh.

Multiple choice questions

1. Fill in the blanks.

A _____ wraps one or more containers into a unit of management and is used to deploy applications.

- a. Deployment
- b. DeploymentSet
- c. ReplicaSet
- d. Pod

2. Which of the following are popular tools to set up a production Kubernetes cluster? (Select all that apply)

- a. kubeadm

- b. kops
 - c. kubectl
 - d. kubelet
 - e. Both a and b
3. Which of the following components runs on every node in a Kubernetes cluster and is responsible to ensure that containers are running and healthy?
- a. kubectl
 - b. kubelet
 - c. kube-proxy
 - d. supervisord
4. Which of the following components would you find in a control plane? (Select all that apply)
- a. kube-apiserver
 - b. kube-scheduler
 - c. kube-controller-manager
 - d. cloud-controller-manager
 - e. All of the above
5. Which of the following components running on every node is responsible for starting and stopping the containers?
- a. kubelet
 - b. containerd
 - c. kubectl
 - d. Pod
6. Choose the supported mechanisms to communicate with the Kubernetes API.
- a. kubectl

- b. HTTP/REST
 - c. Client libraries
 - d. All of the above
7. What is the function of kubectl?
- a. It is an agent that runs on every worker node enabling users to communicate and manage the worker node.
 - b. It is a control plane component responsible for handling API requests.
 - c. It is a CLI that allows administrators to interact with the cluster.
 - d. It is an installation program used to install Kubernetes.
8. kube-scheduler is a Kubernetes cluster component that is responsible for scheduling Pods and runs on every worker node.
- a. True
 - b. False
9. Which API object ensures that a copy of the Pod runs on all applicable nodes?
- a. ReplicaSet
 - b. DaemonSet
 - c. StatefulSet
 - d. NodeSet
10. Which Kubernetes resource ensures the desired number of stateless Pods are always running?
- a. ReplicaSet
 - b. StatelessSet
 - c. StatefulSet

- d. Deployment
11. Which Kubernetes resource ensures the desired number of stateful Pods are always running?
- a. ReplicaSet
 - b. DaemonSet
 - c. StatefulSet
 - d. Deployment
12. Which of the following resource types would you choose for a workload that requires guaranteed ordering, Pod uniqueness and fixed volume mapping?
- a. ReplicaSet
 - b. Deployment
 - c. StatefulSet
 - d. DaemonSet
13. Which of the following query language would you use to extract specific data from kubectl command outputs?
- a. SQL
 - b. GraphQL
 - c. NoSQL
 - d. JSONPath
14. Select the endpoint to use when using Kubernetes API for objects part of legacy groups such as Pod.
- a. /api/v1
 - b. /apis/v1
 - c. /apis/apps/v1
 - d. /apps/v1

15. Which of the following scheduling approaches would you use to avoid placing two Pods together?
- Affinity
 - Anti-affinity
 - nodeSelector
 - nodeName

Answers

1	d.
2	e.
3	b.
4	e.
5	b.
6	d.
7	c.
8	a.
9	b.
10	d.
11	c.
12	c.
13	d.
14	a.
15	b.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Container Orchestration with Kubernetes

Introduction

This chapter covers many objectives of the *Container Orchestration* domain of the exam with a weightage of 22% and expects knowledge on the following areas:

- Container orchestration fundamentals
- Runtime
- Security
- Networking
- Service mesh
- Storage

Topics such as *container orchestration fundamentals* and *runtimes* in Kubernetes were already covered in [*Chapter 2, Understanding Containers and the Need for Container Orchestration*](#). This chapter will focus on the security, networking, and storage aspects of Kubernetes that are also

necessary for container orchestration. We will also learn about the fundamentals of service mesh.

Structure

This chapter covers the following topics:

- Security
- Networking
- Service mesh
- Storage

Objectives

The objective of this chapter is to cover the security, networking and storage aspects when orchestrating containers using Kubernetes. The chapter will provide readers with information on various objects, capabilities, and features of Kubernetes that can be used to address the above needs. We will learn about security components such as **role-based access control (RBAC)**, Network Policies, ConfigMaps, and Secrets; networking objects such as Service and its types, DNS, Ingress, etc.; and storage objects such as **Persistent Volume (PV)** and **Persistent Volume Claim (PVC)**, etc. The chapter also covers service mesh.

Security

Cloud-native principles such as distributed microservices and containers combined with dynamic scaling can introduce a sprawling attack surface. Traditional security measures built for monolithic applications cannot keep pace with the ever-shifting landscape of cloud-native deployments. This inherent complexity necessitates a security strategy woven into the fabric of cloud-native development and operations. Additionally, Kubernetes, which is the preferred container orchestration platform of choice for many cloud-native deployments, becomes a critical focal point for securing this dynamic environment. Securing Kubernetes involves

implementing robust security practices for both the cluster and the workloads running in it.

In this section of the chapter, we will look at general security best practices and Kubernetes objects that can be used for a secure cloud-native design.

The 4 Cs of Cloud Native Security

When planning to secure Kubernetes and the workloads running in it, a defense-in-depth strategy is recommended, as there are multiple layers that need to be secured. The 4 Cs of Cloud Native Security address the security needs of a cloud-native system while implementing a layered approach to security.

The four layers or the 4 Cs are:

- **Cloud:** This layer involves the security of the cloud infrastructure, or in general, any infrastructure used for Kubernetes deployment. For cloud deployments, the cloud provider is responsible for providing physical security, platform security, and security controls, but you are responsible for implementing these controls following provider-recommended practices. Examples of cloud infrastructure can be Microsoft Azure, AWS, GCP, etc.
- **Cluster:** This layer involves securing Kubernetes and its workloads. Implementing encrypted communication, using TLS, securing access to *kube-apiserver*, and using RBAC, ConfigMaps, Secrets, and NetworkPolicies where applicable are some of the steps. We will discuss some of these in this chapter later.
- **Container:** At this layer, you secure the container images, scan them for vulnerabilities, check the use of untrusted and insecure base images, check for containers with privilege escalations, etc.
- **Code:** At the code layer, you must focus on the security of the code of your application. Areas like secure communication using TLS, encryption, code analysis, cross-site scripting, use of untrusted or unknown libraries and dependencies, etc., fall here.

The following figure visualizes the four layers:

4 Cs of Cloud Native Security

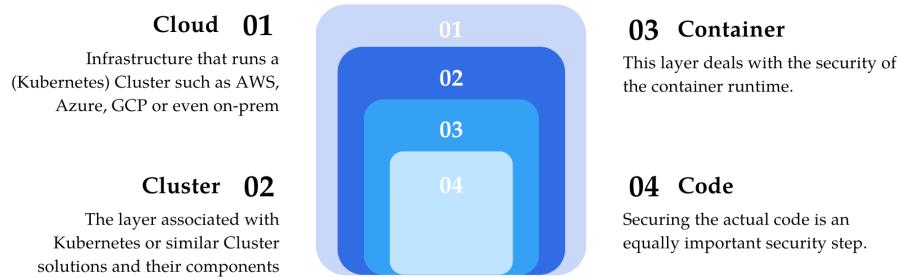


Figure 5.1: Illustration of the 4 Cs of Cloud Native Security

Securing Kubernetes falls under the cluster security layer of the 4 Cs of Cloud Native Security. It is often separated as security of the components that make up the cluster and securing the workloads running in the cluster. To implement cluster and workload security, Kubernetes provides various capabilities and objects that can be used.

We will discuss some of these objects, such as those that are part of the role-based access control mechanism like Roles, cluster roles, role binding, and cluster role binding, along with objects like ConfigMaps and Secrets in the sections that follow.

Cluster security

Keeping in mind the scope of the exam, here is a list of some of the important steps to secure a Kubernetes cluster:

1. Control access to Kubernetes API using TLS combined with proper authentication and authorization. Additionally, you can use admission control to modify or reject requests.
2. In addition to using encryption for data in transit, you can also implement encryption at rest for data stored in *etcd*. To encrypt

etcd, you can generate or use an encryption key generated using a tool. Alternatively, you can also use a **Key Management Service (KMS)** provider like AWS KMS, Google Cloud KMS, Azure Key Vault, HashiCorp Vault, etc.

3. Create and use audit policies to track activities by users, applications and the cluster itself.
4. When implementing RBAC, follow the principles of least privilege, which means assigning only the required permissions for a role.
5. Keep the cluster up to date with security fixes.

Role-based access control

RBAC is a general term used outside of Kubernetes, too. It fundamentally means controlling access to resources based on the role of an individual, app, or device. It follows the principle of least privilege access. The API group for this is **rbac.authorization.k8s.io**.

There are four objects within the Kubernetes RBAC API group that help control access to resources:

- **Role**: A **Role** is a list of rules with a set of permissions and is limited to a namespace. Rules are always added, you cannot create deny rules.
- **ClusterRole**: A **ClusterRole** is also a role but not limited to any namespace. It applies to the entire cluster. So, they are used to define permissions to cluster-scoped resources, non-resource endpoints or for resources across all namespaces.
- **RoleBinding**: A **RoleBinding** is used to grant a role to a list of subjects such as users, groups or service accounts. It grants permissions within a namespace.
- **ClusterRoleBinding**: A **ClusterRoleBinding** does the same thing as **RoleBinding** but cluster wide. You will use a **ClusterRoleBinding** object to bind a **ClusterRole** across all namespaces.

Consider reading the RBAC good practices at:

<https://kubernetes.io/docs/concepts/security/rbac-good-practices/>

Service accounts

A *ServiceAccount* is a non-human account or identity that can be granted permissions using the RBAC method discussed earlier, which can then be used by components such as Pods, system components, or other entities for authentication and authorization. It helps in implementing identity-based security policies for non-user accounts that follow the principles of least privilege. You can define granular access control to such entities, reduce the attack surface, and improve overall cluster security.

Service accounts is ideal for use cases such as:

- Ensuring least privilege access to Pods
- Pods needing access to sensitive data stored as a Secret or configuration data stored in a ConfigMap. (Secrets and ConfigMaps are covered below)
- Pods communicating with Kubernetes service, custom resources or other services with a cluster and requiring authorization.
- Pods requiring elevated permissions for a task that can be provided through a service account (the Pod impersonates the service account to gain these privileges).

Network policies

Kubernetes network policies are rules that can be specified to control network traffic between Pods and other network endpoints in a cluster.

You can use network policies to control:

- Which pods can talk to which Pods
- Network traffic to and from one or more namespaces
- IP based traffic control

You need a networking plugin to implement network policies.

A network policy is defined as a YAML file of kind NetworkPolicy and includes three important specs: *podSelector*, *policyTypes*, and rules (ingress and/or egress). The policy types are ingress and egress. A network policy may have one or both policy types.

The following is an example of a *NetworkPolicy* in YAML:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
  ports:
    - protocol: TCP
```

```
port: 3306
```

ConfigMaps

Kubernetes ConfigMaps provides a way to store non-confidential application configuration data like configuration files, environment variables, or even API keys that are safe to share. This simplifies container deployments by separating configuration from container images. An example of this is a web application requiring the configuration for a database connection. Instead of hardcoding it into the image, you can store it in a ConfigMap. This allows for easier updates to the configuration without rebuilding the image, keeping your deployments agile and adaptable.

The following YAML shows how to create a ConfigMap:

```
---
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: db-connection
data:
  database_engine: mysql
  database_url: mysql://localhost:3306
```

Secrets

A Kubernetes Secret provides a secure way to store sensitive information like passwords, API keys, or tokens within your containerized applications. This is crucial for isolating sensitive data and preventing accidental exposure. Imagine a database connection string containing a password. Storing it directly in a container image would be risky. Instead, you can store it securely in a Secret and inject it into your

application at runtime using environment variables. This keeps sensitive data separate from your application code and ensures it is not inadvertently leaked.

In contrast to *Secrets*, *ConfigMaps* are designed for non-confidential configuration data like environment variables, configuration files, or even API keys that are safe to share. They offer a simpler way to manage application configurations without the extra security measures needed for Secrets.

Secrets can be one of the following built-in types:

Secret type	Description
Opaque	It is the default type of Secret mostly used for random user-defined data.
kubernetes.io/basic-auth	This type is used to store basic authentication data that mostly use a username and a password.
kubernetes.io/ssh-auth	It is used to store SSH keys.
kubernetes.io/tls	SSL/TLS certificates are stored using this type of Secret. It typically includes a key and a certificate data.
kubernetes.io/dockercfg	This Secret type is used to store credentials for a container image registry. The file stored is the <code>~/.dockercfg</code> in serialized format.
kubernetes.io/dockerconfigjson	This type does the same thing as the previous but used the new format of <code>~/.docker/config.json</code>

	and stores the data in serialized JSON.
bootstrap.kubernetes.io/token	Bootstrap data used to bootstrap nodes are stored using this Secret type.
kubernetes.io/service-account-token	It is used to store Service Accounts.

Table 5.1: Types of built-in Kubernetes Secrets

The following YAML files describe a simple secret of type *Opaque*:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: ZGJhZG1pbg==
  password: QVN0cm9uZ1Bhc3N3b3Jk
```

Secrets are stored in etcd and are unencrypted by default. Anyone who can access the etcd data store or the Kubernetes APIs can view the secret or modify it. To secure the Secrets themselves, you can look at the following recommendations:

- Use encryption at rest.
- Use RBAC to implement access controls such as disallowing reading of Secrets, creating and modification of Secrets, etc.

Consider reading the good practices for secrets at:

<https://kubernetes.io/docs/concepts/security/secrets-good-practices/>

Networking

Let us start with some networking basics in Kubernetes. A Pod gets a unique private IP address, which all containers within the Pod share. If a Pod contains one or more containers, they work on different ports and can talk to each other via *localhost*.

The out of the box networking model in Kubernetes is simple and follows a flat networking model. The following networking rules apply:

- All Pods across all nodes can communicate with one another without Network Address Translation (NAT).
- Kubelet can communicate with all Pods on that particular node.

We will now look at networking objects available in Kubernetes and their use.

Service

Pods are exposed through a service to access them from within the cluster, outside the cluster, or from the internet. Pods of the same ReplicaSet or Deployment are tied together by a single service acting as a load balancer. A service provides a static and consistent way to access the Pods, even if they are spread across multiple nodes. Services are mapped to Pods using labels.

Services are used to expose resources such as *Deployments* and *StatefulSets*.

Kubernetes provides the following service types:

- ClusterIP
- NodePort
- LoadBalancer

ClusterIP

The ClusterIP service type is the default service type.

The following figure is an illustration of how the ClusterIP works:

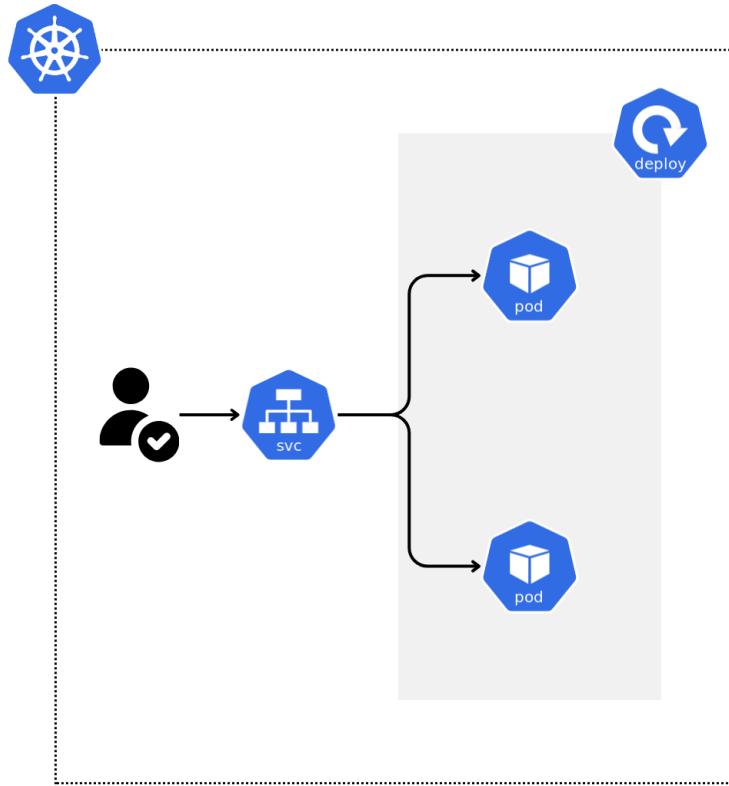


Figure 5.2: Traffic flow in ClusterIP service type

Following is an example of a YAML file for ClusterIP service type:

```
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: my-service  
spec:  
  selector:  
    app.kubernetes.io/name: MyApp  
  ports:
```

```
- protocol: TCP  
  port: 80
```

targetPort: 9376

NodePort

A NodePort service type extends the ClusterIP service type by adding routing rules. This enables the service to be accessible from outside the cluster.

The actual node port value is optional and can be specified in the YAML or omitted for Kubernetes control plane to automatically assign a port from the default range of 30000-32767.

The following figure shows how a NodePort service type works:

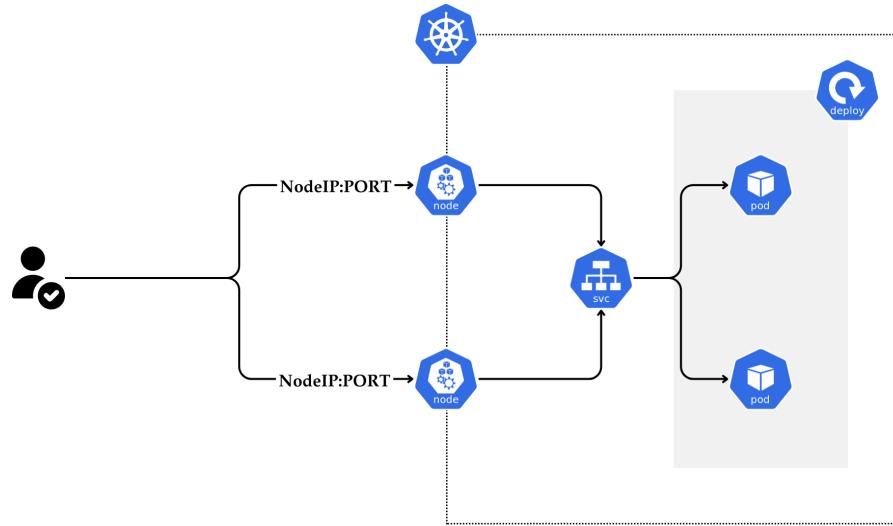


Figure 5.3: Traffic flow in NodePort service type

Consider the following example of a YAML file for NodePort service type:

```
---  
apiVersion: v1  
kind: Service  
metadata:
```

```
  name: my-service
  spec:
    selector:
      app.kubernetes.io/name: MyApp
    ports:
      - protocol: TCP
        port: 80
        targetPort: 9376
    type: NodePort
```

Load balancer

This type of service is typically used with cloud providers to create an external load balancer for load balancing traffic to the backend Pods. The load balancer service type uses the NodePort service type, or in other words, it is an extension of it. The cloud-controller-manager is engaged in this scenario to communicate with the cloud provider to create the load balancer and then forward the traffic to the assigned node port.

The following figure shows the traffic flow when using a load balancer service:

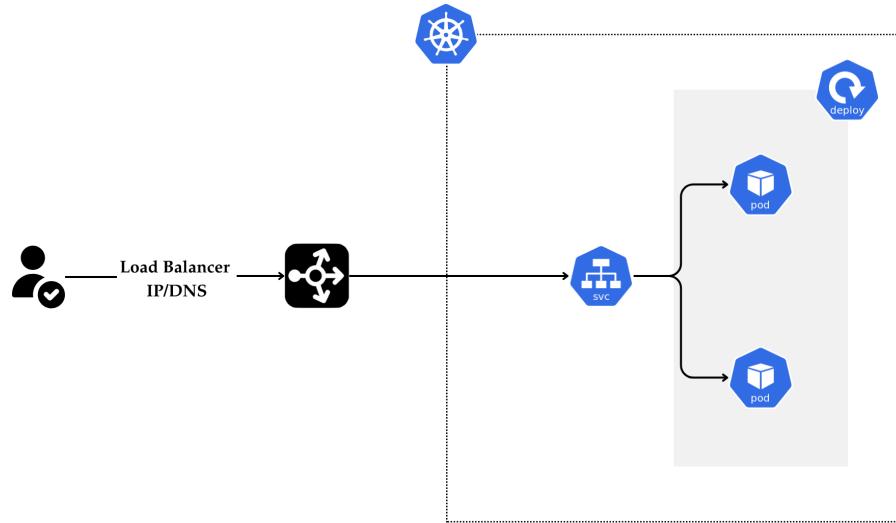


Figure 5.4: Traffic flow in load balancer service type

The following is an example of a YAML file for LoadBalancer service type:

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
type: LoadBalancer

```

Note: To expose applications for public access, you can use NodePort, LoadBalancer, or Ingress methods.

Headless service

In Kubernetes, a headless service is a unique service that operates differently from the standard service type. Unlike a regular service that has a cluster IP address for the Pods, it manages and routes the client request to a random Pod, a headless service does not have a cluster IP and does not route client requests. Instead, a headless service gives back a list of all the Pods behind it and their IP addresses, allowing the client to connect directly to a particular Pod using the Pod's IP address. *Kube-proxy* does not manage these services, hence there is no load balancing or proxying either.

This is implemented by categorically specifying *clusterIP* address as **None**. Consider the following example of a headless service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  clusterIP: None
  ports:
    - protocol: TCP
      port: 80
  targetPort: 9376
```

Headless services are useful in scenarios such as:

- *StatefulSets* where there is a need to connect back to the same Pod in case of disconnect since the state is maintained in that Pod. For example, when deploying databases like MySQL or messaging services such as RabbitMQ or Kafka.
- The need to connect to all Pods behind a service instead of one random Pod.
- When Pods behind a service must communicate with every other Pod behind that service. For example, in scenarios where clustering is implemented or for use cases where failover is required, like databases.
- Health check of all Pods behind a service.
- When you do not want to use the native service discovery mechanism and load balancing methods and want to implement a custom load balancing logic for traffic distribution or routing.

DNS

Kubernetes supports DNS-based service discovery and name resolution. DNS is used for communication between different components, such as Pods and Services, which are the only two objects that get DNS records.

CoreDNS is the default DNS service in Kubernetes and typically runs as a deployment with two replicas. While it is typically used for DNS requirements within the cluster, it can also leverage external DNS providers.

When a service is created for one or a set of Pods, a DNS name is also created for it. Behind the scenes, the CoreDNS service running in Kubernetes watches the API server for new services that are created and when they are, it creates a DNS entry mapping the IP of the Service to a DNS name of the format **<ServiceName>.<Namespace>.svc.<Domain>**. For example, **kcnaprep.default.svc.cluster.local**.

Note: The default domain name is **cluster.local**.

Ingress

Ingress in Kubernetes is an API object that provides external access through HTTP/HTTPS to services in a cluster. Traffic routing is done using rules specified when creating the ingress resource.

Depending on the ingress controller, an ingress may provide additional networking services such as SSL termination, load balancing, etc.

There are many prerequisite steps to be performed before creating an ingress resource. Going into the details of them is beyond the scope of this book. However, they typically include creating an ingress class, installing an Ingress Controller in the Kubernetes cluster, creating necessary service accounts, roles and cluster roles. It also requires configuring SSL/TLS certificates for the domain(s) using Secrets which can be done manually or through automated tools such as LetsEncrypt. There could be more steps depending on the ingress controller.

https://www.canva.com/design/DAF_XE9na5o/_aixPoSCTynD9qYk5s_8jw/edit

The following figure shows an example of traffic routing using ingress:

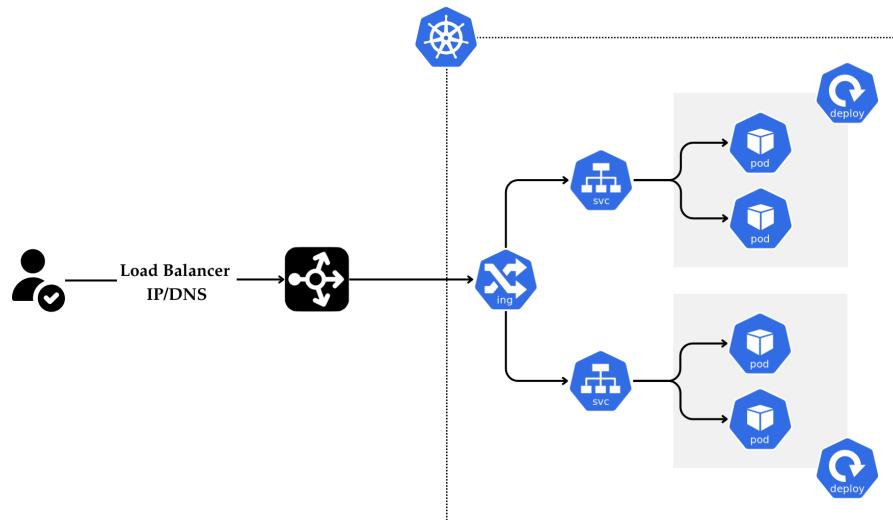


Figure 5.5: Ingress traffic flow

The following is an example of a simple *Ingress* YAML file (without TLS):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
      backend:
        service:
          name: test
          port:
            number: 80
```

Service mesh

An important characteristic of a distributed microservices application is service-to-service communication.

While a Kubernetes Service provides basic communication functionality by exposing Pods using a service name and then load balancing requests to them, this cannot meet advanced communication requirements and other functionalities that complex microservices deployments need. For example, it does Layer 4 (L4) or transport level load balancing only, which is based on IP address and port number, and is not capable of Layer 7 (L7) routing that operates at the application layer and can load balance based on advanced parameters, such as HTTP headers, cookies, URLs, content data, etc. Refer to the following link for more information on the layers of the **Open Systems Interconnection (OSI)** model:

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

Similarly, a growing number of microservices would need better observability. Microservices also need capabilities such as service security through authentication and authorization, traffic splitting, rate limiting, circuit breaking, encryption, etc. Coding these capabilities into each microservice becomes challenging as the complexity and number of microservices increase.

A service mesh is an infrastructure layer that abstracts away these capabilities. So, instead of coding them into each microservice, a service mesh provides a platform or framework that contains all of these and is automatically added to the application. It is added to the application, i.e., the microservices, to enable the capabilities mentioned earlier, such as which service can talk to which service, implementing a new version of a service but A/B testing, etc.

The following figure illustrates the general architecture of a service mesh (Frontend, Auth and Payment are examples of services):

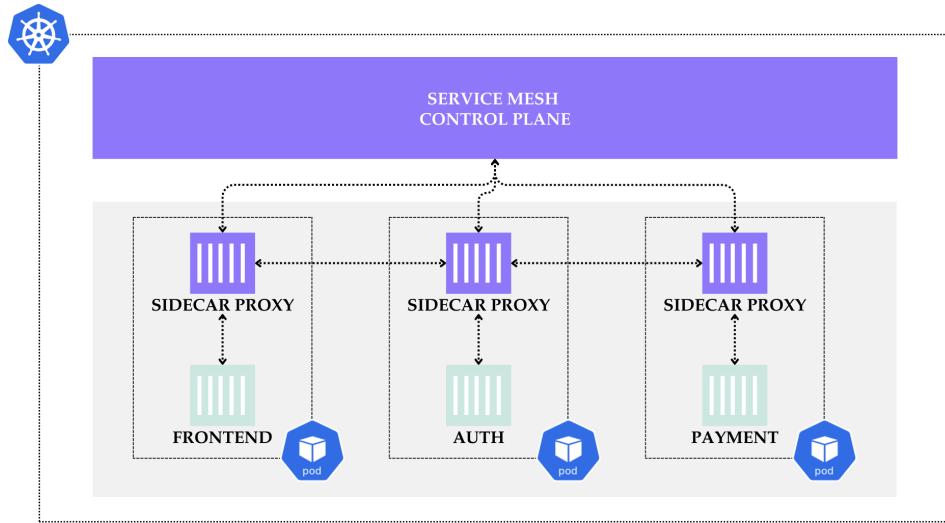


Figure 5.6: Service mesh architecture

Istio

Istio is an open-source service mesh, and a **Cloud Native Computing Foundation (CNCF)** graduated project. It provides a layer on top of distributed applications that provides service-to-service communication, load balancing, authentication and authorization, monitoring, and other security features, such as encryption with minimal to zero code changes.

It solves numerous challenges around traffic management, observability and security of services in a complex distributed microservices architecture. Additionally, it is extensible to handle a wide range of deployment needs outside of Kubernetes.

Istio's capabilities include intelligent routing, fine-grained access controls, and extensive telemetry, all of which contribute to enhanced performance, security, and operational efficiency in microservice environments. Through its powerful features, Istio simplifies the management of service dependencies and accelerates the adoption of microservices architectures in enterprise environments.

At a very high level, its architecture includes the control and the data planes:

- The **control plane** is responsible for managing and configuring the proxies to route traffic. It also enforces policies and collects telemetry.
- The **data plane** is made up of proxies, or **envoy proxies**, deployed as sidecars within each service Pod. These proxies intercept all network communication between microservices and perform various tasks such as traffic management, secure service-to-service communication using mutual TLS or mTLS, collect telemetry data for observability, etc.

Linkerd

Linkerd is also an open-source service mesh tool written in Rust. It is lightweight and provides enterprise level security, observability, and reliability to a Kubernetes cluster.

Here is a quick summary of its features that empower both developers and operations teams:

- **Automatic mTLS:** Linkerd enforces mTLS by default, ensuring encrypted communication between all services within the mesh. This adds a layer of protection against scenarios such as eavesdropping and data tampering.
- **Automatic proxy injection:** It automates injecting sidecar proxies into your applications, streamlining deployment, and ensuring all service communication flows through the mesh.
- **Granular authorization:** Linkerd provides fine-grained control over how services interact with each other. You can define authorization policies to restrict unauthorized access and enforce communication policies within your microservices ecosystem.
- **Integrated monitoring and tracing:** It offers comprehensive monitoring and tracing capabilities. You gain real-time insights into service health, request latencies, and error rates, allowing you to proactively identify and troubleshoot issues within your service mesh.

- **Traffic splitting and canary deployments:** Linkerd facilitates safe and controlled deployments. You can perform traffic splitting to gradually roll out new service versions and canary deployments to minimize risk and ensure a smooth transition for your users.

Linkerd is also a CNCF graduated project.

Consul

Consul is a product from HashiCorp. It is also a popular service mesh solution that seamlessly integrates with your existing infrastructure, acting as a transparent layer that orchestrates secure and reliable communication between services.

Here are some of Consul service mesh's features:

- **Zero-trust security:** Consul enforces a zero-trust security model, ensuring no service is inherently trusted. This is achieved through features like automatic mTLS encryption.
- **Service identity and RBAC:** It assigns unique identities to services, enabling granular control over access. You can define RBAC policies to dictate which services can communicate with each other and what actions they are authorized to perform. This approach minimizes the attack surface and strengthens the overall security posture of your microservices ecosystem.
- **Simplified traffic management:** It streamlines traffic management with features like service discovery, health checks, and load balancing. Services can easily discover each other and route traffic efficiently based on real-time health data. This reduces complexity and ensures consistent performance for your applications.
- **Enhanced observability:** Consul service mesh integrates with existing monitoring tools, providing valuable insights into service-to-service interactions. You can gain real-time visibility into request latencies, error rates, and overall service health, allowing for proactive troubleshooting and performance optimization.

- **Multi-platform support:** It is flexible, offering deployment options across various platforms, including Kubernetes, Nomad, and even virtual machines. This versatility allows you to leverage Consul Service Mesh within your existing infrastructure, regardless of your chosen container orchestration tool or deployment environment.

Traefik Mesh

Traefik Mesh is another lightweight option for service mesh capabilities. However, unlike other service meshes that rely on sidecar proxies, it uses a host-based approach.

It provides the following features and capabilities:

- **Lightweight and non-invasive:** Traefik Mesh throws away the sidecar model. Instead, it leverages proxy endpoints running on each cluster node, minimizing resource consumption and simplifying deployment. This non-invasive approach avoids modifying your existing container images or disrupting your application workflows.
- **Automatic service discovery:** Gone are the days of manual configuration. Traefik Mesh automatically discovers services within your Kubernetes cluster, eliminating the need for complex service registration processes. This streamlines service mesh adoption and reduces the potential for configuration errors.
- **Traffic management features:** It offers robust traffic management capabilities. You can define routing rules, implement load balancing strategies, and configure service health checks. This ensures efficient traffic distribution and optimal performance for your microservices.
- **Security by default:** Traefik Mesh prioritizes security. You can enforce mTLS encryption for secure communication between services within the mesh. Additionally, it integrates with existing authentication and authorization solutions, enabling granular access control within your microservices ecosystem.

- **Open source and extensible:** As an open-source project, it fosters a collaborative development environment. It boasts a rich plugin ecosystem, allowing you to extend its functionality to integrate with various tools and monitoring solutions within your cloud-native environment.

Storage

A Pod (with the containers inside it) is usually stateless. To run stateful workloads, one can provide persistent volumes to the Pod. All containers inside the Pod will share the volume.

To support different types of external storage, Kubernetes implements a pluggable storage layer using **Container Storage Interface (CSI)**. This means that you can bring in a storage system of your choice, provided they use CSI. A storage provider that supports Kubernetes will typically provide a CSI-based plugin for this.

StorageClass

A *StorageClass* works as a blueprint for provisioning persistent storage. It defines categories of storage with specific characteristics, allowing administrators to offer various storage options to their applications. For example, you might create a StorageClass named `high-performance` that provides SSD storage with a certain number of replicas for critical applications such as an application handling real-time data. Similarly, a StorageClass named `standard-performance` could offer standard HDD storage with one or no replicas for less demanding applications. Pods requesting storage simply reference the desired StorageClass in their `PersistentVolumeClaims (PVCs)`, and Kubernetes handles the provisioning process based on the pre-defined configuration automatically. This approach streamlines storage management, offering flexibility and control over how persistent storage is allocated within the cluster.

A StorageClass uses a provisioner to specify the volume backend (the CSI plugin) being used. You can set a StorageClass as the default option to be

used when a PVC does not specify one.

The following is an example of a StorageClass that uses VMware vSphere datastore as the backend:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: high-performance
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: zeroedthick
  datastore: SSDDatastore
```

PersistentVolume

A *PersistentVolume*, PV for short, is used to persist data even after the associated Pod terminates. They are provisioned manually or dynamically using a **StorageClass**, which defines the type and configuration of storage available. For example, a web application Pod that stores user-uploaded pictures on a persistent volume. If the Pod restarts due to an update or scaling event, the uploaded data is not lost and remains accessible as it is stored on the persistent volume, independent of the Pod's lifecycle.

The separation between storage provisioning (handled by PVs) and storage requests (made by applications through PersistentVolumeClaims) keeps your applications' storage needs decoupled from the underlying storage infrastructure. This approach ensures data persistence and simplifies storage management within the cluster.

The following is an example of a YAML file for a manually created PV of storage capacity 5GB using NFS:

```
---
```

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: web-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /tmp
    server: 191.168.100.100
```

PersistentVolumeClaim

Finally, a PersistentVolumeClaim represents requests for storage resources by the Pod, acting as a bridge between your applications and the available PVs. Let us take the same example of a web application needing persistent storage for user-uploaded images. The application does not care about the specifics of the underlying storage (disk type, location, etc.). Instead, it expresses its storage needs through a PVC, specifying aspects like size and access mode (read-write or read-only). Kubernetes then searches for a suitable Persistent Volume that matches the PVC's requirements and binds them together. This decoupling simplifies storage

management where applications declare their needs, and Kubernetes handles the allocation of appropriate storage resources from available Persistent Volumes.

In simple words, a `PersistentVolume` provides the actual storage resource, while a `PersistentVolumeClaim` is a request for storage made by an application. The PVC binds to a suitable PV based on requirements like capacity and access modes, allowing applications to use storage without worrying about the underlying infrastructure.

The following is a YAML version of a PVC that requests for a storage of size 5GB:

```
---
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: web-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

In [*Chapter 9: Hands-on Kubernetes*](#), we will see a practical exercise on PV/PVC and how to use a PVC in a Pod.

Conclusion

Container orchestration is complex and involves multiple components and infrastructure resources such as runtimes, storage, networking, service

mesh, security, and more. Runtimes such as *Containerd* provide the container engine that helps manage the lifecycle of containers in a cluster. Storage backends, combined with Kubernetes resources such as *PV* and *PVC*, provide data persistence capabilities. Networking resources such as *services*, *Ingress*, *DNS* provide basic networking functionalities to access workloads from within the cluster and outside. Additionally, *service mesh* enables advanced communication capabilities required by microservices design patterns. Finally, security is crucial in Kubernetes. At a basic level, you must implement recommended controls to secure your cluster such as *encryption*, *RBAC*, *security updates* and similarly the workloads using resources such as *Secrets*, *ConfigMaps* and *Network Policies*.

In the next chapter, which covers the *cloud-native architecture* domain of the exam, we will learn dig a bit deeper into the tenets of a cloud-native architecture and see how to implement some of these tenets in Kubernetes. We will also learn about the important of governance and open standards in standardizing cloud-native followed by a list of cloud-native job roles and their responsibilities.

Multiple choice questions

1. Which of the following statements accurately describes the primary function of an ingress?
 - a. It exposes HTTP(S) routes between Services within the cluster.
 - b. It exposes HTTP(S) routes between Pods within the cluster.
 - c. It exposes external HTTP(S) routes to Services inside the cluster.
 - d. It exposes external HTTP(S) routes to Pods inside the cluster.
2. What of the following statements accurately describes a network policy?
 - a. A VPN that allows you to connect to your cluster securely.

- b. A gateway service that connects two clusters in different networks.
 - c. A set of controls that help implement network compliance.
 - d. A set of rules that control network traffic between Pods and other endpoints.
3. Which of the following are different types of services in Kubernetes?
- a. ClusterIP
 - b. NodePort
 - c. LoadBalancer
 - d. All of the above
4. Which of the following networking solutions would you use to externally expose HTTP and HTTPS routes for a service?
- a. Load balancer
 - b. Ingress
 - c. Firewall
 - d. VPN
5. Which of the following service mesh tools uses the Envoy service proxy?
- a. Istio
 - b. Linkerd
 - c. Consul
 - d. Traefik
6. What are the possible policy types for a network policy?
- a. Ingress
 - b. Egress

- c. Both a and b
 - d. Neither a or b
7. Which Kubernetes component provide load balancing, SSL termination, and name-based virtual hosting?
- a. Service
 - b. Kube-proxy
 - c. Kube-dns
 - d. Ingress
8. Which of the following can be used to make a service accessible from the internet?
- a. NodePort
 - b. LoadBalancer
 - c. Ingress
 - d. All of the above
9. Select the components of a service mesh.
- a. Control plane
 - b. Management plane
 - c. Data plane
 - d. Both a and c
10. Which service extends ClusterIP service type?
- a. NodePort
 - b. LoadBalancer
 - c. Ingress
 - d. Headless
11. Which of the following service types does not have an IP?
- a. Headless service

- b. ClusterIP service
- c. NodePort service
- d. LoadBalancer service

Answers

1	c.
2	d.
3	d.
4	b.
5	a.
6	c.
7	d.
8	d.
9	d.
10	a.
11	a.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 6

Cloud-Native Architecture

Introduction

In this chapter, we will begin with a recap of cloud-native fundamentals and learn about the key tenets of a cloud-native architecture. This chapter will also cover and, in some cases, revisit how to implement resiliency, autoscaling, and serverless in Kubernetes.

Cloud-native architectures using open-source technologies depend heavily on community, hence, we will learn about governance and open standards that are necessary for standardization. Finally, the chapter will provide a list of some of the cloud-native roles and their responsibilities.

This chapter covers the topics from the *Cloud-Native Architecture* domain, which makes up 16% of the exam, and includes the following official objectives:

- Cloud-native architecture fundamentals
- Autoscaling in Kubernetes
- Serverless
- Community and governance
- Personas

- Open standards

You may find that there are some overlaps in this chapter from [*Chapter 2, Understanding Containers and the Need for Container Orchestration*](#), where some of these concepts were briefly covered. This should act as a revision.

Structure

This chapter has the following topics:

- Cloud-native architecture fundamentals
- Key tenets of a cloud-native architecture
- Resiliency in Kubernetes
- Autoscaling in Kubernetes
- Cloud-native security
- Security in Kubernetes
- Serverless
- Community and governance
- Open standards
- Cloud-native personas

Objectives

The objective of this chapter is to introduce learners to some of the ways resiliency and autoscaling can be achieved in Kubernetes along with an introduction to serverless, its benefits and use cases. The chapter also aims to cover fundamental cloud-native security concepts such as the 4 C's of Cloud Native Security, zero trust and defense-in-depth. It is also the objective of this chapter to cover governance and standards, especially concerning the CNCF. Finally, the chapter aims to provide an overview of various job roles found in cloud-native environments and their responsibilities.

Cloud-native architecture fundamentals

As introduced in [*Chapter 1, Stepping up to Kubernetes and Cloud-native*](#), cloud-native is a modern approach to application development and deployment.

Quoting and slightly simplifying from the CNCF cloud-native definition:

Cloud-native architectures use techniques such as containers, service meshes, microservices, immutable infrastructure, and declarative APIs to enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

In this, we will explore this in more detail, learning what makes up a cloud-native architecture and some of its benefits.

Cloud-native architecture

A cloud-native architecture is an approach to application development and deployment that provides scalability, resiliency, agility, and speed. This approach uses cloud-native principles and technologies. Cloud-native architectures are also expected to be observable, secure, and automated.

A cloud-native architecture stack can be divided into two parts: the application and the infrastructure on which the application runs.

Cloud-native applications are developed as loosely coupled microservices, packaged into containers, and deployed using DevSecOps principles. These techniques provide the resiliency, scalability, agility, and speed necessary for modern applications. These applications are deployed to cloud-native infrastructures based on technologies such as Kubernetes that provide resiliency, agility, and scalability at the infrastructure layer. Cloud-native infrastructures are heavily driven by abstraction, APIs, and automation and must address security. From a business perspective, the preceding translates to a faster time-to-market and lower interruptions, providing a competitive edge.

Cloud-native applications can be deployed in a public cloud platform such as AWS, Azure, or GCP, a private cloud, or on-premises infrastructure running container orchestrators such as Kubernetes (Kubernetes in bare metal).

An agnostic cloud-native architecture uses open-source software implementing open standards, which we covered in [Chapter 2](#) and will recap in this chapter later, and technologies such as Kubernetes that can run on any cloud provider.

If you are deploying cloud-native applications on public cloud platforms, the *Well-Architected Framework* is recommended. Cloud vendors provide a way to assess your infrastructure against this framework and provide recommendations as well. Cloud-native applications work well in cloud infrastructures.

The Twelve-Factor App

Building cloud-native applications can pose many questions and overwhelm complex systems. The Twelve-Factor App methodology can provide a solid foundation for developers and ops engineers who build, deploy, and manage cloud-native apps.

Read more about this at: <https://12factor.net/>

Key tenets of a cloud-native architecture

Now, let us look at some key tenets of a cloud-native architecture.

This chapter covers scalability (autoscaling) and resiliency in Kubernetes, while other tenets, such as observability and automation, are covered in later chapters [Chapter 7: Cloud-native Observability](#) and [Chapter 8: Cloud-native Application Delivery](#).

Scalable

One of the most important tenets of cloud-native architecture is its scalability, more specifically, rapid and high scalability. Both applications

and infrastructure should be able to scale as per demand.

At the application layer, *stateless* microservices provide better scalability than stateful applications. *Containerizing* microservices aids in faster scalability as they are lightweight and have a shorter boot time.

At the infrastructure layer, using a container orchestration solution such as Kubernetes in a cloud environment will provide the needed scalability. A managed Kubernetes cluster from a cloud service provider provides the fastest and highest scalability.

Finally, scalability should balance performance and cost. Using autoscaling should help you scale out when there is demand and scale in when there is less demand.

Loosely coupled

Coupling or dependence among different components or services of an application plays an important part in the resiliency of a system. A highly resilient cloud-native architecture maintains as much loose coupling, also called **decoupling**, as possible. One of the important advantages of loose coupling is independence. Each component in a loosely coupled architecture, whether at the application or the infrastructure layer, functions independently with minimal impact on other components. They can be updated and maintained individually; if one of the components fails, it does not impact the entire system; they are individually scalable, creating a highly flexible architecture.

Microservices in a loosely coupled architecture communicate through RESTful APIs or message queues. This form of communication is typically **asynchronous** but can be **synchronous** too.

Event driven architectures also follow a loosely coupled approach. This is covered in more detail later in the chapter when discussing *Serverless*.

Note: While loose coupling is a best practice, applications may consist of microservices in a tightly coupled fashion as well.

Resilient

Given the many movable parts of a cloud-native architecture, one must expect that something at some point may fail. A resilient architecture, both application and infrastructure in this case, is one that can continue to function in case of failure. High availability is one of the properties of a resilient system. Automation and observability are key to implementing a resilient system.

Observable

In order to take autoscaling or self-healing actions, one needs to observe or monitor the applications and infrastructure in a cloud-native system. Autoscaling and self-healing are only some examples that depend on observability; there are many more actionable insights that can be derived from observability.

Cloud Native Computing Foundation (CNCF) projects include observability tools such as *Prometheus* that can be used to observe Kubernetes and more. We discuss observability in more detail in [Chapter 7, Cloud-Native Observability](#).

Automated

Cloud-native architectures are expected to be agile and fast. You should be able to make changes and upgrades faster in a repeatable manner. This is achieved using automation. Automation is discussed in more detail in [Chapter 8, Cloud-Native Application Delivery](#).

Resiliency in Kubernetes

Resiliency in Kubernetes is driven by one of its core principles, i.e., *desired state*. However, to achieve this, you must use certain resources and API objects when defining the desired state (the YAML file).

Let us revisit the most important ones from the exam point of view. These concepts have already been discussed in [Chapter 4, Kubernetes](#)

Basics, and [Chapter 5, Container Orchestration with Kubernetes](#), and should serve as a recap again.

Deployments

We discussed earlier that stand-alone Pods do not provide any kind of scaling, resiliency or other cloud-native capabilities. So, one fundamental way to ensure these properties can be applied to an application is to use Kubernetes objects called **Deployments**. They support replicas, rollouts, scaling, and more.

ReplicaSets

ReplicaSets are used to maintain a desired number of Pod replicas at any given time. It can be used to implement self-healing where if one of the Pods goes down, a replacement Pod is automatically created to maintain the number of replicas defined. Changing the number of replicas scales the workload.

Deployments and ReplicaSets complement each other where you would create a deployment to run a scalable stateless application that uses ReplicaSets, which in turn manages the desired number of replica Pods.

StatefulSets

StatefulSets are used to run stateful workloads, such as databases, and provide resiliency to workloads that require stability, guaranteed ordering, unique networking, and, importantly, deal with persistent data. These types of resources help maintain high availability of stateful workloads and recover from failures similar to ReplicaSets, where if a Pod fails, it will be rescheduled. Changing the number of replicas scales the workload.

Probes

Kubernetes provides liveness, readiness and startup probes that help in creating a resilient system. These probes are used to implement some of the self-healing capabilities, which are as follows:

- *Liveliness* probes are used to determine when to restart a container(s).
- *Readiness* probes are used to determine when to send traffic to the container(s).
- *Startup* probes help determine if a container has started. When using startup probes, liveliness and readiness probes start after it succeeds.

Storage resiliency

For the resiliency of the storage layer in Kubernetes, one must look at adopting cloud-native storage technologies that provide data protection capabilities such as distributed architecture, high availability of storage nodes, snapshots, BCDR capabilities, etc. Some examples of such storage technologies are *Portworx* by Pure Storage, *Longhorn* (originally by Rancher), *Rook*, etc.

Rook is an open-source cloud-native, CNCF graduated project that automates deployment and management of *Ceph* to provide self-managing, self-scaling, and self-healing storage services.

Longhorn is an incubating CNCF project that provides distributed block storage for Kubernetes.

Note: Ceph is a distributed storage system that provides file, block and object storage and is deployed in large scale production clusters.

Network resiliency

As discussed in earlier chapters, Pod networking is dynamic. In network resiliency, an example is when Pods are recreated in events such as self-healing, they will have a different IP address than earlier. There are many more such network-related complexities that arise in a microservices architecture, which raises questions about network resiliency. Kubernetes services are already available to help with this, but it is not enough.

Consider solutions such as Ingress controllers, load balancers, and service mesh.

Also, look at the (relatively new) Kubernetes Gateway API for Ingress. In depth knowledge of Kubernetes networking is not necessary for the KCNA exam.

Autoscaling in Kubernetes

Kubernetes provides capabilities that allow scaling of applications and infrastructure. Production deployments use a combination of one or more of these for optimal scaling that has a balance of performance and cost.

Horizontal Pod Autoscaler

Horizontal Pod Autoscaler (HPA) is an autoscaling capability in Kubernetes that scales Kubernetes resources, such as deployments, by adding or removing Pods in response to application demand.

HPA supports the default resource metrics, custom metrics, and external metrics. For example, you can scale workloads based on the number of messages in a queue or based on the payload size of an HTTP request. A YAML configuration defining the required HPA characteristics for the application is created, which is then applied to the cluster.

Following is an example of an HPA:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: kcnaprep-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
```

```
kind: Deployment
  name: kcnaprep
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 75
```

Horizontal Pod autoscaling does not apply to objects that cannot be scaled, such as *DaemonSets*.

An HPA can also be created using the imperative approach using the **kubectl autoscale** command. Consider the following example:

```
kubectl autoscale rc kcnaprep --min=2 --max=10 --cpu-percent=75
```

You can also manually scale a deployment in Kubernetes using the **kubectl scale** command. An example of the command is provided as follows:

```
kubectl scale --replicas=3 deploy/kcnaprep
```

Vertical Pod Autoscaler

Unlike HPA, Vertical Pod Autoscaler (VPA) is an autoscaling feature that dynamically adjusts the resource *requests* and *limits* the parameters of the container(s) in a Pod based on usage patterns. This type of scaling is also called **scale-up** and **scale-down**. This helps avoid unnecessary

overprovisioning and improves resource utilization efficiency, hence providing a balance between performance and cost-effectiveness (optimal CPU and memory resources, right size).

VPA is a separate program and is implemented using a **custom resource definition (CRD)** object and requires that the *metrics server* is installed on the cluster.

Consider the following example of a VPA:

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: kcnaprep-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: kcnaprep
  updatePolicy:
    updateMode: "Auto"
```

Cluster Autoscaler

The Cluster Autoscaler is an autoscaling program that automatically adjusts the size of the cluster by dynamically adding or removing nodes based on current resource demands. As workloads fluctuate, the Cluster Autoscaler optimizes resource utilization, ensuring that there are enough nodes to accommodate the application's requirements without unnecessary over-provisioning.

It leverages a set of predefined policies and monitors the cluster's state, detecting scaling needs and initiating node adjustments accordingly. In the background, the Cluster Autoscaler talks to the cloud provider API (AKS, GKE, EKS, etc.) to create or delete nodes.

Note: Cluster Autoscaler does not scale based on actual resource utilization.

The following conditions can trigger an autoscaling event:

- Pods are unable to run due to insufficient resources (*resource request* value), hence going into a *pending* state, where additional nodes are created.
- Underutilized nodes in the cluster, where these nodes are removed from the cluster, and their Pods are moved to other nodes.

Kubernetes Event-driven Autoscaling

Kubernetes Event-driven Autoscaling (KEDA) extends HPA (external metrics) to enable event-driven autoscaling for workloads. Event sources can be Azure Functions, AWS SQS, Kafka, etc., which are monitored using scalers, which in turn scale in and out of Pods. This also means that KEDA can support scaling down the number of Pods to zero when there are no events to process.

Cloud-native security

Cloud-native applications and infrastructure have many different movable parts which introduces a wider and more complex attack surface, making traditional security measures inadequate. Here, we will discuss security principles that can be applied to cloud-native applications and infrastructure.

These practices and principles are not mutually exclusive, and you can use the best possible combination depending on the need.

4 Cs of Cloud Native Security

Let's revisit the 4 Cs of Cloud Native Security, which are as follows:

- **Cloud:** This layer of cloud-native security focuses on securing the cloud infrastructure used to deploy the Kubernetes cluster. Every cloud vendor publishes their respective cloud security guidelines and best practices, and most also provide a native service for **cloud security posture management (CSPM)**. This includes security for identity and access management, network, security practices for APIs, encryption etc.
- **Cluster:** Cluster security is all about securing Kubernetes and its cluster components. This involves authenticating and authorizing API requests, encrypting API communications, **role-based access control (RBAC)**, etc., and applying network policies for the security of workloads within the cluster, such as Pod-to-Pod communication, etc.
- **Container:** To deploy an app or a microservice in platforms such as Kubernetes, they are packaged into containers. This layer deals with practices for securing container images. It includes best practices such as scanning container images, including base images, signing images, using trusted images, and a trusted container registry.
- **Code:** At this layer, the responsibility lies more with the developers to write secure code. Scanning code and dependencies for vulnerabilities and signing generated artifacts are some of the recommended steps.

Zero trust

Zero trust architecture to security is a set of practices implemented with the core principle *never trust, always verify*. Users, devices, or applications are never trusted, and access is always verified, even if they are inside a trusted boundary. Traditionally, an entity is assumed trusted if it is inside a network perimeter, which poses a threat in situations where a malicious actor gains access to a trusted entity as they can use that trusted entity to laterally move until they gain access to a golden

ticket. Since there is no trust in a zero trust architecture, there is constant verification at every stage, thus reducing the attack surface area.

Defense-in-depth

With the aim of protecting data, the defense-in-depth approach uses a layered approach to security creating multiple barriers to cross for accessing data. This helps slow down the speed of attack, giving you the opportunity to identify (through proper detection mechanisms) and take necessary remediation steps.

Depending on whether you are using a cloud service provider or managing your own data center, the responsibility of some of these layers may change. For example, when using a cloud service provider, the physical layer security is the provider's responsibility.

Security in Kubernetes

Here is a quick recap of the mechanisms that can be used for securing Kubernetes cluster components and workloads:

- **RBAC:** Use Kubernetes RBAC to regulate access to clusters and resources. It is implemented by **rbac.authorization.k8s.io** and follows a deny-by-default policy. Allow rules to decide what a requester can do. Create a *Role* to set granular permissions at a single namespace level and use *RoleBinding* to assign this role to an identity. Similarly, create a *ClusterRole* for setting one or more permissions across all namespaces and use *ClusterRoleBinding* to associate it to identities.
- **Secrets:** Use them to store sensitive information such as passwords, tokens, etc. Secrets support encryption, but data stored is not encrypted by default.
- **ConfigMaps:** Use ConfigMaps for non-sensitive data such as environment variables, arguments, etc.
- **Network policies:** Implement network policies when looking to control east-west traffic between Pods. Network policies can be

used to restrict Pod communication based on namespaces, IPs and more. Calico is an example of a network plugin that helps implement network policies.

Note: It is recommended that you are aware of fundamental security terminologies, such as encryption, SSL/TLS, PKI, certificates to help understand certain questions in the exam.

Serverless

Serverless architectures are seen as the next step in the evolution of modern application development and execution process. These architectures emphasize on developing and running code using an approach that does not require administration and management of servers. Developers write code as *functions*, which are executed using a *trigger* or on an *event* and billed for computing only if the function runs. Any backend services required for running this code, such as storage, database, API gateway, messaging, etc., are also serverless.

Serverless architectures use **Functions-as-a-Service (FaaS)** for computing, which typically includes an API gateway for routing requests and triggering the correct function, and **Backend-as-a-Service (BaaS)** for other services such as storage, databases, integrations, etc. They complement microservices architectures.

Benefits

Serverless architectures focus on high scalability and zero server administration. Hence, while there are servers in the background, without which it is not possible to run applications, it is called serverless as there are no servers to manage. This is especially beneficial for developers as they can now focus on only the business logic and application code. Managing servers and other backends becomes the provider's responsibility, and you pay based on consumption, for example, only when the code runs.

In comparison, cloud instances, i.e., IaaS, still have a degree of server management, such as patching, updates, security, runtime installation, etc. Even containers and container orchestrators such as Kubernetes, including managed Kubernetes offerings, require some maintenance, and you pay for idle resources.

As an analogy, serverless computing or platforms are like ride hailing apps. You order a car or a bike only when you need it. You pay for the duration of the ride (and maybe some other convenience fees). You do not own or pay for its maintenance and upkeep.

Serverless architectures can provide massive scalability and cost efficiency, especially when deployed on managed serverless platforms provided by cloud vendors.

Serverless computing can be used in a variety of use cases, such as:

- Jobs
- Data processing
- Stream processing
- Web applications
- Parallel workloads

Serverless computing may not be an ideal choice for long running processes.

Functions-as-a-Service

FaaS is a subset of the serverless architecture and is a computing service model that abstracts the application layer. So, instead of focusing on creating complete applications or microservices, developers focus on individual *functions* that make up the application or service. They write and maintain each task (or subroutine) of an application separately. Referring to our fintech application use case from [Chapter 1, Stepping up to Kubernetes and Cloud-Native](#), validating the QR code could be considered a task or a function. This function is then stored and

executed, mostly using containers, through a FaaS compute offering based on a trigger or event. Hence, it is also referred to as **event-driven computing**. FaaS sits in between **Software-as-a-Service (SaaS)** and **Platform-as-a-Service (PaaS)**.

While we already know that functions can be invoked using triggers or events, they can also be invoked using HTTP requests or another function.

FaaS offerings execute these functions in lightweight execution environments (preconfigured to support certain programming frameworks), which are mostly containers but can also be other forms of custom-built execution environments. Some platforms are also known to provide FaaS services using light weight VMs.

The benefits of FaaS include:

- **All eyes on code:** Developers get to focus on writing code (functions) without worrying about infrastructure provisioning and management.
- **Faster releases:** All eyes on code means developers can release new code, features, fix bugs faster.
- **Efficient scaling:** FaaS workloads can scale up based on demand and, more importantly, scale down to zero during idle times.
- **Cost efficient:** Billing in serverless is based on a consumption model. Hence, you pay only if the code runs. There is no cost incurred for computing during idle times when the workload has scaled down to zero.

Notes of caution

While the preceding benefits of serverless and FaaS are tempting, it is important to note that developers must write (or modify existing) code that can run in a serverless environment, which, in some cases, can be a considerable effort. They must also consider factors such as *cold start*, which may affect application performance.

Similarly, while this model is cost-effective, it may also result in huge costs when there are many triggers, such as receiving millions of requests per second or minute.

Additionally, since a serverless architecture may include other backend services, such as storage, database, integration, etc., there may still be charges which are always incurred.

Managed versus self-managed serverless

Users can choose to use serverless solutions from cloud providers or build their own serverless platform using open-source or custom solutions.

The scalability of serverless platforms depends heavily on the underlying infrastructure. Managed serverless platforms from cloud providers can scale as much as is required; however, a self-managed serverless platform, for example, on a self-managed Kubernetes cluster, can only scale in proportion to available resources (worker nodes).

Additionally, self-managed serverless platforms provide more control than serverless platforms provided by cloud vendors, such as control over runtime, infrastructure, configuration of code, testing, etc. However, at the same time, a self-managed serverless platform will have more responsibilities, especially around scaling parameters to ensure that there are enough infrastructure resources to provide high scalability.

Open source serverless solutions in Kubernetes

The following is a list of open source serverless platforms for self-hosting and managed platforms for reference. The most common serverless platforms that work with Kubernetes are:

- **Knative:** This is an open-source CNCF incubated project originally created by Google along with other contributors. Knative enables Kubernetes to run serverless workloads. Knative has two main components, which are as follows:

- **Serving:** This runs on serverless containers using the containerized code and takes care of networking, autoscaling, and revision tracking. It can scale down to zero.
- **Eventing:** This component takes care of the trigger management. The triggers are based on Kubernetes API events.
- **OpenFaaS:** OpenFaaS is another serverless platform that offers developers an easy way to deploy event-driven functions and microservices to Kubernetes without repetitive, boiler-plate coding by packaging the code into an **Open Container Initiative (OCI)** compatible image. It features autoscaling, metrics and more.
- **faasd:** This is a variation of OpenFaaS that can run on a single host and uses containerd and CNI under the hood along with some OpenFaaS components.
- **Apache OpenWhisk:** This serverless platform was initially developed by IBM. It works with container frameworks such as Kubernetes, OpenShift and Docker Compose but requires Docker for running containers. IBM Cloud Functions is based on Apache OpenWhisk.

Managed serverless platforms

The following are some examples of various serverless solutions from the popular cloud vendors:

Computing	AWS Lambda
	Azure Functions
	Google Cloud Functions
Storage	AWS S3
Database	Amazon DynamoDB
	Amazon Aurora Serverless

	Azure CosmosDB
	Amazon API Gateway
Integrations	Amazon SQS
	Amazon SNS

Table 6.1: Examples of serverless solutions from different CSPs

Community and governance

Kubernetes is an open-source product that is heavily driven by the community. Since a community has contributors who are individuals or belong to various organizations, there is a need for governance to build a product that meets various standards such as maturity, adoption, security, enterprise readiness, etc.

The CNCF community consists of developers, users, non-developer contributors, advocates and more.

The Kubernetes community has four core principles:

- Open
- Welcoming and respectful
- Transparent and accessible
- Merit

The community also abides by a code of conduct, which means all members must abide by it too. An excerpt from the code of conduct is as follows:

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

To read more about the Kubernetes code of conduct and community values and to join the community, visit <https://kubernetes.io/community/>.

The following table describes the CNCF structure and the governance approach:

Governing Board (GB)	<p>The GB is responsible for marketing and other business oversight and budget decisions for the CNCF. They do not make technical decisions for the CNCF other than working with the TOC to set the overall scope for the CNCF.</p>
Technical Oversight Committee (TOC)	<p>The TOC provides technical leadership to the cloud-native community. Their functions include:</p> <ul style="list-style-type: none">• Defining and maintaining the technical vision for the CNCF• Approving new projects and creating a conceptual architecture for the projects• Aligning projects and removing or archiving projects• Accepting feedback from end user committee and mapping to projects• Aligning interfaces to components under management (code reference implementations before standardizing)• Defining common practices to be implemented across CNCF projects
Special Interest Groups now called SIGs	<p>SIGs, under the guidance of TOC, provide high-quality</p>

	Technical Advisory Groups (TAGs)	technical expertise, unbiased information and proactive leadership within their category. They essentially act as advisors to the TOC and bring in expertise in specific areas. An example of an SIG is security. They report to the TOC.
End User Community (EUC)	EUC are the end users of the CNCF projects, mostly organizations. They use cloud-native technologies to build products and in return provide requirements and feedback on these projects.	
	End Technical Advisory Board (TAB)	The CNCF TAB acts as a vital voice of end users within the CNCF community. It plays a key role in advancing topics of concern to end users, enhancing visibility into end user adoption of CNCF projects, and raising awareness about the needs and perspectives of end users in the cloud-native ecosystem.

Table 6.2: Committees and their responsibilities in governance

The committees may have sub-committees.

Open standards

Open standards have been covered in great detail in *Chapter 2, Understanding Containers and the Need for Container Orchestration*, covering topics such as OCI, CRI, CNI, CSI, etc.

The following table quickly summarizes them in the context of the exam. The exam does not deep dive into them:

Open Container Initiative (OCI)	<p>OCI is a Linux Foundation project with its own open governance structure that creates and maintains open standards for container runtimes, images, and distribution. The OCI standard applies to low-level runtimes. Examples of runtimes that are OCI compliant are runC, crun, runhcs, etc. <i>RunC</i> was donated by Docker and is the default native runtime used in most implementations.</p>
Container Runtime Interface (CRI)	<p>CRI is a plugin interface that allows <i>kubelet</i> to use an OCI compatible container runtime. Kubelet talks to the container runtime using gRPC. Examples of runtimes that support CRI are CRI-O, containerd, gVisor, kata containers, etc.</p>
Container Network Interface (CNI)	<p>CNI is a networking standard that consists of a specification and libraries for creating plugins that can be used to configure networking in containers. Specifically, in Kubernetes, a CNI plugin is</p>

	<p>required to implement its networking model, cluster networking and to implement capabilities like <i>Network Policies</i> which can be used to control traffic flow within and outside the cluster.</p> <p>Examples of CNI plugins are Calico, Flannel, Weave, etc.</p>
Container Storage Interface (CSI)	<p>CSI is also a plugin interface, albeit for storage, that enables the use of third-party storage products with container orchestrators such as Kubernetes through CSI volume drivers.</p> <p>For example, using Ceph, Dell EMC, GlusterFS, or cloud provider options such as EBS, GCE persistent disks, Azure disks, etc.</p>
Service Mesh Interface (SMI)	<p>On a high level, service mesh facilitates communication between different parts of an application and plays an important role in complex microservices architectures.</p> <p>SMI is a standard that specifies common service mesh capabilities such as traffic policy, traffic telemetry, traffic management etc.</p> <p>An example of service mesh is Istio.</p>

Table 6.3: Open standards overview

Cloud-native personas

Cloud-native touches multiple areas, not just tools, and technologies. Working with cloud-native involves designing and developing applications, designing and implementing infrastructure, monitoring the apps and infrastructure, cost management, securing it, and more.

A single team or role cannot meet these varied levels of skill and experience requirements. Hence, it is obvious and recommended to have a separation of duties through roles. Organizations into cloud-native are observed to have one or more of the following roles depending on their stage of adoption, maturity etc.

Cloud architect

A cloud architect typically designs the overall system and possesses not just technical skills (cloud or otherwise) but business skills, soft skills etc. A cloud architect is aware of multiple vendors, services, technologies, etc., on a high level that can be used to design the complete solution.

Cloud architects are expected to be familiar with all the nuances of a cloud-native architecture (both applications and infrastructure) to design an optimal solution. In many organizations, they are also responsible for showing the ROI and TCO for adopting cloud-native. Hence, they should be familiar with cost calculators, TCO, and ROI tools, the difference between CapEx and OpEx models, etc.

Cloud solutions architect

In comparison to a cloud architect, a cloud solutions architect has deeper expertise on a given cloud vendor or service. For example, an Azure Cloud Solutions Architect will design a solution to implement an ERP system on Azure. There may be multi-cloud solutions architects as well. These roles are seen more in organizations focused on multiple public cloud platforms, such as service companies.

Cloud engineer

The role of a cloud engineer involves more operational tasks at the cloud platform level. The responsibilities include but may not be limited to:

- Deploying cloud resources as per the agreed design
- Configuration management
- Security
- Performance optimization
- Monitoring and troubleshooting cloud infrastructure

Given the preceding responsibilities, cloud engineers are expected to possess skills across many areas, such as automation, IaC, scripting, cloud security, etc.

A cloud engineer may also be involved in cloud architecture design stages and sometimes work on multiple cloud platforms simultaneously.

DevOps engineer

Traditionally, development and operations teams have been known to work in silos. DevOps is a practice that emphasizes bridging this gap with the use of people, processes, and technologies.

A DevOps engineer is the *people* part of the puzzle who use DevOps practices and tools to create, manage, and release software. As the name suggests, they will have skills in *development* and *operations*. A professional in the DevOps engineer role will typically be familiar with the entire stack of an end-to-end software delivery lifecycle which includes skillsets around development and operations. The skillsets can be broadly categorized into the following:

- Development skillsets include Git (source and version control), continuous integration/continuous deployment (CI/CD) pipelines, configuration management, testing, etc.
- Operations skillsets include infrastructure provisioning, mostly using Infrastructure as Code (IaC) and automation, system

administration, storage, some level of database familiarity, networking, etc.

This may feel overwhelming, but DevOps engineers are generally expected to be familiar with a lot of these tools, technologies and processes and more. This is precisely the reason these roles are high paying and in high demand.

DevOps engineers play an important role in advocating cloud-native practices with their organizations and outside.

Site Reliability Engineer

While the DevOps role focuses on the complete lifecycle of application delivery from development to deployment, the **Site Reliability Engineer (SRE)** role is more focused on the reliability of the environment running the application, which is primarily the production environment. Reliability is an umbrella term that includes availability, scalability, resiliency, uptime, etc.

SREs use observability and automation to take corrective actions such as scaling and self-healing to ensure reliability of a system. In some cases, they may use AI and ML to predict possible issues and take proactive actions. SREs may even go to the extent of automating these proactive actions. A failure mindset is important for the SRE role.

An important responsibility of the SRE team is to create a monitoring baseline. They work with stakeholders in defining various service level metrics which form this monitoring baseline and are important to measure reliability. Here is a quick definition of these metrics:

- **Service level objective:** A service level objective (SLO) is a specific, measurable target that defines the acceptable level of performance or reliability for a service or system over a given period. For example, the percentage of failed requests for an API service should be less than 10% in a month.

The role of an SREs will be to ensure that the *percentage of failed requests* remains below 10%.

- **Service level agreement:** Service level agreement (SLA) is an explicit or implicit contract between the provider and the consumer that outlines the expected level of service (SLO) and the consequences if the level of service is not met. SLA may also define vendors responsibilities in case of an issue. For example, if the percentage of failed requests is (any number) above 10%, then the SLO has not been met and as a consequence, customers get 100 free API calls. This is just an example, in the real world, there is a slab model for such scenarios.
- **Service level indicator:** Service level indicator (SLI) represents a quantifiable measurement of a key metric of the service. In our example, this key metric is *failed requests*. An example of SLI would be the percentage of failed requests in the last month was 7%. This also means that SLO was met.

Exam tip: The SRE role is the focus on the exam.

Security engineer

A security engineer's responsibilities primarily revolve around the security of cloud-native infrastructure, mostly known to be focused on public cloud infrastructures. Following is a list of tasks a security engineer may perform:

- Cloud security posture management is based on recommendations and best practices. They may use tools and services to aid in the implementation of these.
- They work with multiple teams, such as developers, data engineers, operations, DevOps, and SRE for implementing security best practices.
- They advocate the need for security at every stage of the lifecycle, share new security insights, etc.

Security engineers must have a continuous learning mindset and keep themselves updated on new and emerging threats.

DevSecOps engineer

These roles call for DevOps engineers with security skillsets. They apply DevOps principles and practices to security and advocate a shift-left approach to security. DevSecOps engineers incorporate security into all stages of the DevOps pipeline.

Data engineer

Data engineers are responsible for designing, building, and maintaining secure data architectures and infrastructure in a cloud-native environment. Apart from core responsibilities that include data modeling, schema design, data integration, and designing efficient and scalable data processing pipelines, their role may also involve ensuring data availability, compliance, monitoring, and troubleshooting.

Data engineers are also expected to incorporate DevOps practices into data processing, such as automation of pipeline creation, teardown, testing, etc.

Full stack developer

The role of a full stack developer is very common these days in startups as well as mature organizations. Since a full stack developer can work on frontend and backend applications, they are highly sought after. A full stack developer can work on every layer of the application stack, from user interface design to server-side logic and database management. This versatility allows them to contribute to various stages of the development lifecycle and make informed decisions about technology choices based on project requirements.

A full stack developer knows more than one programming language and usually works on languages such as Go, Python, JavaScript, .NET core,

etc. They follow development best practices such as version control, secure and clean coding, and use a continuous testing approach.

Becoming a full stack developer requires putting a lot of effort into learning multiple programming languages, being familiar with microservices and serverless patterns, and having a problem-solving mindset. They are also familiar with technologies such as containers and Kubernetes. A continuous learning mindset is important for a full stack developer, given the fast pace of change in technology today. They must be up to date with industry trends, emerging technologies, and best practices and quickly adapt to new tools and technologies.

Conclusion

Cloud-native architectures are key to developing and deploying modern scalable applications rapidly in an agile and resilient way. Architectures using open-source technologies like Kubernetes use open standards and rely on a strong community and a robust governance model. There are cloud-native personas that possess the necessary skills to design, implement, secure, manage, and monitor these architectures.

In the next chapter, we will look into the next domain, i.e., cloud-native observability, where we will learn what observability is, the components that make up observability in a cloud-native environment, and some tools such as Prometheus and Grafana.

We will also look into cost management practices in cloud-native.

Multiple choice questions

1. Which of the following are potential benefits of using Functions-as-a-Service in a serverless architecture? (Select all that apply)
 - a. Effort on code development
 - b. Scale down to zero during idle times
 - c. Faster time-to-market

- d. Effort on scaling
 - e. Both b and c
2. Which of the following is a fully managed serverless computing offering?
- a. Knative
 - b. Kubernetes
 - c. AWS Lambda
 - d. Apache OpenWhisk
3. Which of the following can be used to invoke a function? (Select all that apply)
- a. Events
 - b. Triggers
 - c. HTTP
 - d. Another function
 - e. All of the above
4. Which of the following autoscaling capabilities in Kubernetes adjusts resource requests and limits based on demand?
- a. Horizontal Pod Autoscaler
 - b. Vertical Pod Autoscaler
 - c. Cluster Autoscaler
 - d. Event Driver Autoscaler
5. Which of the following autoscaling capabilities in Kubernetes adds or removes Pods based on demand?
- a. Horizontal Pod Autoscaler
 - b. Vertical Pod Autoscaler
 - c. Cluster Autoscaler

- d. Event Driver Autoscaler
6. Which of the following autoscaling capabilities in Kubernetes adds or removes nodes based on demand?
- a. Horizontal Pod Autoscaler
 - b. Vertical Pod Autoscaler
 - c. Cluster Autoscaler
 - d. Event Driver Autoscaler
7. Serverless architectures do not use servers. True or false.
- a. False
 - b. True
8. Who in the CNCF governance approach is responsible for business oversight, marketing and budget?
- a. Technical Oversight Committee
 - b. Governing Board
 - c. Special Interest Groups
 - d. End User Community
9. What is the Technical Oversight Committee responsible for?
- a. Approve new projects
 - b. Approve budgets
 - c. Provide business oversight
 - d. Accept feedback from End User Committee and map them to projects
 - e. Both option a and d
10. Which of the following would be part of the day-to-day responsibilities of an SRE?
- a. Monitoring/Observability
 - b. Creating cloud infrastructure resources

- c. Selecting SLIs and defining SLO
 - d. Error budget
- 11. Which of the following skills are expected of a DevSecOps engineer? (Select all that apply)
 - a. Development
 - b. Operations
 - c. Security
 - d. All of the above
- 12. Which of the following cloud-native personas is responsible for creating a monitoring baseline?
 - a. Cloud engineer
 - b. DevOps engineer
 - c. Systems engineer
 - d. Site reliability engineer
- 13. Which of the following networking capabilities allows controlling traffic flows within a Kubernetes cluster?
 - a. Firewall
 - b. Network Policies
 - c. CNI
 - d. Network Security Groups
- 14. Which of the following plugins are required to implement Network Policies?
 - a. CSI
 - b. CRI
 - c. CNI
 - d. SMI

15. Select the most appropriate resource type for deploying a containerized stateless application in Kubernetes that can support scaling and rolling updates.
- a. DaemonSet
 - b. StatefulSet
 - c. ReplicaSet
 - d. Deployment

Answers

1	e.
2	c.
3	e.
4	b.
5	a.
6	c.
7	b.
8	b.
9	e.
10	b.
11	d.
12	d.
13	b.
14	c.
15	d.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Cloud-Native Observability

Introduction

Cloud-native environments are complex, dynamic, short-lived and scale on demand making traditional monitoring tools and methods unsuitable for deep visibility into the internal state of the environments. These environments need a continuous examination of the outputs from every component to ensure a resilient, reliable, and performant system. This ability is called Observability, and today, they are crucial to cloud-native systems. In this chapter, we will look at what cloud-native observability is and what tools/solutions can you use to observe them.

This chapter covers the *Cloud-Native Observability* domain of the exam, which is 8% of the exam and includes major objectives like telemetry and observability, Prometheus and cost management.

Structure

This chapter has the following topics:

- Telemetry, monitoring and observability
- Native observability in Kubernetes
- Open-source observability tools

- Cost management

Objectives

The focus of this chapter is to understand the importance of observability in cloud-native architectures and have a high-level knowledge of some of the tools that can help in implementing an observable system. We will start with an introduction to telemetry, monitoring and observability in general including understanding terms such as instrumentation, followed by how cloud-native observability is different than normal observability. Next, we will learn how you can use built-in tools in Kubernetes to do basic monitoring and logging before moving on to learn about advanced open-source observability tools and solutions like Prometheus, Grafana, Jaeger, ELK etc. At the end, we will look at cost management and FinOps concepts.

Telemetry, monitoring and observability

Telemetry and observability are terms that have become popular with microservices and cloud-native architectures. Monitoring, however, is not something new and has been an integral part of traditional IT systems for decades.

While the old approach of monitoring can apply to cloud-native applications and in fact is used as well, it is not enough to have a holistic view of the entire system. Monitoring cloud-native systems brings with them a different set of challenges, such as scale, short-lived resources, dynamic nature, distributed architectures, etc., and traditional monitoring tools and solutions are not equipped to handle them.

Let us begin by looking at what telemetry and observability are and define a few related terminologies.

Telemetry

Telemetry is data collected from all different components of a cloud-native system, which become data sources. Its primary objective is data

collection. This data can be metrics such as throughput, error rates, response times, CPU utilization, etc., or *traces* or *logs*. Telemetry data is later used in various scenarios such as troubleshooting, scaling, gaining insight into the behavior of an application and/or infrastructure such as health, performance, cost etc. thus ensuring a reliable system. It can also help in proactively identifying issues before they happen. Telemetry becomes even more important in systems that involve multiple-clouds or hybrid environments.

Let us begin by understanding what metrics, logs and traces are:

- **Metrics** are data measured over a certain time period. This data can then be aggregated or averaged based on the requirement. For example, CPU utilization percentage captured over 24 hours and averaged, or network ingress, etc.
- **Logs** are files that contain timestamped events. You would look at logs when troubleshooting an application or infrastructure issue. Log files could be in plain text, raw, or JSON among some of the formats and contain events categorized into severity levels such as debug, info, warning, critical and error. An example of this is the Apache error logs or system logs in a Windows machine.
- **Traces** are important, especially in our complex cloud-native architectures. As a developer, you may have already seen or experienced tracing in web applications. A trace is the journey of a request across different microservices or components of an application. A single trace may contain events from multiple services. Traces are used to troubleshoot errors and performance issues such as response latency, service faults, etc.

In a cloud-native platform such as Kubernetes, telemetry data can be collected using an agent based or agentless approach. Some data sources support an agentless approach while some may need agents. Another approach is application telemetry, which involves instrumenting the application code itself to collect and transmit data. Systems should also collect network telemetry that deals with data concerning network

behavior. Collecting telemetry data should be automated in a cloud-native system.

Telemetry can generate a lot of data making it difficult to make sense of it. This is where choosing the right observability tools is important. Together metrics, logs and traces form the core pillars of *observability* in distributed systems.

Instrumentation

Instrumentation is in fact the first step to building an observable system. Unless there is telemetry data generated, there is nothing to collect and analyze. Instrumentation is the process of adding code to an application that enables it to emit telemetry data. This can then be either sent (pushed) or scraped (pulled) by telemetry and monitoring tools.

For example, Kubernetes components are instrumented to emit metrics at the `/metrics` endpoint. Observability tools can collect them from this endpoint.

OpenTelemetry is an open-source framework backed by the Cloud Native Computing Foundation (CNCF), that can be used for instrumentation. Proprietary tools may provide their own set of **Software Development Kits (SDKs)** and libraries for instrumenting applications.

Monitoring

Monitoring is usually associated with *metrics*. Traditionally a monitoring tool or solution would collect these metrics, usually through an agent, and display them through dashboard. This approach is still popular and can be seen in cloud platforms as well. Monitoring aids in answering important questions surrounding system performance, service uptime, CPU and memory utilization, and helps in debugging and troubleshooting when something breaks. A simple example of monitoring is, *Is my server up?*, which can be achieved by an Internet Control Message Protocol (ICMP) ping metric.

A good way to map metrics and monitoring to telemetry and observability is to consider them as subsets. Metrics is a subset of telemetry and monitoring can be considered a subset of observability. Monitoring in general is not very detailed and usually provides only an aerial view of the system behavior. Observability on the other hand provides a detailed and internal view of the application and infrastructure along with visibility into the mapping of components (in the case of applications, its microservices) and the ability to trace the flow of requests.

Observability

Observability is the ability to measure and gain insight into a system's internal workings, current state, and more by analyzing its behavior and external outputs. It encompasses monitoring, logging, and tracing as a unit, analyzing the collected telemetry data, to provide deeper information and context on the behavior of a system. The goal is to have a holistic view of the system (application and infrastructure) and not as separate components. This also helps in correlating events or issues to identify the cause. Unlike monitoring, observability is more detailed. Observability is a mindset, like DevOps. The more you observe, the better placed you are in navigating the complexities of a cloud-native system.

Cloud-native observability

It is very evident at this point in the book that cloud-native architectures are complex microservices based designs that are highly resilient and scalable. Further, there is a dynamic nature to apps and infrastructure in cloud-native. Let us not forget the massive amount of data that is generated and processed in them. This puts a strong case for observability, on the need to collect, store and analyze data from different components, both infrastructure and applications (including data) alike. However, traditional observability or monitoring tools that are mostly based on a snapshot-based approach may not be equipped for this.

Cloud-native observability is a property applicable to cloud-native systems. It involves using observability tools to monitor, log and trace all the components of a distributed system. The more you observe, the better insight you can have into your system. Cloud-native observability tools are centralized and use an automated approach to provide continuous, detailed, and real-time visibility into environments which is necessary for addressing possible issues, preventing downtime, troubleshooting, etc. at speed. When combined with ML and AI, they can also help you predict and understand potential future issues, or events. They can zoom in on an event, including past events. An example of such a tool for metrics and monitoring is *Prometheus*. *Jaeger* is a related tool for tracing.

Cloud-native observability is also referred to as full stack observability, as it provides detailed visibility across the entire stack, that is, business, application, data, and infrastructure across areas, such as performance, health, scalability, resiliency and cost. It can also help with security. It requires tools that are purpose-built for such use cases and provide the required capabilities to collect, store, and analyze telemetry data continuously and at scale. As discussed earlier, it is also equally important for microservices to be instrumented programmatically using SDKs or using methods such as sidecar containers in Kubernetes to send high-quality telemetry data. Quality of data is crucial to make decisions.

Note: Examples of business metrics include API response times, transactions per second, error rates, etc.

The benefits of such an observability system include enhanced debugging, faster incident response times, improved system reliability, and the ability to proactively address potential issues to name a few. This eventually contributes to a more resilient and scalable system. Cloud-native observability is crucial for identifying, troubleshooting, and resolving issues in complex, distributed systems, allowing developers and operators to detect anomalies, understand system behavior, and optimize performance. This also enables continuous improvement, which is an important part of DevOps.

The steps that make up an observability implementation include instrumentation, collection, storage (backend), analysis, and visualization. We will learn about these steps in a little more detail, including open-source frameworks and tools that can help implement cloud-native observability, later in this chapter in the *open-source observability tools* section.

Native observability in Kubernetes

In this section, we will take a moment and quickly look at how to monitor Kubernetes and work with metrics, logs, events, etc., using out-of-the-box options.

This section is divided into monitoring, which covers metrics collection, logging, and tracing.

Metrics and monitoring

Metrics are primarily associated with the term monitoring. Metrics collection, aggregation and monitoring in Kubernetes has undergone changes over the years and the following information is the current state of monitoring in Kubernetes.

Kubernetes comes with some basic tools out of the box and additionally provides add-on components for specific use cases and extensibility. Monitoring in Kubernetes continues to follow the pluggable design approach.

The following monitoring capabilities come with Kubernetes:

- **cAdvisor:** It is a container metrics collection and monitoring tool from Google and is integrated with the *kubelet* binary. It collects metrics on CPU, memory, disk and network for all containers running on a node. cAdvisor does not store data for long term and the data is only available on that node.
- **Probes:** They are like health checks and are used to monitor the state of Pods (and the containers in them), and, depending on the

state, take actions. Kubernetes provides liveness, readiness and startup probes. Liveness probes can be used to restart failing containers, readiness probes can be used to ensure traffic is routed once containers are ready, as some containers may take time to start services, and startup probes can be used to verify that an application inside a container has started. Startup probes, if configured, run before liveness and readiness probes.

- **metrics-server (add on):** Metrics server is an add on component that collects and aggregates resource metrics from *kubelet* primarily for autoscaling needs, such as Horizontal Pod Autoscaler and Vertical Pod Autoscaler. It implements a first-class API, called the Metrics API, that components, such as Kubernetes scheduler and autoscaler, can use to access resource usage metrics of nodes and Pods. It also enables the **kubectl top** command to show metrics. The metrics available through metrics-server are limited to CPU and memory only. For additional metrics support, the *Custom Metrics API*, which has richer and wider support for external tooling, can be implemented.

Metrics server should not be used for forwarding metrics to external monitoring solutions.

- **kube-state-metrics (add on):** It is a service that generates metrics from Kubernetes API objects, such as Nodes, Deployments, DaemonSets, Pods, etc. These metrics are then available at the **/metrics** endpoint of the HTTP server in Prometheus format which is a structured plain text format. From this endpoint a centralized monitoring tool, such as *Prometheus*, can scrape them.

Both *metrics-server* and *kube-state-metrics* are Prometheus friendly and expose metrics in Prometheus supported format.

Managed Kubernetes installations provide their own default set of metrics, and some may even provide a vendor-specific monitoring solution and dashboard.

Logging

The *kubelet* in every node watches for log streams from containers and makes them available for consumption. You can view the logs using the **kubectl logs** command. These logs are stored locally on the node and may not be available in case of a node failure. Viewing logs using **kubectl** may be ideal for small environments or for ad hoc troubleshooting, but they are limited in capabilities for large complex environments. They are also visually challenging to understand.

Following are some commands that can be used to view logs from a terminal using **kubectl** for ad hoc requirements:

Use case	Command syntax
View logs of a Pod with a single container	kubectl logs <podname>
View logs of a specific container in a Pod (for Pods that have multiple containers)	kubectl logs -c <containername> <podname>
View logs of all containers in a Pod (Pod with multiple containers)	kubectl logs <podname> --all-containers
View 100 recent lines of logs from a Pod (single container)	kubectl logs --tail=100 <podname>
Stream logs of a Pod	kubectl logs -f <podname>
View logs of a previously failed Pod	kubectl logs --previous <podname>

Table 7.1: *kubectl logs* command examples

A central logging solution or tool such as *Prometheus* is recommended for building a highly observable system. To send logs to such tools, a log shipping method must be used. Two popular methods are as follows:

- **Node-level logging using an agent:** In this method, an agent is installed on each node of the cluster. The agent is deployed in its own Pod and there is only one agent per node. It has access to a path where all applications running on that node store logs. The *node-level logging agent* method does not require any application changes to be made.
- **Container/Application logging using sidecar:** This logging method uses a sidecar container to ship logs. It runs in a container in the Pod that has the application (in a separate container). This solution may require instrumenting the application code.

Events

Out of the box, Kubernetes provides events that can be used to monitor the state of resources in a Kubernetes cluster. These events are generated when one of the following happens:

- State change, for example a Pod is created, or a Pod failed at creation, changes in status of a Pod from pending to running, etc. State change events can be used to understand why a Pod failed. Events for state change are captured for nodes, deployments, Pods, etc.
- Scheduling, for example scheduling was successful or scheduling failed due to unsuitable nodes, etc.
- Configuration change, such as horizontal Pod autoscaling.

The events can be viewed with the command **kubectl get events**. The **kubectl describe <resource>** command can also be used to view the events of that specific resource.

These events do not persist forever and are available for only one hour after the event is generated. You must use a separate centralized monitoring solution to ship them and analyze them if needed. For example, you can use *Grafana* to get insights from these events.

Open-source observability tools

While Kubernetes provides tools and add-ons for monitoring metrics and viewing logs, they are not centralized and stored for long term availability. Additionally, to achieve a highly observable system that provides detailed information and context, we should be able to analyze them together through querying or other methods. This is where full blown observability solutions that extend cloud-native platforms, such as Kubernetes, sell themselves.

Let us look at some of the open-source solutions spread across monitoring, logging and tracing. For the exam, awareness of *Prometheus* and *Jaeger* is enough, however this section also introduces other popular observability solutions.

The following table shows the mapping of these tools to their functionality:

Function	Tool
Metrics and monitoring	Prometheus, Thanos
Logging	Jaeger, ELK, Fluentd
Tracing	OpenTelemetry
Instrumentation	OpenTelemetry
Visualization	Grafana, Kibana

Table 7.2: Tools to functionality mapping

Prometheus

Prometheus is an open-source monitoring solution that is quite popular and common to have in production Kubernetes deployments. It provides powerful metrics, alerting capabilities and more for cloud-native environments. Prometheus is also a CNCF graduated project.

Behind the scenes, Prometheus uses a time-series database to store the metrics in plain text format and acts as the backend for your telemetry, which we discussed in the previous section.

Prometheus provides a functional query language called PromQL for real-time selection and aggregation of time series data using expressions. The results of these expressions can be viewed in tabular and graph format in the expression browser, or consumed by external systems through the HTTP API. It is common to use *Grafana*, an open-source analytics and visualization tool, along with Prometheus for creating dashboards and visualization.

For exam purposes, it is not required to have a deep knowledge of Prometheus. In fact, there is a separate certification that tests a candidate's knowledge of Prometheus—Prometheus Certified Associate. However, the following is a high-level overview of how it works:

1. Step one is to begin by installing Prometheus and its components. It is commonly deployed into a Kubernetes cluster using Helm, mostly the one it is going to monitor. It can also be installed on a Linux server directly or run directly as a Docker container.
2. Step two is making metrics available for Prometheus to scrape. Some components may already be emitting metrics that are ready to scrape, such as the Kubernetes metrics at the `/metrics` endpoint (provided the `kube-state-metrics` is configured) and some will require instrumentation to publish metrics for scraping.
3. Step three is to collect and analyze metrics.

Finally, you will always need a way to view the stored time-series data in Prometheus and often create dashboards and visualization out of it for consumption. You can have the following few options:

- **Prometheus Expression Browser:** It can be used for quick visual representations. You write the metric name in the expression field and click **Execute** to display the results which can be in a table format, referred to as console, or as a graph. The Prometheus

expression browser web console is available at the `/graph` endpoint. The following figure shows an example of the Prometheus web console in graph view:

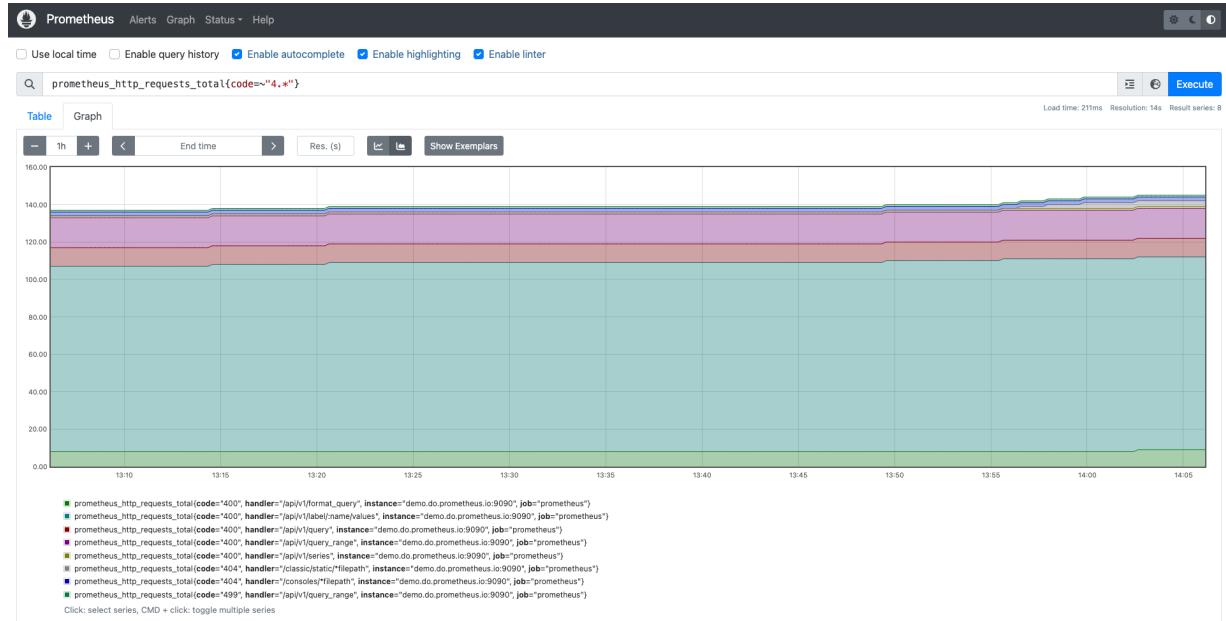


Figure 7.1: Prometheus web console view in graph mode

- Grafana, covered in slightly more detail in the next section, is the preferred choice for analytics, visualization, dashboards and more. It comes packaged into the Prometheus installation.

Prometheus metric types

While not important for an exam, here is a quick list of metric types supported by the Prometheus client libraries:

- **Counter:** A counter is a metric whose value can only increase or be reset to zero on restart. For example, the number of **4xx** errors.
- **Gauge:** A gauge is a metric whose value can go up and down. For example, the number of messages in a queue.
- **Histogram:** A histogram samples observations such as the duration of a request or the response size and counts them in configurable buckets. It also provides a sum of all observed values. Latency is a good example of a histogram metric type.

- **Summary:** Like the histogram, a summary samples observations as well and provides a total count of observations and a sum of all observed values. It can also calculate configurable quantiles over a sliding time window.

Grafana

Grafana is primarily a multi-platform open-source visualization tool that provides insights into your metrics data with additional capabilities, such as alerting, querying, etc. It helps you create visually appealing dashboards and share them with others.

Among other capabilities, it supports transformations on queries and data sources, annotations, a panel editor and plugins to connect data sources, tools and teams to Grafana.

There are many official and community dashboards to get started with. Grafana can work with popular time series databases, like InfluxDB and Graphite.

The following figure shows an example of a Grafana dashboard with default metrics:

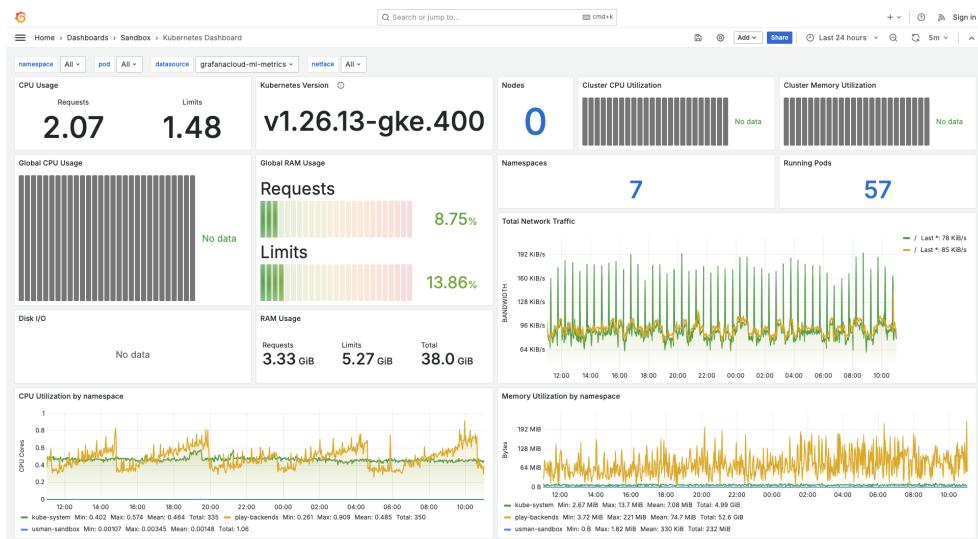


Figure 7.2: Example of the Grafana dashboard

Grafana is part of Grafana Labs that has other popular open-source products, such as Loki, Tempo, Mimir, Faro, and Pyroscope. They also have a SaaS offering called Grafana Cloud.

Jaeger

Jaeger is an open-source distributed tracing platform for storing and querying tracing data. It was created by Uber (the cab hailing company), inspired by Dapper (from Google) and OpenZipkin, and is a CNCF graduated project.

It provides capabilities such high scalability, supports data exported by OpenTelemetry, supports multiple storage backends, has a modern web UI, etc. In Kubernetes, Jaeger is deployed as a Kubernetes Operator.

The use cases of Jaeger include the following:

- Monitoring distributed workflows
- Finding and fixing performance bottlenecks
- Root cause analysis
- Analyzing service dependencies

Elasticsearch, Logstash and Kibana

ELK is a log analytics solution that is built using the following three tools:

- **Elasticsearch:** It is a real-time and distributed search engine for logs. It indexes logs as JSON documents which are stored in a distributed object-storage. The engine also provides search and analytics capabilities.
- **Logstash:** It is used as a data ingestion tool. It collects, transforms and ships the logs to Elasticsearch.
- **Kibana:** It is used for visualization of the indexed Elasticsearch data. It supports features, such as time-series analysis, location analysis, machine learning capabilities and more.

Fluentd

Fluentd is a log collector that can collect logs from various sources and send them to a centralized log management tool or storage sink.

Thanos

Thanos is a highly available implementation of *Prometheus* with additional features, such as long-term storage and retention, data deduplication, horizontal scalability, federation across multiple Prometheus instances to name a few. It is a CNCF incubating project.

OpenTelemetry

OpenTelemetry is an open-source, vendor neutral observability framework and provides a collection of APIs, SDKs, tools and protocols that can be used to instrument, generate, collect, and export telemetry data (metrics, logs, and traces). It is a CNCF project in the incubating stage. It was created from a merger of the *OpenTracing* and *OpenCensus* projects.

OpenTelemetry is tool agnostic and hence can work with many popular open source backends, such as Prometheus, Jaeger, etc. It integrates with a broad set of programming libraries and frameworks making it easy to instrument applications running anywhere. It is not an observability backend, meaning the storage and visualization is not the task of OpenTelemetry.

Cost management

Given the nature of complexity and heterogeneous nature of today's cloud-native applications, which consist of many microservices and infrastructure spanning multiple clouds, on-premises, etc., it becomes important to keep an eye on costs.

One of the most important areas that contribute to costs is sizing. It is common for teams to overprovision resources, at least during the initial stages of deployment, mostly for performance reasons as there is a lack of

insight into the utilization metrics. For example, in the context of Kubernetes and cloud-native, you may set the *resource requests and limits* value for containers in Pods higher than the benchmarked numbers. Often, these values are not modified or optimized based on actual usage, unless there is a Pod in *pending* state.

Now, some may argue that the preceding problem can be solved by autoscaling features, such as cluster autoscaler. However, it can still result in increased cost, especially if the existing resources are underutilized and new workloads could have been fit into the existing cluster size.

The bottomline is that unless there is an issue with resource provisioning, due to lack of resources such as Pod in *pending* state, or unless there is a huge bill at the end of the billing cycle, no one is looking into the cost and the preceding resource size values remain unchanged.

Observability plays an important part in enabling teams and stakeholders to gain visibility. This is done by monitoring metrics that can help with rightsizing, identifying unused resources and more.

Some of the best practices that can help with cost management include the following:

- Using labels in Kubernetes resources. This can help visualize costs based on department, environment type, customer, etc.
- Implementing observability into applications, services and infrastructure. This can again help visualize and find resources that are underutilized, idle, etc. and make necessary changes.
- Features like cluster autoscaler, combined with rightsizing, can help save costs by scaling up in case of demand or in specific times and scaling down during lower demand or zero job scenarios.
- To curb waste of resources, especially in scenarios where a workload is running but not doing anything, use *serverless* approach. Serverless allows scaling down to zero when there is nothing to process, making a huge impact on the cost.

- Control technical debt by optimizing the application architecture to reduce the number of Pods.

Finally, you need tools to translate metrics into cost. For managed Kubernetes, you need platforms from cloud vendors, such as Azure, AWS and GCP. This is provided by vendors through their native cost management services.

For self-hosted Kubernetes or for scenarios where you want your own cost management solution, tools like *KubeCost* can help.

FinOps

A term used for cost management in the cloud-native world is FinOps although they are not exactly the same.

The definition of FinOps from the *FinOps Foundation Technical Advisory Council* is as follows:

FinOps is an operational framework and cultural practice which maximizes the business value of cloud, enables timely data-driven decision making, and creates financial accountability through collaboration between engineering, finance, and business teams.

Just like DevOps, FinOps is a culture where cost management is everyone's responsibility. All cross-functional teams in a company work together to deliver a product in a predictable manner, faster while managing costs. Cost management should not be a one-time task, or an afterthought but a continuous process.

Conclusion

Observability is an important and must have feature in a cloud-native architecture as it provides detailed and actionable insights into the internal workings of the system. Metrics, logs and traces provide the necessary telemetry data to analyze for building an observable system. Open-source observability tools are used to instrument, collect, store,

analyze and visualize these data. Observability also enables effective cost management and FinOps.

In the next chapter, *Cloud-Native Application Delivery*, we will learn about application development and delivery processes, such as CI/CD, and look into the tools that can help implement these processes.

Multiple choice questions

1. Which of the following telemetry data would you need if you wanted to measure the network ingress bandwidth?
 - a. Metrics
 - b. Logs
 - c. Traces
 - d. Errors
2. What is a term used to describe the end-to-end tracking of a request as it flows through a distributed system?
 - a. Log
 - b. Span
 - c. API
 - d. Trace
3. Which of the following is the most popular log format in cloud and cloud-native platforms?
 - a. JSON
 - b. RAW
 - c. Text
 - d. CSV
4. Select the correct Kubernetes command to view logs of a Pod named kcnaprep. The Pod has only one container.
 - a. kubectl logs pod kcnaprep

- b. kubectl logs kcnaprep
 - c. kubelet logs kcnaprep
 - d. kubectl pod logs kcnaprep
5. Select the correct command to view logs of one of the containers, called `kcnaprep-sidecar`, in a Pod named `kcnaprep`. The Pod has multiple containers.
- a. kubectl logs kcna-prep
 - b. kubectl logs kcna
 - c. kubectl logs -c kcnaprep-sidecar kcna
 - d. kubectl logs -f kcna
6. What is the default endpoint where Prometheus tries to pull metrics from?
- a. /health
 - b. /prometheus
 - c. /
 - d. /metrics
7. Which of the following specific database type is used to store metrics in Prometheus?
- a. NoSQL
 - b. Object store
 - c. Timeseries
 - d. SQL
8. You plan to create a dashboard with meaningful visualization for external stakeholders to view a set of chosen performance metrics. Which of the following tools can be used to create it with minimal effort?
- a. PowerBI

- b. Build one with Go programming language
 - c. Grafana
 - d. Prometheus Expression Browser
- 9. As a developer, you have been asked to instrument the applications so that detailed telemetry data can be collected from the applications. Which of the following frameworks or tools could you use to do so?
 - a. OpenTracing
 - b. OpenCensus
 - c. OpenTelemetry
 - d. OpenMetrics
- 10. Which of the following are benefits of cloud-native observability?
 - a. Faster incident response
 - b. Improved reliability
 - c. Detect potential issues
 - d. Efficient resource utilization
- 11. Which of the following is an add on components that generates metrics from Kubernetes API objects and makes them available at the /metrics endpoint?
 - a. cAdvisor
 - b. Probes
 - c. metrics-server
 - d. kube-state-metrics
- 12. Node-level logging method, which uses an agent, does not require application changes.
 - a. True

- b. False
13. Which of the following commands can be used to view events related to the state change of a Pod called kcnaprep?
- kubectl logs kcnaprep
 - kubectl get pod kcnaprep
 - kubectl events
 - kubectl describe pod kcnaprep
14. What kind of a tool is Jaeger?
- Monitoring
 - Logging
 - Tracing
 - Visualization
15. Choose an architecture that can help save costs by scaling down to zero when there's nothing to process.
- Cloud-native architecture
 - Serverless architecture
 - Microservices architecture
 - Monolithic architecture

Answers

1	a.
2	d.
3	a.
4	b.
5	c.
6	d.
7	c.
8	c.

9	c.
10	a. b. c. d.
11	d.
12	a.
13	c. d.
14	c.
15	b.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Cloud-Native Application Delivery

Introduction

In this last theoretical chapter, we address concepts and tools that enable cloud-native application delivery. The chapter covers GitOps, which is a Git based approach to automating infrastructure and application deployment, and continuous integration and continuous delivery/deployment (CI/CD), which emphasizes automation of different stages of application development and deployment. The chapter also introduces tools that help implement GitOps and CI/CD.

This chapter covers the last domain of the KCNA exam, that is, *cloud-native application delivery*, which makes up 8% of the exam and includes the following core objectives:

- Application delivery fundamentals
- GitOps
- CI/CD

Structure

This chapter has the following topics:

- Cloud-native application delivery
- CI/CD
- GitOps
- GitOps tools

Objectives

The objective of this chapter is to discuss how a cloud-native application delivery model works and highlight some of the tools that can be used for it. The chapter will introduce you to continuous integration, continuous delivery, and continuous deployment concepts that are at the core of cloud-native application delivery, along with GitOps concepts, its workflow, Infrastructure-as-Code, etc. The chapter also covers GitOps tools such as Flux and ArgoCD.

Cloud-native application delivery

A cloud-native application delivery model implements a pipeline-based architecture to continuously deliver features and updates to a cloud-native application, primarily by automating various stages of an application delivery lifecycle. These stages include writing code, building, testing, packaging, deploying, etc. Advanced or fully automated cloud-native application delivery model may also include managing and operating the application in their pipeline. Apart from automation, these pipelines are auditable and traceable as they use a version control system.

A CI/CD pipeline, which is the backbone of DevOps or DevSecOps principles and practices, along with GitOps are some of the core components used to implement a cloud-native application delivery model.

We will begin with CI/CD first and then go onto GitOps as it acts as an extension to CI/CD.

CI/CD

CI/CD is short for the combination of continuous integration and continuous delivery/deployment practices that enable frequent and small changes to be repeatedly delivered with speed and accuracy in an automated way.

Continuous integration

Continuous integration is a development practice, part of the bigger DevOps practices, in which application build and testing steps are automated for frequent, repeatable and error-free code integration. A centralized and shared version control system is a key component of CI, apart from automated build and test stages. CIs that follow DevSecOps practices include security tests as part of the pipeline too.

A typical workflow in CI involves the following steps:

1. Developer(s) work on an issue and make changes to the code. They usually do this in a separate branch.
2. Once they are satisfied with the change, they commit the changes to the version control system like GitHub.
3. On commit, an automated build process is triggered, which compiles the code and generates artifacts.
4. The next automated step in CI is to perform tests on the generated artifacts, such as unit tests, integration tests, etc.
5. The results of the tests are immediately available to the developers to address.
6. If both the build and tests are successful or within accepted parameters, the changed code is integrated into the main branch of the code repository. This may be automated or controlled through a gated approach:

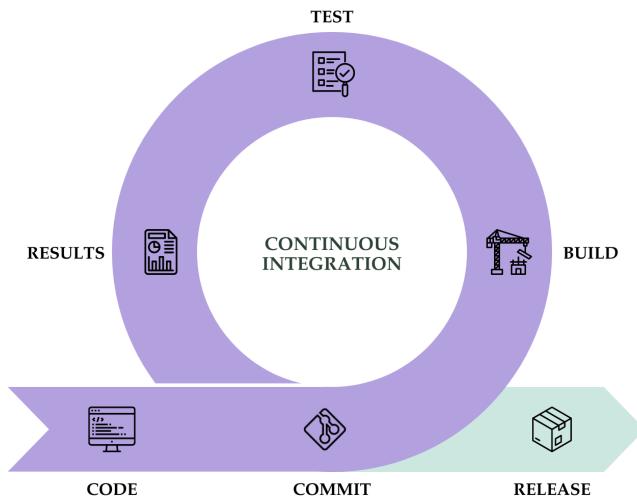


Figure 8.1: Continuous integration

Continuous delivery

Continuous delivery practice ensures that artifacts generated through the CI process are ready to be deployed to production when approved **manually**. When using a staging or test environment before production, continuous delivery can deploy the changes to it for review. However, changes to production are not deployed automatically.

The following figure illustrates a typical continuous delivery pipeline, which includes one or more acceptance tests, deployment to a staging environment and finally, controlled deployment to production:

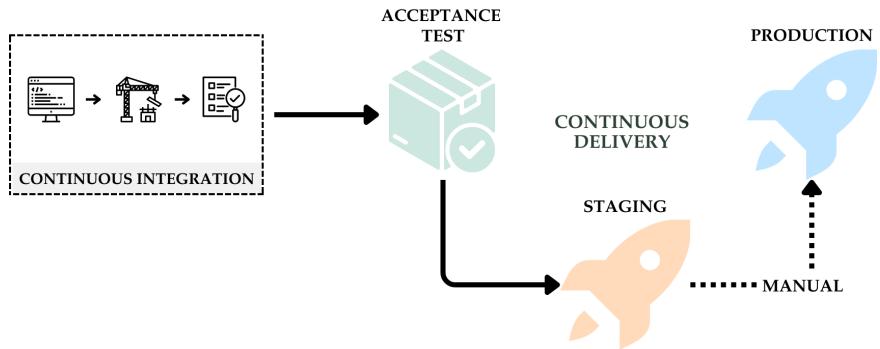


Figure 8.2: Continuous delivery

Continuous deployment

Continuous deployment is the practice of automatically deploying to production if all stages have passed in a production pipeline. This is a fully automated approach, and changes are pushed frequently. If a staging or test environment is used, the changes are deployed to it first, and automated acceptance tests are run against it. If all tests pass, the code is deployed to production automatically.

Since this enables rapid and frequent changes, there will be concerns of increased risk of bugs, vulnerabilities, and other issues such as performance. This can be alleviated by using automated testing methods like running pre-defined tests such as unit tests, integration tests, functional tests, regression tests, etc., at various stages, starting from the development phase so that the issues are caught earlier. You can also include automated load test steps in the pipeline to check for performance issues. Similarly, for security, automated security testing steps, including but not limited to static analysis, dependency checks, etc., must be part of the pipeline. Finally, monitoring will provide real-time visibility into the application's behavior in production, enabling teams to detect, respond to, and rollback or resolve issues swiftly.

The following figure illustrates a high-level continuous deployment pipeline which is similar to a continuous delivery pipeline but with automatic deployment to production:

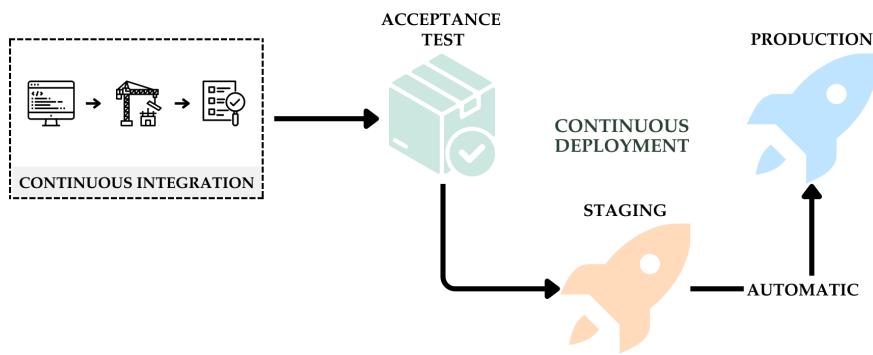


Figure 8.3: Continuous deployment

GitOps

GitOps is a modern approach to continuous delivery where operations are managed through Git, hence the term GitOps. It follows the principle of using Git as the single source of truth for both application code and infrastructure configurations. In GitOps, all changes to the system, that is, *operations* are driven through version-controlled Git repositories, where infrastructure configurations, application code, and deployment manifests are stored. Automated workflows triggered by Git events ensure that the desired state of the system, defined in Git repositories, is continuously applied to the target environment. This approach promotes transparency, repeatability, and collaboration, enabling teams to manage infrastructure and application deployments declaratively and efficiently. By leveraging Git as the central control plane, GitOps simplifies operational complexity, enhances auditability, and facilitates rapid iteration and experimentation in cloud-native environments.

Infrastructure as Code

Infrastructure as Code (IaC) is an important principle in GitOps. Let us look into it quickly first before learning more about GitOps.

IaC takes the approach of defining infrastructure using code. In IaC, you use code to create a blueprint of your infrastructure and then apply or execute that code to create the infrastructure.

For GitOps to work for infrastructure operations, it is important to define the required infrastructure resources and configurations as code, for example, a *YAML* file, which we are already familiar with in Kubernetes, or a *.tf* file popularly used with Terraform. This code is then stored in a version controller Git repository.

Some popular examples of IaC tools are:

- Terraform
- Ansible
- Azure ARM templates
- AWS CloudFormation
- Google Cloud Deployment Manager

In the context of Kubernetes, IaC can be used to create Kubernetes clusters, and it can also be used to create Kubernetes resources such as Pods, Services, etc.

The following is a simple example of a terraform file that creates a Pod:

```
provider "kubernetes" {  
    config_path = "~./kube/config"  
}  
  
resource "kubernetes_pod" "kcnaprep" {  
    metadata {  
        name = "kcnaprep"  
        labels = {  
            app = "kcnaprep"  
        }  
    }  
}
```

```
    app = "kcnapprep"
  }
}

spec {
  container {
    image = "kcnapprep:1.0"
    name = "kcnapprep"
    port {
      container_port = 80
    }
  }
}

}
```

Note: Knowledge of Terraform or how to write Terraform code is not required for the exam.

GitOps concepts

GitOps is primarily used for infrastructure and (application) configuration automation in Kubernetes. While GitOps can work on platforms other than Kubernetes, it works best with Kubernetes. For the purposes of this book and the exam, we will focus on using GitOps for the continuous delivery to Kubernetes.

GitOps is enabled by Kubernetes capabilities such as control loop, reconciliation and operators. GitOps tools are deployed as *operators* which create a custom resource (CRD) as an extension of the Kubernetes API. This is similar to standard resource controllers in Kubernetes that use control loops to monitor resources and use reconciliation to sync the

desired state to the current state (think *Deployments*, *ReplicaSets*, etc.). The difference being that in a non-GitOps approach, you apply the YAML file directly (**kubectl**) while in a GitOps approach, the YAML file is stored in Git, and the changes are applied automatically on the trigger.

There are two ways GitOps can be implemented, that is, *push* and *pull*. A push-based approach uses the standard CI/CD pipeline to push changes to the Kubernetes environment. It is unidirectional, from the Git repository to the cluster. They are simpler to implement initially and better suited for small environments. A push style GitOps will require storing or using a mechanism to provide cluster credentials in the CI/CD. This may bring security concerns and additional efforts like sharing credentials with developers or contractors.

Pull-based GitOps uses an agent running in the Kubernetes cluster that monitors changes happening in a Git repo and pulls changes when there is one to match the current state in the cluster. It works bi-directionally. Pull GitOps is common in cloud-native and Kubernetes environments as they are more consistent, better integrated with the cluster to be managed, and provide better control and governance. Since pull-style GitOps have agents running inside the cluster that pull changes, there is no need to store or share cluster credentials.

GitOps can be used for the following:

- IaC operations like deploying cloud resources, creating Kubernetes clusters, etc.
- Application configuration operations through *ConfigMaps*, *Secrets*, etc.
- Release strategies, such as blue-green, canary and rolling updates.
- Kubernetes operations, such as creating **role-based access control (RBAC)** policies, network policies, and resources such as ingress and more.
- Detecting, and if required correcting, drift in infrastructure and configuration changes.

GitOps workflow

On a high-level, a GitOps workflow has the following stages:

1. You write your desired state as code using a *declarative* approach. The desired state includes everything from infrastructure resources required for application deployment and resource configuration to environment variables and policies. Use Kubernetes YAML files, terraform files, etc., to implement the principle of IaC to do this. You can also template your applications using *Helm* or *Kustomize*.
2. You then use version control to store the code. This is where you follow Git practices and use tools such as GitHub to do this.
3. Next, you need a GitOps tool to implement the continuous delivery of the desired application state. On a higher level, the GitOps tool has a controller that continuously compares the desired state in the repo with the current state in Kubernetes. This is called **reconciliation**.
4. You propose changes through a *pull request* and merge the changes, mostly to the main branch, on approval. This merge is captured by the GitOps agent responsible for reconciliation for any actions. If the current state deviates from the new desired state, the GitOps tool can sync it. A continuous delivery model automatically syncs the states; however, this can be done manually as well.

The following figure shows a typical GitOps workflow in Kubernetes:

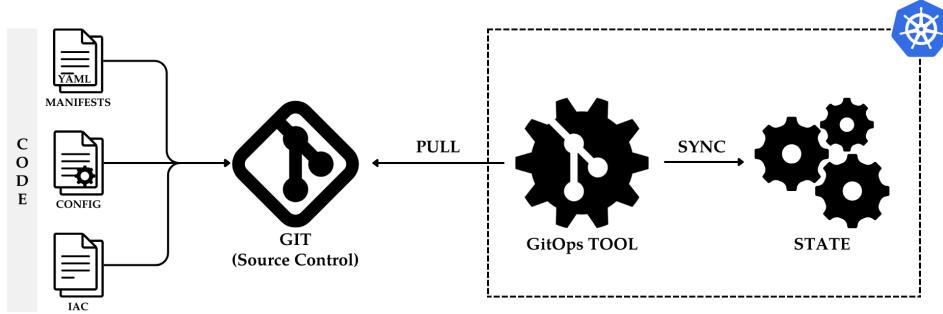


Figure 8.4: GitOps workflow

GitOps tools may support additional features and capabilities beyond the standard workflow described previously. Although, it is not in the scope of this book to dive deep into them.

GitOps tools

GitOps tools are used to implement the principle of continuous reconciliation. Here are two of the most popular and widely used GitOps tools with Kubernetes. Both Flux (or FluxCD) and ArgoCD are *pull*-style GitOps tools.

Flux

Flux is a continuous delivery solution for Kubernetes. It is constructed with the *GitOps toolkit* components consisting of the following:

- Specialized tools and Flux controllers
- Composable APIs
- Reusable Go packages for GitOps

Flux also supports progressive delivery through *Flagger*, a specialized controller that implements gradual rollout techniques such as feature flags, canary releases, A/B testing, etc. Gradually rolling out changes to a

subset of users is also called as progressive delivery, this helps in testing changes in a more controlled manner.

At its core, Flux is a GitOps tool for apps and infrastructure resources in Kubernetes with built-in dependency management, uses a declarative approach to defining the desired state and synchronizes changes automatically. Changes are controlled through pull requests, hence providing recoverability and auditability. Another notable feature is its multi-tenancy capability, supporting multiple Git repositories and the ability to manage multiple Kubernetes clusters.

Flux uses Kubernetes' API extension system and integrates well with the Kubernetes ecosystems of tools. It supports Helm, Prometheus, Grafana, Istio and more.

Flux, written in Go, is a Cloud Native Computing Foundation (CNCF) graduated project.

ArgoCD

ArgoCD is also a declarative continuous delivery tool for Kubernetes and is popular for an easy-to-use UI. Like Flux, it allows you to manage application definitions, configurations, and environments in a declarative and version-controlled approach. Application deployment and lifecycle management is automated and auditable.

ArgoCD, part of the Argo open-source tools, is also written in Go. Argo is a graduated CNCF project. Other tools from the Argo project include the following:

- **Argo Workflows:** A Kubernetes-native workflow engine supporting Directed-Acyclic Graph (DAG) and step-based workflows.
- **Argo Rollouts:** A tool that enables deployment strategies, such as canary and blue-green deployments.
- **Argo events:** An event-based dependency management for Kubernetes.

Conclusion

Cloud-native application delivery uses practices such as GitOps and CI/CD to automate various stages of the application delivery lifecycle, deploying them from development to production continuously. You can choose between a manual or automated delivery to production based on need. GitOps tools such as *Flux* and *ArgoCD* integrate well with Kubernetes, providing automated infrastructure (Kubernetes clusters), resources (Pods, deployments, etc.), and application deployments, enabling use cases such as rolling updates, canary releases, blue/green deployments, managing drifts in configurations, etc. IaC helps you define your infrastructure and resources as code, such as YAML, enabling version-controlled, automated, and consistent infrastructure management. *Ansible* and *Terraform* are popular examples of tools used for IaC.

Through this chapter, you now have an understanding of what GitOps and CI/CD are, and what tools can help you implement them.

In the next chapter, we will put into action some of the learnings of the Kubernetes chapters through a series of hands-on exercises.

Multiple choice questions

1. What is the full form of CI/CD?
 - a. Continuous involvement/continuous delivery
 - b. Continuous integration/continuous deployment
 - c. Continuous integration/continuous development
 - d. Cloud integration/cloud deployment

2. What is the full form of IaC?
 - a. Integration as Code
 - b. Information as Code
 - c. Infrastructure as Code
 - d. Innovation as Code

3. Which of the following steps are associated with CI in the CI/CD process? (Select all that apply.)

 - a. Build
 - b. Test
 - c. Deploy
 - d. Monitor
4. Which one of the following CD tools is constructed using a set of GitOps toolkit components?

 - a. Flux
 - b. Jenkins
 - c. AWS CodeDeploy
 - d. Azure DevOps
5. Which component of Flux supports progressive delivery strategies?

 - a. ArgoCD
 - b. Helm
 - c. Kustomize
 - d. Flagger
6. Your organization is looking to implement GitOps. They do not want to share the production cluster credentials to developers or use them in a CI/CD pipeline. You recommend using the GitOps push type approach.

Does this address the organization's requirement?

 - a. No
 - b. Yes
7. Your organization is looking to implement GitOps. They do not want to share the production cluster credentials to

developers or use them in a CI/CD pipeline. You recommend using the GitOps pull type approach.

Does this address the organization's requirement?

- a. Yes
 - b. No
8. Which of the following are characteristics of a cloud-native application delivery model?
- a. Continuous delivery
 - b. Automation
 - c. Version control
 - d. Everything as code
9. In a continuous deployment practice, changes are automatically deployed to production on passing all acceptance tests.
- a. True
 - b. False
10. What is the process of syncing the current state of the cluster to the desired state otherwise called as?
- a. Restoration
 - b. Resurrection
 - c. Reconciliation
 - d. Remediation

Answers

1	b.
2	c.
3	a. b.
4	a.

5	d.
6	a.
7	a.
8	a. b. c. d.
9	a.
10	c.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 9

Hands-on Kubernetes

Introduction

This chapter provides a hands-on experience of running containerized applications and workloads using Kubernetes. The chapter includes demonstrations on Kubernetes installation using minikube, deploying Kubernetes resources such as Pods, Deployments, Services, Persistent Volumes, Persistent Volume Claims, StatefulSets and DaemonSets. It also has exercises covering common **kubectl** commands for monitoring and troubleshooting in day-to-day operations.

Hands-on knowledge of Kubernetes installation, along with the knowledge of Kubernetes (**kubectl**) commands line empowers developers and operators to efficiently manage and troubleshoot containerized applications. It also enables them to deploy, scale, and monitor workloads directly, ensuring quick responses to operational issues and sometimes making real-time adjustments.

While a practical knowledge of Kubernetes, especially production Kubernetes, is not required for the KCNA exam, it is expected that test takers are aware of basic **kubectl** commands and may expect questions on them in the exam too.

Note: Do not copy and paste the commands from the book (especially when using the e-book format) as this may break the command, for example it may add additional spaces or not run a multi-line command properly.

Type the commands, which is also a better way to learn and helps with retention. Additionally, the GitHub repository accompanying the book will have the commands.

Structure

This chapter covers the following exercises followed by a quiz at the end.

- Creating a Kubernetes cluster using minikube
- Creating a Kubernetes cluster using Play with Kubernetes
- Understanding the environment
- Creating resources in Kubernetes
- Basic operations and troubleshooting
- Persistent storage in Kubernetes
- StatefulSets and DaemonSets

Objectives

The objective of this hands-on chapter is to bridge the gap between the theoretical understanding of Kubernetes concepts from previous chapters and the practical management of resources in Kubernetes. It provides a series of exercises that will help you get started with Kubernetes and its commands. The chapter will demonstrate how to prepare a Linux machine and create a Kubernetes cluster using minikube. It will then take you through various **kubectl** commands that help you create and manage resources in Kubernetes such as Pods, Deployments, Services, Persistent Volumes, Persistent Volume Claims, StatefulSets and DaemonSets. It also has exercises covering common kubectl commands for monitoring and troubleshooting in day-to-day operations.

Creating a Kubernetes cluster using minikube

As discussed in [Chapter 4: Kubernetes Basics](#), there are various tools available to create a Kubernetes cluster for local development and testing. This exercise demonstrates minikube.

The following instructions show how to install minikube on a Linux machine. For instructions on how to install it on Windows or MacOS, visit <https://minikube.sigs.k8s.io/docs/start/> and select your target operating system. You will also be able to choose the architecture type, release type, and installer type.

Here is a screenshot of the installation step from the Get Started! page:

The screenshot shows the configuration options for installing minikube. It includes dropdown menus for Operating system (Linux selected), Architecture (x86-64 selected), Release type (Stable selected), and Installer type (.exe download selected). Below these, instructions are provided for installing the stable release on x86-64 Windows using .exe download. It includes PowerShell commands for creating a directory, downloading the executable, and setting the PATH environment variable. A note at the bottom states that if used a terminal like powershell, the terminal needs to be closed and reopened before running minikube.

Click on the buttons that describe your target platform. For other architectures, see [the release page](#) for a complete list of minikube binaries.

Operating system Linux macOS Windows

Architecture x86-64 arm64 arm

Release type Stable Beta

Installer type .exe download Windows Package Manager Chocolatey

To install the latest minikube **stable** release on **x86-64 Windows** using **.exe download**:

1. Download and run the installer for the [latest release](#).
Or if using `PowerShell`, use this command:

```
New-Item -Path 'c:\' -Name 'minikube' -ItemType Directory -Force  
Invoke-WebRequest -OutFile 'c:\minikube\minikube.exe' -Uri 'https://github.com/kubernetes/minikube/releases/latest/download/minikube.exe'
```
2. Add the `minikube.exe` binary to your `PATH`.
Make sure to run PowerShell as Administrator.

```
$oldPath = [Environment]::GetEnvironmentVariable('Path', [EnvironmentVariableTarget]::Machine)  
if ($oldPath.Split(';') -inotcontains 'C:\minikube'){  
    [Environment]::SetEnvironmentVariable('Path', ${'{}';$oldPath}; $oldPath, [EnvironmentVariableTarget]::Machine)  
}
```

If you used a terminal (like powershell) for the installation, please close the terminal and reopen it before running minikube.

Figure 9.1: minikube install methods

Prerequisites

The following are the bare minimum prerequisites for setting up a local minikube playground:

- A Linux machine (desktop, VM, VPS, etc.) with a newer OS.

- An Ubuntu 22.04 machine has been used here for demonstrations.
- Experience working with command line.
- Skip to the next section for instructions.

Note: If you are unable to install minikube due to any reason, this chapter also provides instructions on creating a Kubernetes cluster using a free browser-based Kubernetes playground, Play with Kubernetes.

Install Docker

A container or virtual machine manager is a prerequisite for minikube. For this hands-on, we will use Docker-CE, where CE stands for **Community Edition**. The following steps install *Docker-CE* using an official script:

1. Use **curl** to download the install script to a folder of your choice:

```
curl -fsSL https://get.docker.com -o get-docker.sh
```

2. Next, execute the script using the following command to install Docker:

```
sudo sh get-docker.sh
```

On the successful execution of the script, Docker should be installed and running. You should see an output similar to the following screenshot:

```
Client: Docker Engine - Community
 Version:          25.0.0
 API version:     1.44
 Go version:      go1.21.6
 Git commit:       e758fe5
 Built:            Thu Jan 18 17:09:49 2024
 OS/Arch:          linux/amd64
 Context:          default

Server: Docker Engine - Community
Engine:
 Version:          25.0.0
 API version:     1.44 (minimum version 1.24)
 Go version:      go1.21.6
 Git commit:       615dfdf
 Built:            Thu Jan 18 17:09:49 2024
 OS/Arch:          linux/amd64
 Experimental:    false
containerd:
 Version:          1.6.27
 GitCommit:        a1496014c916f9e62104b33d1bb5bd03b0858e59
runc:
 Version:          1.1.11
 GitCommit:        v1.1.11-0-g4bccb38
docker-init:
 Version:          0.19.0
 GitCommit:        de40ad0
```

To run Docker as a non-privileged user, consider setting up the Docker daemon in rootless mode for your user:

```
dockerd-rootless-setuptool.sh install
```

Visit <https://docs.docker.com/go/rootless/> to learn about rootless mode.

To run the Docker daemon as a fully privileged service, but granting non-root users access, refer to <https://docs.docker.com/go/daemon-access/>

WARNING: Access to the remote API on a privileged Docker daemon is equivalent to root access on the host. Refer to the 'Docker daemon attack surface' documentation for details: <https://docs.docker.com/go/attack-surface/>

Figure 9.2: Output of docker install script

3. Add the current user to the docker group to avoid running **docker** commands with **sudo** every time. Run the following command to do so:

```
sudo usermod -aG docker $USER
```

4. Verify that you can run **docker** commands without **sudo** using the following command:

```
docker -version
```

Note: You may have to logout and login again for the changes to take effect

Installing minikube

We are now ready to install minikube. Refer to the following steps in sequence to do so:

1. Begin by downloading the latest version of the minikube binary using curl:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

2. Install the binary to the local bin folder through the following command:

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

3. Next, start a two node Kubernetes cluster using the **minikube start** command which is as follows:

```
minikube start --nodes=2 -mount
```

The output is as follows:

```

student@kcnaprep:~$ minikube start --nodes=2
🕒 minikube v1.32.0 on Ubuntu 22.04
    ✨ Automatically selected the docker driver
    🛠 Using Docker driver with root privileges
    🤖 Starting control plane node minikube in cluster minikube
    ⚡ Pulling base image ...
    🎨 Creating docker container (CPUs=2, Memory=2200MB) ...
    🐳 Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
        ■ Generating certificates and keys ...
        ■ Booting up control plane ...
        ■ Configuring RBAC rules ...
    🌐 Configuring CNI (Container Networking Interface) ...
        ■ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    🏷️ Verifying Kubernetes components...
    🌟 Enabled addons: storage-provisioner, default-storageclass

    🤖 Starting worker node minikube-m02 in cluster minikube
    ⚡ Pulling base image ...
    🎨 Creating docker container (CPUs=2, Memory=2200MB) ...
    🌐 Found network options:
        ■ NO_PROXY=192.168.49.2
    🐳 Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
        ■ env NO_PROXY=192.168.49.2
    🏷️ Verifying Kubernetes components...
    🎉 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
student@kcnaprep:~$ █

```

Figure 9.3: Output of minikube start

- Finally, verify that all necessary components are running with the **minikube status** command:

minikube status

Install kubectl

As discussed in the previous chapters, **kubectl** is the CLI tool for interacting with Kubernetes clusters and serves as the primary means for users to manage, deploy, and troubleshoot containerized applications. It also enables you to execute various commands to inspect cluster resources, create and update configurations, deploy applications, and view logs and metrics.

Perform the following steps to install **kubectl**:

- Start by downloading and installing the **kubectl** binary:

```

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

```

```
sudo install -o root -g root -m 0755 kubectl /usr/  
local/bin/kubectl
```

2. Verify that **kubectl** command works:

```
kubectl --version
```

Creating a Kubernetes cluster using Play with Kubernetes

Play with Kubernetes (<https://labs.play-with-k8s.com/>) is similar to *Play with Docker* used in [Chapter 3: Hands-on Docker and Containerd](#). It is an online playground that provides short lived instances where one can install Kubernetes and use it for learning purposes.

Note: Keep in mind that each session is valid for a duration of 4 hours, so try to complete the exercises within the timeframe. Also note that the instances are ephemeral hence all data in it is lost at the end of the session.

Prerequisites:

The following prerequisites are required to access Play with Kubernetes:

- A GitHub account or a Docker Hub account.
- A modern browser such as Chrome or Firefox.

Note: As of this writing login using Docker Hub account is failing, so login using GitHub account has been demonstrated. Use GitHub if Docker login fails.

Instructions:

Follow the steps in sequence to create a two-node Kubernetes cluster:

1. Visit <https://labs.play-with-k8s.com/>, click Login and select **github**, as shown in the following figure:



Figure 9.4: Login to play with kubernetes

2. On the login popup screen shown in the following figure, and login to your GitHub account:

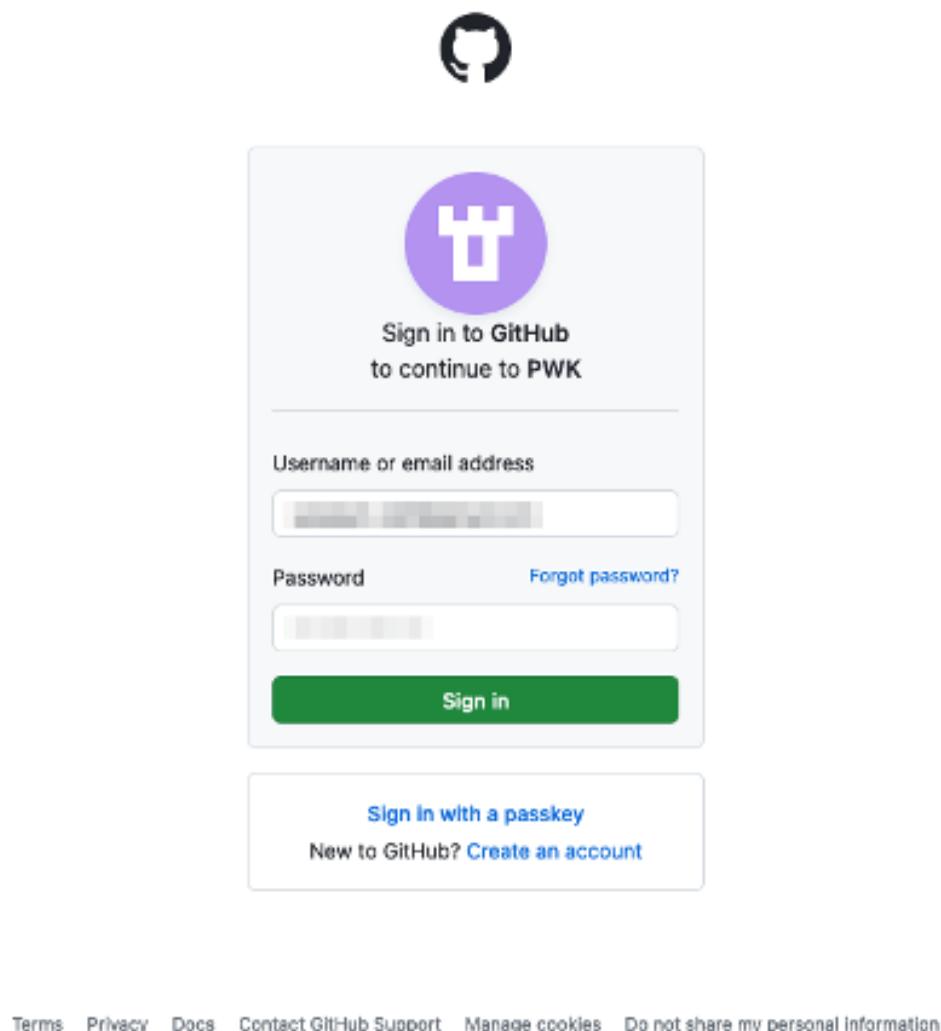


Figure 9.5: Sign in using github

3. On successful login, you should see a green **Start** button shown in the following figure:



Figure 9.6: Start a session

4. Click the **Start** button to start a session. Remember that a single session's duration is 4 hours. Try to complete the exercises, including installation, in this timeframe. At the end of the session, the cluster is terminated, meaning you must reinstall Kubernetes again. Fortunately, installation is just a few steps and doing it a few times is great for learning.
5. In the new session window, click **+ADD NEW INSTANCE** to add an instance and repeat the step again to add another instance. You should have two instances at the end of this step. One will act as the control plane (or master) node and the other a worker node:

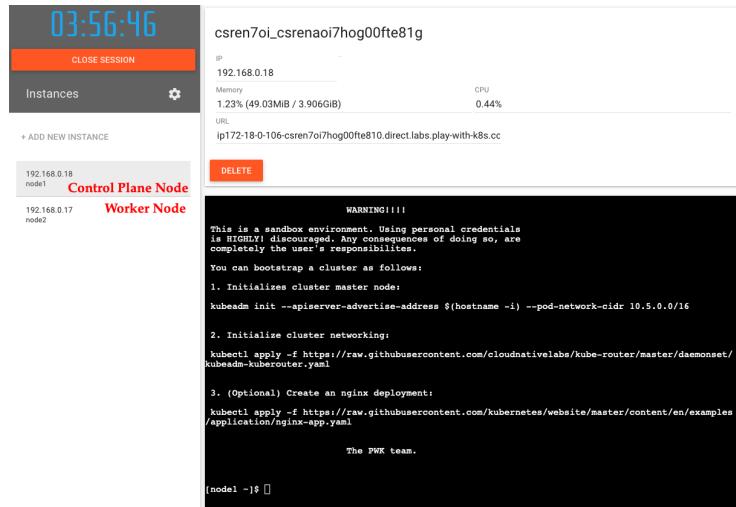


Figure 9.7: Add instances

6. Select **node1**, copy and paste the **kubeadm init** command displayed in *Step 1* on the **node1** terminal to bootstrap and initialize the control panel node.

For example:

```

kubeadm init --apiserver-advertise-address $(hostname -i) \
--pod-network-cidr 10.5.0.0/16

```

Remember to copy the command as it shows on your session screen as visible in the following figure, and not from this book, as the CIDR range may differ:

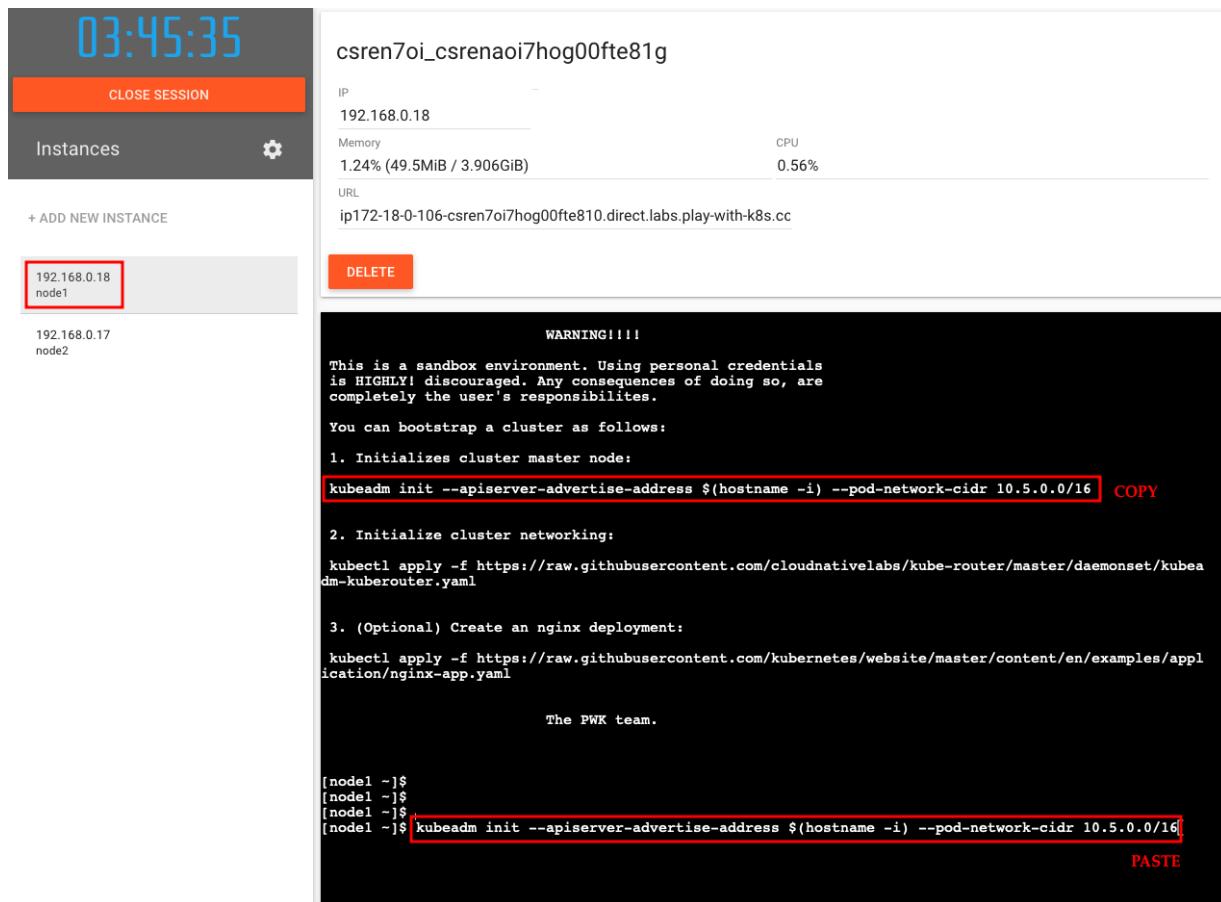


Figure 9.8: Copy and paste the kubeadm init command

7. The control plane should be initiated successfully, followed by further instructions. Copy the command displayed at the end starting with **kubeadm join** to join a worker node later in the steps. Do not paste and run it on this node.

Refer to the following figure for example:

```

03:31:49 csren7oi_csrenaoi7hog00fte81g
CLOSE SESSION
Instances
+ ADD NEW INSTANCE
192.168.0.18
node1
192.168.0.17
node2
DELETE
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for nodes to get long term certificate credentials
[bootstrap-token] Configured RBAC rules to allow the csraprover controller automatically approve CSRs from a Node Bootstrap Token
[bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client certificates in the cluster
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy
Your Kubernetes control-plane has initialized successfully!
To start using your cluster, you need to run the following as a regular user:
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
Alternatively, if you are the root user, you can run:
export KUBECONFIG=/etc/kubernetes/admin.conf
You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/
Then you can join any number of worker nodes by running the following on each as root:
kubeadm join 192.168.0.18:6443 --token r8naf0.u8p46ng0luxca7up \
--discovery-token-ca-cert-hash sha256:b5faef50b879d8a4d134c1e3ald8b5927988ca8d543497278ae29d09a6620663
Waiting for api server to startup
Warning: resource daemonsets/kube-proxy is missing the kubectl.kubernetes.io/last-applied-configuration annotation which is required by kubectl apply. kubectl apply should only be used on resources created declaratively by either kubectl create --save-config or kubectl apply. The missing annotation will be patched automatically.
daemonset.apps/kube-proxy configured
No resources found
[nodel ~]$ 

```

Figure 9.9: Copy kubeadm join command

8. Next, while still on the control plane node, run the following command to initialize the cluster networking. The following command was also displayed as *Step 2* in the cluster bootstrap process and can be copied from there:

```
kubectl apply -f \ https://raw.githubusercontent.com/cloudnativelabs/kube-router/master/daemonset/kubeadm-kuberouter.yaml
```

The screenshot shows the output of the command:

```

03:26:07
CLOSE SESSION
Instances
+ ADD NEW INSTANCE
192.168.0.18
node1
192.168.0.17
node2
DELETE

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:
export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:
kubeadm join 192.168.0.18:6443 --token r8naf0.u8p46ng01uxca7up \
--discovery-token-ca-cert-hash sha256:b5faef50b879d8a4d134c1e3a1d8b5927988ca8d543497278ae29d09a6620
663
Waiting for api server to startup
Warning: resource daemonsets/kube-proxy is missing the kubectl.kubernetes.io/last-applied-configuration annotation which is required by kubectl apply. kubectl apply should only be used on resources created declaratively by either kubectl create --save-config or kubectl apply. The missing annotation will be patched automatically.
daemonset.apps/kube-proxy configured
No resources found
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ kubectl apply -f https://raw.githubusercontent.com/cloudnative-labs/kube-router/master/daemonset/
kubeadm-kuberouter.yaml
configmap/kube-router-cfg created
daemonset.apps/kube-router created
serviceaccount/kube-router created
clusterrole.rbac.authorization.k8s.io/kube-router created
clusterrolebinding.rbac.authorization.k8s.io/kube-router created
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ 
[node1 ~]$ 

```

Figure 9.10: Initialize networking on control plane node

9. Next, run the following **kubectl get nodes** command to verify that the control plane node is ready:

kubectl get nodes

10. We are now ready to join the second instance as worker node to this cluster. Select **node2** from the instances list, paste and run the **kubeadm join** command copied earlier. The second instance will also display the same commands on the terminal as node1, do not run them.

An example of the **kubeadm join** command is as follows:

```

kubeadm join 192.168.0.13:6443 --token 3hqjnt.nnwu
n2x2q11g7z1d \
--discovery-token-ca-cert-hash \

```

**sha256:f8a6754a946fc934687e77c88aac04d5b610c2004c1
5443d43c3cc02bae787b**

The following screenshot shows an example of running the **kubeadm join** command:

The screenshot shows a terminal window titled "03:18:06" with a "CLOSE SESSION" button. Below the title, there's a "Instances" section with a "DELETE" button. A list of instances shows "node1" (IP: 192.168.0.18) and "node2" (IP: 192.168.0.17). The "node2" entry is highlighted with a red box. The main terminal area contains the output of the "kubeadm join" command. The output is as follows:

```
[node2 ~] $ kubeadm join 192.168.0.18:6443 --token r8naf0.u8p46ng0luxca7up \
>     --discovery-token-ca-cert-hash sha256:b5faef50b879d8a4d134cle3ald8b5927988ca8d543497278ae29d09a66
20663
Initializing machine ID from random generator.
W1115 07:16:40.229816      8534 initconfiguration.go:120] Usage of CRI endpoints without URL scheme is deprecated and can cause kubelet errors in the future. Automatically prepending scheme "unix" to the "criSocket" with value "/run/docker/containerd/containerd.sock". Please update your configuration!
[preflight] Running pre-flight checks
  [WARNING Swap]: swap is enabled; production deployments should disable swap unless testing the Node Swap feature gate of the kubelet
  [preflight] The system verification failed. Printing the output from the verification:
KERNEL VERSION: 4.4.0-210-generic
OS: Linux
CGROUPS_CPU: enabled
CGROUPS_CPUACCT: enabled
CGROUPS_CPUSET: enabled
CGROUPS_DEVICES: enabled
CGROUPS_FREEZER: enabled
CGROUPS_MEMORY: enabled
CGROUPS_PIDS: enabled
CGROUPS_HUGETLB: enabled
CGROUPS_BLKIO: enabled
  [WARNING SystemVerification]: failed to parse kernel config: unable to load kernel module: "configs", output: "", err: exit status 1
  [WARNING FileContent--proc-sys-net-bridge-bridge-nf-call-iptables]: /proc/sys/net/bridge/bridge-nf-call-iptables does not exist
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.
```

Figure 9.11: Run *kubeadm join* command on worker node

11. Come back to the terminal of node1 and run the following command **kubectl get nodes** to verify that you can see two nodes, one control plane, and one worker node. The node2 may not have a label called worker and may display <none>:

kubectl get nodes

The following screenshot shows the output of the command reflecting two nodes, one control plane and one worker:

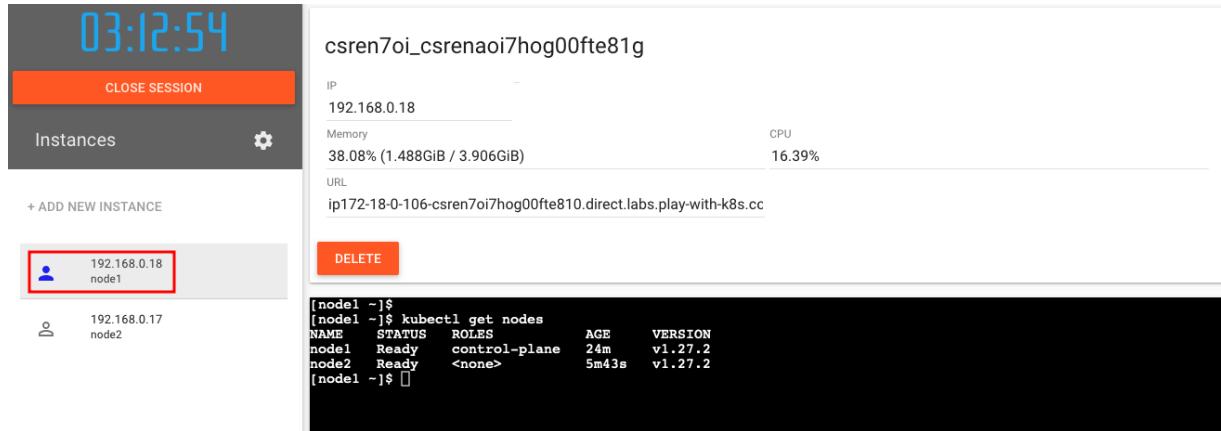


Figure 9.12: Command verifying the nodes and their status

This completes the cluster creation process. You can now proceed with the rest of this chapter.

Understanding the environment

Start by running a few commands to understand the Kubernetes cluster you just created.

List all nodes of the cluster with the following command:

kubectl get nodes

To view a more detailed list run the following:

kubectl get nodes -o wide

You should see an output similar to the following screenshot for the preceding commands:

```
student@kcnaprep:~$ kubectl get nodes
NAME      STATUS   ROLES      AGE      VERSION
minikube  Ready    control-plane   4m48s   v1.28.3
minikube-m02 Ready    worker      4m24s   v1.28.3
student@kcnaprep:~$ 
student@kcnaprep:~$ kubectl get nodes -o wide
NAME      STATUS   ROLES      AGE      VERSION  INTERNAL-IP  EXTERNAL-IP  OS-IMAGE           KERNEL-VERSION   CONTAINER-RUNTIME
minikube  Ready    control-plane   4m58s   v1.28.3  192.168.49.2  <none>        Ubuntu 22.04.3 LTS  6.5.0-1018-azure  docker://24.0.7
minikube-m02 Ready    worker      4m34s   v1.28.3  192.168.49.3  <none>        Ubuntu 22.04.3 LTS  6.5.0-1018-azure  docker://24.0.7
student@kcnaprep:~$
```

Figure 9.13: List nodes

View all Kubernetes API resources in a table format showing information such as the **NAME**, **SHORTNAMES**, **APIVERSION**, whether namespaced or not, and its **KIND** using the following command:

kubectl api-resources

Here is a screenshot showing the output of the command:

NAME	SHORTNAMES	APIVERSION	NAMESPACED	KIND
bindings		v1	true	Binding
componentstatuses	cs	v1	false	ComponentStatus
configmaps	cm	v1	true	ConfigMap
endpoints	ep	v1	true	Endpoints
events	ev	v1	true	Event
limitranges	limits	v1	true	LimitRange
namespaces	ns	v1	false	Namespace
nodes	no	v1	false	Node
persistentvolumeclaims	pvc	v1	true	PersistentVolumeClaim
persistentvolumes	pv	v1	false	PersistentVolume
pods	po	v1	true	Pod
podtemplates		v1	true	PodTemplate
replicationcontrollers	rc	v1	true	ReplicationController
resourcequotas	quota	v1	true	ResourceQuota
secrets		v1	true	Secret
serviceaccounts	sa	v1	true	ServiceAccount
services	svc	v1	true	Service
mutatingwebhookconfigurations		admissionregistration.k8s.io/v1	false	MutatingWebhookConfiguration
validatingwebhookconfigurations		admissionregistration.k8s.io/v1	false	ValidatingWebhookConfiguration
customresourcedefinitions	crd, crds	apiextensions.k8s.io/v1	false	CustomResourceDefinition
apiservices		apiregistration.k8s.io/v1	false	APIService
controllerrevisions		apps/v1	true	ControllerRevision
daemonsets	ds	apps/v1	true	DaemonSet
deployments	deploy	apps/v1	true	Deployment
replicasetss	rs	apps/v1	true	ReplicaSet
statefulsets	sts	apps/v1	true	StatefulSet

Figure 9.14: List API resources

To view the fields and structure of an API resource, use the **kubectl explain** command. For example, run the following command:

kubectl explain pod

To manage and modify **kubeconfig** files, used to access Kubernetes clusters, the **kubectl config** command can be used. Following are a few examples:

Command	Description
kubectl config view	View a combined configuration of all configured Kubernetes clusters, users and contexts (a context ties a user to a cluster).
kubectl config view -f ~/.kube/customconfig	View configuration of a specific kubeconfig file.
kubectl config current-context	Display the current context.

kubectl config set-context	Create a new context.
kubectl config use-context	Switch to another context.
kubectl config delete-context	Delete a context.

Table 9.1: Examples of `kubectl config` commands with their syntax and description

Creating resources in Kubernetes

In this section, you will write YAML definition files for common objects such as Pods, deployments, service, etc. and use them to create resources.

All YAML files used here are available in the accompanying GitHub repository for this book.

Creating your first Pod

The following steps create a simple bare Pod using the docker image created and published to *Docker Hub* in [Chapter 3, Hands-on Docker and Containerd](#), but before continuing, ensure that the image is available:

1. Begin by creating a folder which will serve as a working directory for the YAML files that we will create.
2. Create a file, say **pod.yaml**, for declaring the object, a Pod in this case, using the YAML format:

touch pod.yaml

3. Open the file in your editor of choice, such as **vi** or **nano**, and write the YAML file shown as follows.

Replace **USERNAME** with your username from Docker Hub.

For example: **sangramrath/kcnaprep:1.0**

apiVersion: v1

kind: Pod

```
metadata:  
  name: kcnaprep  
spec:  
  containers:  
    - name: kcnaprep  
      image: USERNAME/kcnaprep:1.0  
      ports:  
        - containerPort: 3000
```

4. Create the Pod by using the **kubectl apply** command:

```
kubectl apply -f pod.yaml
```

5. Verify that the Pod is running through the following command:

```
kubectl get pods
```

This may take a few seconds.

Note: **kubectl apply** is a declarative command whereas **kubectl create** is an imperative command. Both accept YAML files. The following commands in step 6 and 7 demonstrate other ways of creating Pods and how they work. They are not a continuation of the previous steps.

6. Start by trying to create a Pod using the **kubectl create** command:

```
kubectl create -f kcnaprep.yaml
```

You will find that you are presented by the error **Error from server (AlreadyExists): pods "kcnaprep" already exists**. The imperative approach will error if the resource already exists

Delete the **pod** using the following command and re-run the preceding **kubectl create** command again:

```
kubectl delete pod kcnaprep
```

Alternatively, you can change the name of the **pod** in the YAML file to create a Pod with a different name.

7. One can also create a Pod using the **kubectl run** command (like **docker run**). Run the following command replacing **USERNAME** with your docker hub username:

```
kubectl run kcnaprep-using-run --image=USERNAME/kcnaprep:1.0
```

Note: Here **kcnaprep-using-run** is the Pod name, used to avoid errors.

The recommended approach is to use the declarative approach.

Deployments

The preceding Pod is a standalone Pod and cannot provide workload management capabilities such as scaling, autoscaling, self-healing, application updates/rollouts, etc. To implement many of the features for which we have selected a container orchestrator, such as Kubernetes, we must create higher level Kubernetes objects such as Deployments.

Create a new **deployment.yaml** file to deploy the **kcnaprep** application as a deployment object. The final YAML would look like the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kcnaprep
spec:
  replicas: 1
  strategy:
```

```
type: RollingUpdate
selector:
  matchLabels:
    app: kcnaprep
template:
  metadata:
    labels:
      app: kcnaprep
spec:
  containers:
    - name: kcnaprep
      image: sangramrath/kcnaprep:1.0
      ports:
        - containerPort: 3000
```

Here is a quick rundown of the anatomy:

- The name of the deployment is **kcnaprep-deployment** specified under **.metadata.name**.
- The number of replicas is **1** as specified under **.spec.replicas**. A deployment uses a **ReplicaSet**, the default number of replicas is 1.
- The default strategy type is **RollingUpdate**.
- The **.spec.template** section of the deployment specifies the container details.

Create and verify the deployment with the following commands:

```
kubectl apply -f kcnaprep-deployment.yaml
```

kubectl get deployments

To view the associated Pods, run the following command:

kubectl get pods

If you have not deleted the bare Pod created earlier, you will find two Pods including one from the deployment that will have a different naming convention.

Here is a screenshot showing the output of the previous commands:

```
student@kcnaprep:~/chapter/09$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
kcnaprep   1/1     1           1           7m41s
student@kcnaprep:~/chapter/09$ 
student@kcnaprep:~/chapter/09$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
kcnaprep              1/1     Running   0          12m
kcnaprep-df468c4c5-zhnpx 1/1     Running   0          7m45s
```

Figure 9.15: List deployment and Pods

To scale a deployment, you can use one of the following methods:

- Edit the deployment (in place) and change the **replica** count.
- Edit the deployment YAML file, change the **replica** count and re-apply the YAML file.
- Run the **kubectl scale** command.

For this step, we will scale using the imperative approach. Run the following command to scale the number of **replicas** to 2:

kubectl scale --replicas=2 deployment/kcnaprep

Verify the deployment and the associated Pods again, as shown in the following figure:

```

student@kcnaprep:~/chapter/09$ kubectl scale --replicas=2 deployment/kcnaprep
deployment.apps/kcnaprep scaled
student@kcnaprep:~/chapter/09$ kubectl get deployments
NAME      READY  UP-TO-DATE  AVAILABLE  AGE
kcnaprep  2/2    2           2          14m
student@kcnaprep:~/chapter/09$ kubectl get pods
NAME                  READY  STATUS   RESTARTS  AGE
kcnaprep              1/1    Running  0          19m
kcnaprep-df468c4c5-2gwfk 1/1    Running  0          3m50s
kcnaprep-df468c4c5-zhnpx 1/1    Running  0          15m

```

Figure 9.16: List deployment and Pods after scale

Services

To access the deployment from within or outside the cluster, it is exposed using a Service. For this demonstration, we will use the **NodePort** service.

Create a YAML file, let us call it **service.yaml** with the following definition:

```

apiVersion: v1
kind: Service
metadata:
  name: kcnaprep
spec:
  type: NodePort
  selector:
    app: kcnaprep
  ports:
  - port: 3000
    targetPort: 3000
    nodePort: 30001

```

Create and verify that the service was created using the following commands:

```
kubectl apply -f service.yaml
```

```
kubectl get services
```

If you are performing these steps in minikube, the following additional **port-forward** command is required to be able to access the service from outside the cluster:

```
kubectl port-forward --address 0.0.0.0 service/kcnaprep  
30001:3000
```

It is also assumed that port **30001** is open at the firewall (if any).

To test that the service works, use the IP address of the node followed by **30001**:

```
curl http://NODE_IP:30001
```

For example:

```
curl http://127.0.0.1:30001
```

Basic operations and troubleshooting

In this section, we will see a few more examples of the **get** command which you are already familiar with from earlier, the **describe** command and learn how to view logs and events.

For a complete reference to **kubectl** commands, visit:
<https://kubernetes.io/docs/reference/kubectl/generated/>.

Displaying resources

The **kubectl get** command prints a table showing information about specific resource types.

Unless specified with the **-n** flag, the **kubectl** displays resources from the default namespace always.

Following are some more examples of it:

Example	Description
kubectl get all	List resources for all resource types in the default namespace.
kubectl get pods --all-namespaces	List Pods across all namespaces. -A instead of --all-namespaces can be used.
kubectl get deployments -n cncf	List all deployments in the cncf namespace. (-n is short for --namespace).
kubectl get pods -l environment=production	List all Pods with the label environment=production in the default namespace. You can also use --selector flag instead of -l .

Table 9.2: Examples of kubectl get commands with their syntax and description

Working with namespaces

To view all namespaces in the cluster, run the following command:

kubectl get ns

To create a new namespace, use the **kubectl create** command. For example:

kubectl create ns cncf

Describing resources

The **kubectl describe** command is used to view a detailed description of one or more resource of a type. It also displays related resources. Its syntax is as follows:

kubectl describe pods <podname>

For example:

```
kubectl describe pods kcnaprep-df468c4c5-2gwfk
```

The following figure shows a typical output of the **kubectl describe pods** command:

```
student@kcnaprep:~/chapter/09$ kubectl describe pods kcnaprep-df468c4c5-2gwfk
Name:           kcnaprep-df468c4c5-2gwfk
Namespace:      default
Priority:       0
Service Account: default
Node:           minikube/192.168.49.2
Start Time:     Sun, 21 Apr 2024 15:26:09 +0000
Labels:         app=kcnaprep
                pod-template-hash=df468c4c5
Annotations:    <none>
Status:         Running
IP:             10.244.0.3
IPs:
  IP:          10.244.0.3
Controlled By: ReplicaSet/kcnaprep-df468c4c5
Containers:
  kcnaprep:
    Container ID:  docker://cb24fcbfe358fbfd0b749b9682e34272d34de946e44c411cadc957caa0535a53
    Image:         sangramrath/kcnaprep:1.0
    Image ID:      docker-pullable://sangramrath/kcnaprep@sha256:377aae9530fe536fd166f0d556d5a8018f31
    Port:          3000/TCP
    Host Port:    0/TCP
    State:         Running
      Started:   Sun, 21 Apr 2024 18:21:41 +0000
    Last State:   Terminated
      Reason:    Completed
      Exit Code:  0
      Started:   Sun, 21 Apr 2024 15:26:09 +0000
      Finished:  Sun, 21 Apr 2024 15:32:16 +0000
    Ready:        True
    Restart Count: 1
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-ktnrm (ro)
Conditions:
  Type  Status
  Initialized  True
  Ready  True
  ContainersReady  True
  PodScheduled  True
Volumes:
  kube-api-access-ktnrm:
    Type:            Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:    kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI:     true
  QoS Class:      BestEffort
  Node-Selectors:  <none>
  Tolerations:    node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                  node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
```

Figure 9.17: Describe a Pod

Similarly, use the **describe** command to view details about the service. For example:

kubectl describe services kcnaprep

The **describe** command can output a lot of information that may be difficult to read. It does not provide a way to export it to a file, such as a YAML or JSON file. One can use the **kubectl get** command here to output the description to a file, for example a YAML file. The syntax is of the following format:

```
kubectl get pod <podname> -o yaml > <filename>
```

For example:

```
kubectl get pod kcnaprep-df468c4c5-wmw2c -o yaml > get-kcnaprep-pod.yaml
```

Viewing logs

Logs are important to troubleshoot a problematic Pod. The **kubectl logs** command helps you view logs of a container in a Pod. You must provide the Pod name and additionally the container name as well if there are multiple containers in a Pod.

For a Pod with one container use the following syntax:

```
kubectl logs <podname>
```

For example:

```
kubectl logs kcnaprep-df468c4c5-2gwfk
```

The following figure shows the output of **kubectl logs** command for the Pod created earlier:

```
student@kcnaprep:~/chapter/09$ kubectl logs kcnaprep-df468c4c5-2gwfk
Using sqlite database at ./data/kcnaprep.db
Listening on port 3000
```

Figure 9.18: Output of kubectl logs

For a Pod with multiple containers, use the following syntax:

```
kubectl logs <podname> -c <containername>
```

Viewing events

Kubernetes generates a lot of events related to various operations, and these events can provide important information when troubleshooting the cluster, Kubernetes resources, and more. To view all the events, run the following command, which should display an output similar to the following screenshot:

```
kubectl get events
```

The output is as follows:

3h36m	Normal	NodeHasSufficientMemory	node/minikube-m02	Node minikube-m02 status is now: NodeHasSufficientMemory
3h36m	Normal	NodeHasNoDiskPressure	node/minikube-m02	Node minikube-m02 status is now: NodeHasNoDiskPressure
3h36m	Normal	NodeHasSufficientPID	node/minikube-m02	Node minikube-m02 status is now: NodeHasSufficientPID
3h36m	Normal	NodeReady	node/minikube-m02	Node minikube-m02 status is now: NodeReady
3h36m	Normal	RegisteredNode	node/minikube-m02	Node minikube-m02 event: Registered Node minikube-m02 in Controller
3h36m	Normal	Starting	node/minikube-m02	Node minikube-m02 event: Registered Node minikube-m02 in Controller
6m48s	Normal	RegisteredNode	node/minikube-m02	Starting kubelet.
6m44s	Normal	Starting	node/minikube-m02	Node minikube-m02 status is now: NodeHasSufficientMemory
6m44s	Normal	NodeHasSufficientMemory	node/minikube-m02	Node minikube-m02 status is now: NodeHasNoDiskPressure
6m44s	Normal	NodeHasNoDiskPressure	node/minikube-m02	Node minikube-m02 status is now: NodeHasSufficientPID
6m44s	Normal	NodeHasSufficientPID	node/minikube-m02	Node minikube-m02 status is now: NodeReady
6m44s	Normal	NodeAllocatableEnforced	node/minikube-m02	Updated Node Allocatable limit across pods
6m43s	Normal	NodeReady	node/minikube-m02	Node minikube-m02 status is now: NodeReady
6m39s	Normal	Starting	node/minikube-m02	Starting kubelet.
3h36m	Normal	Starting	node/minikube	Updated Node Allocatable limit across pods
3h36m	Normal	NodeAllocatableEnforced	node/minikube	Node minikube status is now: NodeHasSufficientMemory
3h36m	Normal	NodeHasSufficientMemory	node/minikube	Node minikube status is now: NodeHasNoDiskPressure
3h36m	Normal	NodeHasNoDiskPressure	node/minikube	Node minikube status is now: NodeHasSufficientPID
3h36m	Normal	NodeHasSufficientPID	node/minikube	Node minikube status is now: NodeReady
3h36m	Normal	RegisteredNode	node/minikube	Node minikube event: Registered Node minikube in Controller
3h36m	Normal	Starting	node/minikube	Starting kubelet.
7m6s	Normal	Starting	node/minikube	Node minikube status is now: NodeHasSufficientMemory
7m5s	Normal	NodeHasSufficientMemory	node/minikube	Node minikube status is now: NodeHasNoDiskPressure
7m5s	Normal	NodeHasNoDiskPressure	node/minikube	Node minikube status is now: NodeHasSufficientPID
7m5s	Normal	NodeHasSufficientPID	node/minikube	Node minikube status is now: NodeAllocatableEnforced
6m52s	Normal	Starting	node/minikube	Updated Node Allocatable limit across pods
6m48s	Normal	RegisteredNode	node/minikube	Node minikube event: Registered Node minikube in Controller

Figure 9.19: View events

To watch the events, use the **--watch** flag like in the following command:

```
kubectl get events --watch
```

To view events from a specific namespace, use the **-n** flag. For example:

```
kubectl get events -n cncf
```

For a complete list of all possible parameters and flags use **kubectl get events --help**.

You can also view the events from the **kubectl describe** command. The events section at the end provides events related to the resource.

Following is a screenshot of a **kubectl describe pod <podname>** command:

Events:				
Type	Reason	Age	From	Message
Normal	Scheduled	3h14m	default-scheduler	Successfully assigned default/kcnaprep-df468c4c5-2gwfk to minikube
Normal	Pulled	3h14m	kubelet	Container image "sangramrath/kcnaprep:1.0" already present on machine
Normal	Created	3h14m	kubelet	Created container kcnaprep
Normal	Started	3h14m	kubelet	Started container kcnaprep
Normal	SandboxChanged	18m	kubelet	Pod sandbox changed, it will be killed and re-created.
Warning	Failed	18m (x2 over 18m)	kubelet	Error: services have not yet been read at least once, cannot construct envvars
Normal	Pulled	18m (x3 over 18m)	kubelet	Container image "sangramrath/kcnaprep:1.0" already present on machine
Normal	Created	18m	kubelet	Created container kcnaprep
Normal	Started	18m	kubelet	Started container kcnaprep

Figure 9.20: View events through kubectl describe

Persistent storage in Kubernetes

In this section, you will learn how to create a persistent volume and a persistent volume claim. You will also see how to use this persistent volume claim in a deployment. This is a simple demonstration and should not be considered a complete solution. The scenario is you want to persist any files generated in the **/var/log** directory of the container.

As usual, you can find the completed YAML files in the GitHub repository.

These steps have been tested in a Linux environment only. While the **Persistent Volume (PV)** and **Persistent Volume Claim (PVC)** creation steps will not be impacted by the OS, the Verifying data persistence step may not work as intended in a different OS.

Creating a Persistent Volume

For this exercise, create a YAML file named **pv.yaml** that defines a persistent volume based on host-path. Select a YAML file based on where you are running these steps, that is, minikube or Play with Kubernetes.

The following is the complete YAML file for **minikube**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
name: log
labels:
  type: local-volume
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/minikube-host/data/log"
```

The following is the complete YAML file for [Play with Kubernetes](#):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: log
  labels:
    type: local-volume
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
```

```
hostPath:
```

```
    path: "/data/log"
```

Create the PV and verify with the following commands:

```
kubectl apply -f pv.yaml
```

```
kubectl get pv
```

The output should be similar to the following screenshot:

```
student@kcnaprep:~/chapter/09$ kubectl apply -f pv.yaml
persistentvolume/log created
student@kcnaprep:~/chapter/09$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM      STORAGECLASS   REASON   AGE
log       1Gi         RWO           Retain        Available   manual      manual          4s
```

Figure 9.21: View persistent volumes

Creating a Persistent Volume Claim

Now, to request storage, we must create a PVC, which when used through a Pod will look for a suitable PV to bind the Pod to.

The YAML file for the PVC is:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: log
spec:
  storageClassName: manual
resources:
  requests:
    storage: 1Gi
accessModes:
  - ReadWriteOnce
```

Create the PVC and verify with the commands:

```
kubectl apply -f pvc.yaml  
kubectl get pvc
```

The following screenshot shows the output of these commands:

```
student@kcnaprep:~/chapter/09$ kubectl apply -f pvc.yaml  
persistentvolumeclaim/log created  
student@kcnaprep:~/chapter/09$  
student@kcnaprep:~/chapter/09$ kubectl get pvc  
NAME      STATUS    VOLUME   CAPACITY  ACCESS MODES  STORAGECLASS  AGE  
log       Bound     log       1Gi       RWO          manual        6s
```

Figure 9.22: View persistent volume claims

Updating the deployment to use the volume

Edit the **deployment.yaml** file to add the **volumeMounts** and the volumes specifications. The updated file should look like the following one:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: kcnaprep  
spec:  
  replicas: 1  
  strategy:  
    type: RollingUpdate  
  selector:  
    matchLabels:  
      app: kcnaprep  
  template:  
    metadata:
```

```
labels:
  app: kcnaprep

spec:
  containers:
    - name: kcnaprep
      image: sangramrath/kcnaprep:1.0
      volumeMounts:
        - name: log
          mountPath: /var/log
      ports:
        - containerPort: 3000
    volumes:
      - name: log
        persistentVolumeClaim:
          claimName: log
```

Apply the updates to the deployment using:

kubectl apply -f deployment.yaml

You will notice that if your deployment from earlier exercise existed, the output will say `deployment.apps/kcnaprep configured`. This will also scale down the deployment to 1 replica.

Verifying data persistence

To verify data persistence, you will connect to the container in the Pod and create a dummy log file. You will then exit from the container and verify that the file is present in the path.

Note: The `--mount` flag used during the minikube cluster

creation mounts the user home directory to **/minikube-host** inside the docker container. This means any data written to **/minikube-host** is available at the user home directory.

Begin by retrieving the Pod name and the node it is running on with the following command:

```
kubectl get pods -o wide
```

Next, use the following command to open a terminal to the container. Since this Pod has a single container, specifying the container name is not required:

```
kubectl exec -it <podname> -- /bin/sh
```

For example: **kubectl exec -it kcnaprep-7d7cf5f96c-1drwk -- /bin/sh**

On the container terminal, run the following commands to create a dummy log file, check that the file was created, and exit from the container:

```
head -c 1000 /dev/urandom > /var/log/error.log
```

```
ls /var/log
```

```
exit
```

For minikube, refer to the following the steps to verify:

1. List the contents of your user home directory, you should find a **data** folder. Listing the data folder should reveal the **error.log** file in it:

```
ls ~
```

```
ls data/log
```

For Play with Kubernetes, refer to the following steps to verify:

1. Use the information gathered earlier about the node on which the deployment Pod is running for the next step. (This will be mostly node2)

2. Click on the node from the navigation menu on the left.
3. On the node, list the **/data/log** folder to view the error log file.

StatefulSets and DaemonSets

In this section of the hands-on exercises, we will learn how to create StatefulSets and DaemonSets. Let us look at it in detail.

StatefulSets

For this part of the hands-on, you will create a StatefulSet for MySQL database. A StatefulSet requires a persistent volume through a persistent volume claim. A recommended way to do this is through dynamic volume provisioning. Since in our example (either the minikube or the Play with Kubernetes) we are using a sandbox environment, we will use a storage provisioner from *Rancher* that allows creating dynamic volumes using local paths.

Enable the Rancher storage provisioner in minikube using the following command:

```
minikube addons enable storage-provisioner-rancher
```

The YAML file for deploying the MySQL StatefulSet for this step looks like this:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  replicas: 2
  serviceName: mysql
  selector:
```

```
matchLabels:
  app: mysql

template:
  metadata:
    labels:
      app: mysql

spec:
  containers:
    - name: mysql
      image: mysql:5.7
      ports:
        - name: tcp
          protocol: TCP
          containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: strongpassword
      volumeMounts:
        - name: mysqldb-pvc
          mountPath: /var/lib/mysql

volumeClaimTemplates:
  - metadata:
      name: mysqldb-pvc
  spec:
```

```
storageClassName: "local-path"
```

```
accessModes:
```

```
- ReadWriteOnce
```

```
resources:
```

```
requests:
```

```
storage: 1Gi
```

Create the StatefulSet and verify with by running the following commands in sequence:

```
kubectl apply -f StatefulSet.yaml
```

```
kubectl get StatefulSets
```

```
kubectl get pods -o wide
```

The StatefulSet is created with one replica on each node, along with a PV and PVC for each replica, as shown in the following figure:

```
student@kcnaprep:~/chapter/09$ kubectl apply -f statefulset.yaml
statefulset.apps/mysql created
student@kcnaprep:~/chapter/09$ kubectl get statefulsets
NAME      READY   AGE   CONTAINERS   IMAGES
mysql     2/2    2m11s  mysql        mysql:5.7
student@kcnaprep:~/chapter/09$ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP          NODE   NOMINATED NODE   READINESS   GATES
kcnaprep-df468c4c5-2gwk   1/1   Running   1 (3h28m ago)  3h34m  10.244.0.3  minikube <none>           <none>
kcnaprep-df468c4c5-k42ng  1/1   Running   0            38m   10.244.0.4  minikube <none>           <none>
mysql-0       1/1   Running   0            2m27s  10.244.1.5  minikube-m02 <none>           <none>
mysql-1       1/1   Running   0            2m26s  10.244.0.7  minikube <none>           <none>
student@kcnaprep:~/chapter/09$ kubectl get pv
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                                     STORAGECLASS   REASON   AGE
log           1Gi        RWO           Retain          Bound   default/log                               manual        10m
pvc-531f1e11-1f06-4ae3-ae26-5ec9ec6f2e8c  1Gi        RWO           Delete         Bound   default/mysqldb-pvc-mysql-0             local-path   4m6s
pvc-7a024d08-3c0a-4257-8ba2-9c2b328b4da9  1Gi        RWO           Delete         Bound   default/mysqldb-pvc-mysql-1             local-path   3m45s
student@kcnaprep:~/chapter/09$ kubectl get pvc
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
log           Bound   log                                       1Gi        RWO           manual        10m
mysqldb-pvc-mysql-0  Bound   pvc-531f1e11-1f06-4ae3-ae26-5ec9ec6f2e8c  1Gi        RWO           local-path   4m15s
mysqldb-pvc-mysql-1  Bound   pvc-7a024d08-3c0a-4257-8ba2-9c2b328b4da9  1Gi        RWO           local-path   3m55s
```

Figure 9.23: View StatefulSets and its associated PVs and PVCs

DaemonSets

For the DaemonSets hands-on, we will deploy the Prometheus node exporter component. Deploying it as a DaemonSet ensures that an instance of node-exporter is running on all nodes exposing metrics from all of them.

The following is just a simple demonstration of a **DaemonSet** YAML file and does not reflect a completely working Prometheus setup:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: prometheus-node-exporter
spec:
  selector:
    matchLabels:
      tier: monitoring
      name: prometheus-node-exporter
  template:
    metadata:
      labels:
        tier: monitoring
        name: prometheus-node-exporter
    spec:
      containers:
        - name: prometheus-node-exporter
          image: quay.io/prometheus/node-exporter
          ports:
            - containerPort: 9100
```

Create the DaemonSet and verify with the following commands:

```
kubectl apply -f daemonset.yaml
```

```
kubectl get daemonsets
```

```
kubectl get pods -o wide
```

The following screenshot shows the output of these commands:

```
student@kcnaprep:~/chapter/09$ kubectl apply -f daemonset.yaml
daemonset.apps/prometheus-node-exporter created
student@kcnaprep:~/chapter/09$ kubectl get daemonsets
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
prometheus-node-exporter   2         2        2       2           2          <none>    50s
student@kcnaprep:~/chapter/09$ kubectl get pods -o wide
NAME             READY   STATUS    RESTARTS   AGE     IP          NODE   NOMINATED NODE   READINESS GATES
kcnaprep-df468c4c5-2gwk   1/1    Running   1 (3h34m ago)   3h40m  10.244.0.3  minikube  <none>      <none>
kcnaprep-df468c4c5-k42ng  1/1    Running   0          44m    10.244.0.4  minikube  <none>      <none>
mysql-0              1/1    Running   0          8m13s  10.244.1.5  minikube-m02 <none>      <none>
mysql-1              1/1    Running   0          8m12s  10.244.0.7  minikube  <none>      <none>
prometheus-node-exporter-8cjlh  1/1    Running   0          60s    10.244.1.6  minikube-m02 <none>      <none>
prometheus-node-exporter-pjrv5   1/1    Running   0          60s    10.244.0.8  minikube  <none>      <none>
```

Figure 9.24: View DaemonSets and its associated Pods

Conclusion

There are various ways to create a Kubernetes cluster for learning and development. minikube is one of them and is easy to setup. You can also use online platforms like Play with Kubernetes for a quick sandbox environment. You must install Docker before installing minikube and will also need **kubectl** for interacting with the cluster.

Using **kubectl** you can create and manage Kubernetes resources/API objects including being able to manage the nodes. The chapter has covered multiple examples of them for resources such as Pods, Deployments, Services, Persistent Volumes, Persistent Volume Claims, StatefulSets, DaemonSets etc. You can also use it to describe and troubleshoot resources, view logs and events. The exercises in this chapter will help you warm up and gain confidence in working with Kubernetes.

In the final chapter, we will learn about the KCNA exam such as the pattern, duration, passing percentage, format, etc. We will also learn about the exam environment when taking it from home and go through tips and tricks for a better chance at success.

Multiple choice questions

1. Which command would you run to create objects such as pods using a declarative approach?
 - a. kubectl create
 - b. kubectl apply
 - c. kubectl run
 - d. kubectl
2. Which of the following was used to deploy the MySQL workload?
 - a. DaemonSet
 - b. Deployment
 - c. StatefulSet
 - d. ReplicaSet
3. Which of the following commands would you run to see detailed information of an object such as a Pod?
 - a. kubectl describe
 - b. kubectl get
 - c. kubectl list
 - d. kubectl view
4. Select the Kubernetes commands that can be used to view events.
 - a. kubectl events
 - b. kubectl get events
 - c. kubectl describe
 - d. kubectl logs
5. You want to find out the CONTAINER-RUNTIME installed on the nodes displayed as a list. Select the flag that you

would use along with kubectl get nodes command to get the info.

- a. -o json
 - b. --help
 - c. --show-labels
 - d. -o wide
6. Select the command that you would run to view all Kubernetes API resources in a table format.
- a. kubectl api-versions
 - b. kubectl api-resources
 - c. kubectl version
 - d. kubectl config view
7. Choose the correct command to run for viewing logs of one of the containers called web-api in the Pod kcnaprep.
- a. kubectl logs web-api
 - b. kubectl get logs kcnaprep
 - c. kubectl logs kcnaprep -c web-api
 - d. kubectl logs kcnaprep
8. Which of the following storage provisioning methods is used for the Persistent storage in Kubernetes exercises?
- a. Static provisioning using local paths
 - b. Static provisioning using remote path
 - c. Dynamic provisioning using local paths
 - d. Empty directory
9. Which of the following Kubernetes resource type would you create for a workload that must be deployed on every node in the cluster?

- a. ReplicaSet
 - b. StatefulSet
 - c. DaemonSet
 - d. Deployment
10. Which command displays merged kubeconfig settings?
- a. kubectl cluster-info
 - b. kubectl config view
 - c. kubectl kubeconfig view
 - d. kubectl show

Answers

1	b.
2	c.
3	a.
4	a. b. c.
5	d.
6	b.
7	c.
8	a.
9	c.
10	b.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

About the Exam

Introduction

This final chapter provides information about the **Kubernetes and Cloud Native Associate (KCNA)** exam, such as the duration, number of questions, passing score, and other facts. It also includes information on how to prepare for taking the exam (before, during, and after), how to book the exam, the proctoring method, and more. You will find important links and tips as well.

Structure

This chapter has the following topics:

- The Kubernetes and Cloud Native Associate exam
- Booking the exam
- Preparing for the exam
- Before the exam
- Exam day
- Results

Objectives

The objective of this chapter is to familiarize with the exam format, the duration of the exam, the number of questions, the passing percentage, etc. among other facts. It will also provide you with information on how to book the exam, schedule it and prepare for taking the exam remotely. It covers various aspects of remote proctoring such as system requirements, identity requirements, exam environment, etc. The chapter also provides you recommendations for exam day such as pre-launch checks, how to launch the exam and things to keep in mind during the exam similar including a list of dos and do nots.

The Kubernetes and Cloud Native Associate exam

The KCNA exam is a pre-professional exam that validates a candidate's foundational knowledge of Kubernetes and related cloud-native technologies.

According to the *Linux Foundation*:

The KCNA is a pre-professional certification designed for candidates interested in advancing to the professional level through a demonstrated understanding of Kubernetes foundational knowledge and skills. This certification is ideal for students learning about or candidates interested in working with cloud-native technologies.

The KCNA certification demonstrates a candidate's knowledge of Kubernetes and its architecture, Kubernetes objects such as pods, deployments, services, **kubectl** commands, the Cloud Native Computing Foundation (CNCF) landscape, service mesh, GitOps, observability, cloud-native security etc. The KCNA curriculum has a detailed list of the exam domains.

Quick facts about the exam

Here are some facts about the exam for quick reference:

Duration	90 minutes (about 1 and a half
----------	--------------------------------

	hours)
Passing score	75%
Validity	2 Years
No. of attempts	2 (including one retake)
No. of questions	Approx 60
Question format	Multiple choice
Prerequisites	None
Cost	\$250 (as of this writing)

Table 10.1: Quick facts about the KCNA exam

The exam is proctored by PSI through their proctoring platform called Bridge using their own browser called PSI secure browser. The browser is designed to avoid cheating in the exam. The browser gets installed when launching the exam.

Booking the exam

Booking the exam involves purchasing the exam and scheduling it. You also need to create a Linux Foundation account if this is your first time taking one of their exams.

Purchasing the exam

To purchase the exam head over to the KCNA certification page at <https://training.linuxfoundation.org/certification/kubernetes-cloud-native-associate/> and click Enroll Today if you want to purchase the exam only. Click Buy Bundle if you want to buy the exam and the official course as an additional preparation source.

Sign in if necessary and provide payment information to complete the purchase. Use a coupon code if you have one. A single purchase provides

two exam attempts. You will receive order confirmation and scheduling instructions by email.

Tip: Linux Foundation has offers and promotions running sometimes that may help reduce the certification cost. Subscribe to their newsletter to be informed of these offers and plan accordingly.

Booking the exam does not automatically schedule it. You have 12 months to schedule and take the exam from the date of purchasing the exam. The second attempt, if necessary, must also be taken within 12 months of purchase.

Scheduling the exam

To schedule the exam, login to the Linux Foundation My Portal at <https://trainingportal.linuxfoundation.org/learn/dashboard/>. You should see KCNA under My Learning, click Resume button. This should bring you to the **Exam Preparation Checklist** page.

You must Agree to Global Candidate Agreement and verify your name before you can schedule your exam. It may take up to 30 minutes for the Schedule button to become active.

Here is a screenshot of the exam preparation checklist page for KCNA:

The screenshot shows the 'Exam Preparation Checklist' page from the Linux Foundation My Portal. At the top, there's a 'Menu' button and a title 'Exam'. Below the title, the page header reads 'Exam Preparation Checklist' and displays 'Exam Code: [REDACTED]' and 'Expiration Date: [REDACTED]'. The checklist consists of several items, each with a status indicator (green checkmark for completed, grey circle for pending), a description, and a corresponding action button or link. The items are:

- Agree to Global Candidate Agreement**: Status: Pending, Read Now link.
- Verify Name**: Status: Done.
- Select Platform**: Status: Pending, Platform: Ubuntu.
- Schedule an Exam**: Status: Pending, Exam Date: [REDACTED], Schedule button.
- Check System Requirements**: Status: Pending, Check Now link.
- Get Candidate Handbook**: Status: Pending, Read Now link.
- Read the Important Instructions**: Status: Pending, Read the Important Instructions link.
- Take Exam**: Status: Pending, Pending this section will be available once you have completed all the steps above.

At the bottom, there's a 'Where's my retake?' link, a note about support tickets, and a 'Previous Results' button.

Figure 10.1: Exam Preparation Checklist

Click **Schedule** to get redirected to the PSI dashboard. The page will show your eligibilities to take various tests. Click the exam name, that is, KCNA in this case. Read the *FAQs* and *Test Instructions* if needed and click the **Continue Booking** button when ready to go through a series of steps to schedule the exam. Here is the flow:

- Select the test format, which is already set to **Online Proctored (Live)**.
- Read the requirements and **acknowledge** it. Also **acknowledge** that you have access to a secure room for taking the test.
- Verify the candidate details. You cannot modify them here.
- Register your mobile number for SMS updates and alerts. This is optional.
- Upload a **Photo Identification**. You can also skip this here but it's recommended to do it.
- Choose a **Country** and **Timezone** to view and select available time slots.
- Select a slot that works and click **Book This Time Slot**.
- Review details one final time, **agree** to the terms and conditions, and click **Confirm Booking**.
- Pay if required to complete the process.

You will receive a test schedule confirmation email after this. Remember to set an alert or reminder.

Tip: An important motivating factor is to schedule an exam date. This works as a deadline and helps you get that certification done. In a rare scenario, if you must reschedule or cancel the exam, you can do so up to 24 hours before the exam.

Preparing for the exam

It is important to understand and familiarize oneself with the exam objectives. They help you identify topics that you are comfortable with and the topics that need additional effort. These were mentioned at the beginning of the relevant chapters in this book. The official exam curriculum is available at:

https://github.com/cncf/curriculum/blob/master/KCNA_Curriculum.pdf.

It is also safe to assume that there may be a few questions outside of the official scope but relevant to the nature of the exam.

While this is not a practical exam, hands-on experience working with Kubernetes, especially common **kubectl** commands to deploy, manage, and troubleshooting resources, is recommended as there may be questions around it. This has been covered in the *Hands-on Kubernetes* chapter.

Apart from using this book as a study guide, following are some additional resources you can use for preparing for the exam:

- Official accompanying training from Linux Foundation, the *Kubernetes and Cloud Native Essentials (LFS250)* course. It is a good idea to buy this and the exam as a bundle.
- Official Kubernetes documentation for topics mentioned in the exam curriculum.
- Linux Foundation offers free courses that are great resources, especially if you are new to the whole Linux, Cloud and Kubernetes space. Links are at the end of this chapter:
 - Introduction to Linux (LFS101x)
 - Introduction to Cloud Infrastructure Technologies (LFS151x)
 - Introduction to Kubernetes (LFS158x)

Creating a learning plan for preparation that is broken down will ease the process, and be less stressful and efficient.

There are no prerequisites to taking this exam.

Before the exam

While preparing to answer questions in the exam is important, it is equally important to prepare well in advance for taking the exam on the exam day. In this section, we will talk about the candidate handbook, system requirements, other important instructions and setting up the remote environment.

The Exam Preparation Checklist also has three of these listed as preparatory steps.

Candidate handbook

It is highly recommended to review the candidate handbook, especially if this is the first time taking an exam remotely from home through PSI, which is the proctoring platform used by the Linux Foundation. This is updated from time to time and may contain important changes that may impact your exam experience.

While the entire handbook cannot be covered here, the following are some important highlights from the candidate book:

- You will be monitored closely by a proctor using video, audio and screensharing methods. You must always remain within the camera frame.
- There is zero tolerance for misconduct.
- You should run the PSI Online Proctoring System Check to ensure you meet the system requirements before taking the exam. You must ensure you have a computer that meets the system requirements, more about this is provided later in this chapter.
- You will take the exam through a secure browser provided by PSI called PSI secure browser. This gets installed when you launch the exam.
- You should have a Linux Foundation account.

The candidate handbook is available at <https://docs.linuxfoundation.org/tc-docs/certification/lf-handbook2>.

System requirements

The PSI bridge provides a system check utility that will help you ensure all system requirements are met. It is available at <https://syscheck.bridge.psiexams.com/>. You must run this system compatibility check from the same computer from which you plan to take the exam and do it well in advance to be able to troubleshoot any issues.

A high-level list of the recommended system requirements is as follows:

- Windows 10 or 11 (64 bit only)
- MacOS Ventura (version 13) or above
- Ubuntu 20.04 and above
- A minimum of 8GB RAM, 16GB is recommended
- Free disk space of 500MB or more
- A modern broadband internet connection of 3 Mbps or higher
- A working microphone
- A webcam that can be used for performing a 360-degree scan of the room
- A modern browser, Chrome is recommended
- Administrative permissions on the computer
- Having a power backup would be great
- When using a laptop, ensure it is plugged in during the exam and not running on battery. The battery may drain faster due to the devices and apps running for remote proctoring.

Refer to the link in the table at the end of this chapter for detailed PSI secure browser System Requirements.

Tips: The following are some tips that can help you prepare the system for a better experience:

- Disable any VPN on the system and ensure firewall settings, if any are in order.
- Disable any active anti-virus scanning during the exam. This may help with performance issues regarding memory and CPU.
- It is recommended to take the exam from a personal computer rather than a corporate or work provided computer as the latter usually has restricted permissions for devices such as webcams, microphones or run firewalls and VPNs that may hinder exam experience.

You cannot use multiple monitors

Identity requirements

You must show a valid government-issued ID in the original form on the day of the exam during the check-in process. Some additional points to remember are:

- The ID cannot be a photocopy.
- The ID should contain your full name, photo and signature. The name, especially the first and the last name, should match exactly to that of the verified name in the Linux Foundation account.

Verify your name before the exam for any errors and contact the team for changes. Once you launch the exam, you cannot update your name.

Environment

Since this is a remotely proctored model, it is very important that the environment used to take the exam meets the requirements. Any deviations here usually result in exam suspension:

- Avoid any objects, such as pens, paper, books, electronic devices, etc., on the desk used to take the exam. There should not be any

objects below the desk either. You will be asked to show the desk area and under it during the exam area check step through the webcam.

- It is also not allowed to have printouts on walls. The proctor may ask you to remove any wall hanging that they deem suspicious.
- The exam area should be well lit, although there should not be any bright lights falling at your face since it makes it difficult to see you clearly. Bright lights or windows behind you are also not allowed.
- The exam area should be closed and private, preferably less noisy, and no one should enter the exam area at any time during the exam.
- You can have a glass of water in a transparent container.

Tips:

- Prepare the room much earlier to avoid a last-minute rush and delay in starting the exam. In rare cases where there is too much clutter, the proctor may not allow you to take the exam.
- A web cam with a long cable makes it easy to show the proctor around the room compared to the webcam on a laptop.
- An early morning or a late-night slot is relatively quieter and more distraction free.
- If you work out of an office and have access to a closed cabin, preferably without large open external sides, make use of it. The smaller and less cluttered the space is, the faster the environment check process.

PSI has a video that provides important information about the exam experience. You can watch it at <https://psi.wistia.com/medias/5kidxdd0ry>.

Exam day

Let us look at some recommendations on final checks, launching and taking the exam on the exam day.

Pre-launch checks

On the day of the exam, spend some time doing the following checks one last time before launching the exam to avoid last-minute issues:

- Check that your exam environment is decluttered.
- Verify again that your system is in working order, including the webcam, microphone, etc. Check that there are no updates running in the background, or firewalls and VPN that may hinder the launch process. Run the PSI Online Proctoring System Check process once again.
- If you are using a laptop, ensure it is fully charged. However, as recommended earlier do not take the exam in unplugged mode.
- Ensure your identification documents are in place and keep them next to the exam area.

Launching the exam

To launch the exam, you must click the Take Exam button in the My Portal's Exam Preparation Checklist section. It is activated 30 minutes before the scheduled date and time.

Tip: It is recommended to launch the exam at least 15 minutes or as much as 30 minutes before the scheduled time. This provides ample time for pre-exam steps such as installing/updating the secure exam browser, ID validation, environment verification by the proctor, etc.

On clicking the Take Exam button, you are redirected to the PSI dashboard to launch the exam. Click the Launch button here to start the

Secure Browser installation process. If you already have the latest version of the browser, it will start the self-check-in process.

The self-check-in process captures your photo and a photo of your identification document, does an initial verification, and then passes the control to a proctor who communicates with you through chat. This may take a few minutes, depending on the number of candidates in the queue.

At this stage, the proctor will do another round of checking of the photo and the identification document uploaded earlier and then, most importantly, ask you to pan your webcam around the room for inspection. The proctor will ask you to show your walls, ceiling, floor, under the table, where the system is, around it, and any other areas necessary. Finally, the proctor will share important dos and do not make statements before giving you access to take the exam.

Note: The time spent for the verification process is not part of the exam duration.

You must agree to the terms and conditions of the exam, including the NDA, before you can start answering the questions. You now have 90 minutes (about 1 and a half hours) from here to complete the exam.

During the exam

Relax and focus on the exam itself during the exam. Answer questions that you are confident of and mark questions for review that you are unsure of or need more time to understand.

Relax and focus on the questions, read them carefully. Answer questions that you are confident of. Mark for review and move on to the next question if you are unable to understand them in the first 30 seconds. You can return to answering these later.

Results

The scoring is automatic, and the result is a pass or fail with a score out of 100. Results are out, typically in 24 hours. You can also view the status of the result in the Exam Preparation checklist page of KCNA from your Linux Foundation portal.

A formal email about the exam results is received within three business days, including the Certificate ID number. You will receive a separate email for viewing and downloading the certificate. The certificate can also be downloaded from the Exam Preparation checklist page of KCNA in your Linux Foundation portal.

Linux Foundation uses Credly, a badge management platform, for assigning and sharing badges. It may take up to a week or more before the badge, shown in the following figure, is available on your Credly account:

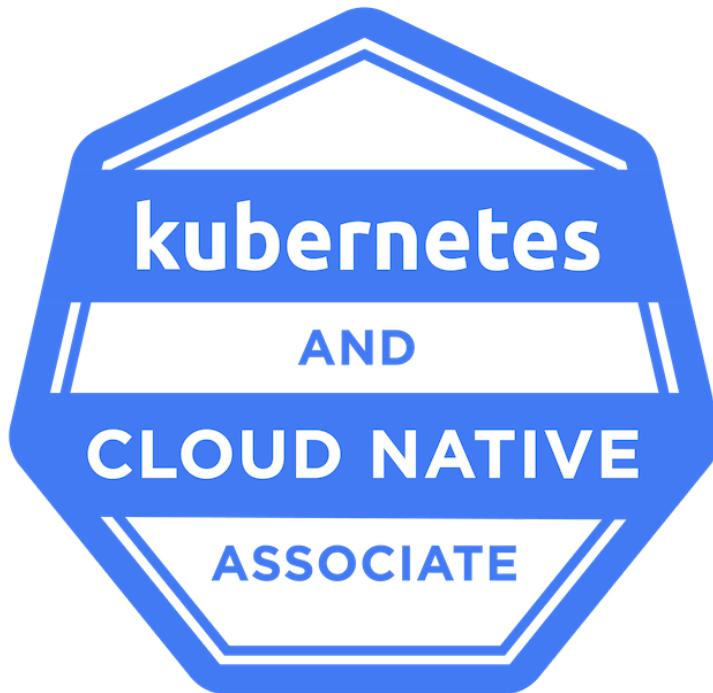


Figure 10.2: KCNA Certification Badge

Share your success story, preparation method, tips, certificate and badge within your network through platforms such as *LinkedIn*, *GitHub* etc.

In case you do not pass the exam on your first attempt, do not lose heart. Prepare and sit for the exam again. A second attempt is included for free in the exam purchase.

Links

Here is a list of important exam-related links:

KCNA Curriculum	https://github.com/cncf/curriculum/blob/master/KCNA_Curriculum.pdf
Linux Foundation My Portal	https://trainingportal.linuxfoundation.org/learn/dashboard/
PSI secure browser System Requirements	https://helpdesk.psionline.com/hc/en-gb/articles/4409608794260-PSI-secure-browser-System-Requirements
System compatibility check	https://syscheck.bridge.psiexams.com/
PSI Bridge FAQs	https://helpdesk.psionline.com/hc/en-gb/articles/4409608794260--PSI-Bridge-FAQ-System-Requirements
Internet bandwidth test	https://videodiagnostics.twilio.com/ https://speed.cloudflare.com/
KCNA Certification page	https://training.linuxfoundation.org/certification/kubernetes-cloud-native-associate/
Microphone test	https://video-

		diagnostics.twilio.com/ www.onlinemictest.com
Webcam test		<p>Windows: https://www.webcamtests.com/</p> <p>MacOS: https://video-diagnostics.twilio.com/</p> <p>Linux: https://www.onlinemictest.com/webcam-test/</p>
PSI Online Experience video	Proctoring	https://psi.wistia.com/medias/5kidxdd0ry

Conclusion

The KCNA exam is a foundational exam that tests your fundamental understanding of containers, Kubernetes, and related cloud-native technologies. It is a multiple-choice exam that lasts 90 minutes with approximately 60 questions to answer.

Through reading this chapter, you are now familiar with the exam format, the duration of the exam, the number of questions, and the passing percentage, among other facts. You should have a good understanding of how to register, schedule, and prepare to sit for the exam. Additionally, you are now familiar with the requirements for remote proctoring, how to prepare for a test-taking environment, and the dos and nots during the exam.

On successful passing of the exam, you get the KCNA certificate that is valid for 2 years.

Best of luck!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord.\(bpbonline.com](https://discord(bpbonline.com)



Index

A

Amazon Elastic Kubernetes Service (EKS) [47](#)
Amazon Web Services (AWS) [7](#)
Apache OpenWhisk [140](#)
application architectures [2](#)
 microservices [3](#)
 monolithic [2](#)
ArgoCD [175](#)
autoscaling in Kubernetes
 Cluster Autoscaler [134, 135](#)
 Horizontal Pod Autoscaler (HPA) [133, 134](#)
 KEDA [135](#)
 Vertical Pod Autoscaler (VPA) [134](#)
Azure Kubernetes Service (AKS) [47, 79](#)

B

Backend-as-a-Service (BaaS) [137](#)
basic operations and troubleshooting [197](#)
bind mount [65](#)
Bridged [34](#)

built-in controllers 81

C

cAdvisor 155

Calico 137

Certified Kubernetes Administrator (CKA) 20

cgroups 30

CI/CD 16, 168

- continuous delivery 169

- continuous deployment 170

- continuous integration 168, 169

cloud computing 6

- community cloud 8

- hybrid cloud 8

- private cloud 7

- public cloud 7

cloud-native

- basics 15, 16

cloud-native application delivery 168

cloud-native architecture 127-129

- fundamentals 128

- key tenets 129-131

cloud-native architectures 15

Cloud Native Computing Foundation (CNCF) 1, 16, 17, 43

- career opportunities 20, 21

- certifications 19, 20

- URL 16

cloud-native landscape 17

cloud-native observability 154, 155

cloud-native personas 143

cloud architect 143
cloud engineer 144
cloud solutions architect 144
data engineer 146
DevOps engineer 144, 145
DevSecOps engineer 146
full stack developer 146, 147
security engineer 146
Site Reliability Engineer 145
cloud-native security 135
4 Cs 135, 136
defense-in-depth 136
zero trust 136
cluster 78
Cluster Autoscaler 134
community cloud 8
compute nodes
 worker nodes 38
computing
 edge computing 14
 fog computing 14
 serverless 13, 14
computing fundamentals 5
 cloud computing 6-9
 containers 10
 future of applications and computing 13, 14
 virtualization 5, 6
Consul 118
features 119

container creation
 workflow [32, 33](#)

containerd [31, 67](#)
 container images, working with [68](#)
 containers, creating [68, 69](#)
 containers, managing [68, 69](#)
 installing [67, 68](#)
 nerdctl utility [69-71](#)
 prerequisites [67](#)
 storage and networking, with ctr utility [69](#)

containerd-shim [31](#)

container engines [31](#)

container images [12, 27-29](#)

container managers [31](#)

container networking
 types [34](#)

Container Network Interface (CNI) [42](#)
 URL [42](#)

container orchestration [12, 36, 37](#)
 benefits [39, 40](#)
 fundamentals [37](#)

container orchestrator [37](#)
 architecture [38](#)
 Docker Swarm [38](#)
 functions [39](#)
 Kubernetes [37](#)
 Marathon [38](#)
 OpenShift [38](#)
 workflow [38](#)

container registry 12, 29, 30
container runtime 12, 30
 containerd runtime 44, 45
 CRI-O runtime 45
 high-level container runtime 31
 low-level container runtime 30, 31
 rkt container runtime 45
 sandboxed container runtime 32
 virtualized container runtime 32
Container Runtime Interface (CRI) 41, 42, 83
containers 26, 30
 benefits 11
 business benefits 11, 12
 versus, virtualization 13
container security 35
 best practices 35, 36
container storage 33, 34
Container Storage Interface (CSI) 42
controllers in Kubernetes
 DaemonSet controller 81
 job controller 81
 StatefulSet controller 81
control nodes
 control plane nodes 38
 manager nodes 38
 master nodes 38
control plane, Kubernetes 80
 API server 80
 cloud controller manager 82

- controllers [81](#)
- etcd store [81](#)
- scheduler [81](#)

CoreDNS [115](#)

cost management [162](#)

- best practices [162, 163](#)

custom resource definition (CRD) [97, 134](#)

D

- DaemonSet controller [81, 89](#)
- DaemonSets [207, 208](#)
- data engineer [146](#)
- deployment controller [85](#)
- DevSecOps engineer [146](#)
- Docker [27, 54](#)
 - additional container operations [60, 61](#)
 - architecture [27](#)
 - cleanup [66, 67](#)
 - container, creating [59](#)
 - container, running [59, 60](#)
 - data persisting, with bind mount [65](#)
 - data persisting, with Docker volumes [64](#)
 - images and image registry, working with [61-63](#)
 - installing [181, 182](#)
 - multiple containers, creating on same network [65, 66](#)
 - networks [65](#)
 - prerequisites [55](#)
 - setting up [54](#)
- Docker Desktop [54](#)
- Docker Engine [27](#)

Dockerfile 27

 creating 57-59

Docker Hub 55, 61

 account, creating 55-57

Docker Swarm 38

Docker Swarm mode 38

Domain Name System (DNS) 34

E

east-west traffic 34

Elasticsearch 161

event-driven computing 138

F

faasd 140

FinOps 163

Flagger 174

Fluentd 161

Flux 174

full stack developer 146, 147

Functions-as-a-Service (FaaS) 13, 137, 138

 benefits 138

G

GitOps 171

 concepts 172, 173

 workflow 173, 174

GitOps tools 174

Google Kubernetes Engine (GKE) 47, 79

Grafana 159, 160

gVisor 32

H

Horizontal Pod Autoscaler (HPA) [133](#)

hybrid cloud [8](#)

hypervisor [5](#)

I

Infrastructure-as-a-Service (IaaS) [8](#)

Infrastructure as Code (IaC) [171](#)

Ingress [115](#)

instrumentation [153](#)

Internet Control Message Protocol (ICMP) [153](#)

inter-process communication (IPC) resources [30](#)

IPVlan [35](#)

Istio [117](#)

J

Jaeger [161](#)

JSONPath template [93](#)

K

K3s [47](#)

Kata [32](#)

KCNA exam day [218](#)

exam, completing [219](#)

exam, launching [219](#)

pre-launch checks [218, 219](#)

related links [221](#)

results [220](#)

KCNA exam preparation checklist

candidate handbook [216](#)

environment [217, 218](#)

identity requirements [217](#)

system requirements [216, 217](#)

Key Management Service (KMS) [106](#)

Kibana [161](#)

Knative [139](#)

Kops [79](#)

Kubeadm [79](#)

kubeapi-server [82](#)

kubectl

installing [183, 184](#)

kubectl api-resources command [97](#)

kubectl label command [94](#)

kubectl utility [79](#)

kube-proxy [82](#)

Kubernetes [15, 18, 19, 37, 43, 44](#)

architecture [76-78](#)

autoscaling [133](#)

control plane [80](#)

creating, with minikube [180](#)

deployments [79](#)

environment [192](#)

installing [78, 79](#)

minimum prerequisites [181](#)

networking [46, 110](#)

resiliency [131](#)

runtime [44](#)

scheduling [93](#)

security [46, 136, 137](#)

security best practices [104](#)

service mesh [116, 117](#)

storage [46](#)

worker node [82](#)

Kubernetes and Cloud Native Associate (KCNA) exam 211, 212

booking 213

preparation checklist 215

preparing for 215

purchasing 213

quick reference 212

scheduling 213, 214

Kubernetes API 96, 97

Kubernetes cluster

creating, with Play with Kubernetes 184-191

Kubernetes Event-driven Autoscaling (KEDA) 135

Kubernetes resources 83

creating 92

CronJobs 90

DaemonSets 89

deployment 85, 86

Jobs 90

kubectl examples 93

manifests 92, 93

namespaces 83, 84

Pods 84

requests and limits 91

StatefulSets 87, 88

storage and networking resources 92

types of containers, in Pods 85

kube-scheduler 81, 93

Kubespray 79

kube-state-metrics 156

L

let me contain that for you (LMCTFY) 11
lightweight Kubernetes distributions 47

K3s 47

MicroK8s 47

Minikube 47

Linkerd 118

Linux Foundation Education website 20

Logstash 161

low-level container runtime 30

examples 31

M

Macvlan 34

managed Kubernetes platforms 46, 47

Manifest file 92

Marathon 38

metrics-server 156

MicroK8s 47

microservices architecture 3

minikube 180

installing 182, 183

Minikube 47

monitoring 153, 154

monolithic architecture 2

versus, microservices 3-5

multi-cloud/Hybrid Kubernetes 47

multi-factor authentication (MFA) 36

N

namespaces 30

inter-process communication namespaces 30

network namespaces 30

- Process ID namespaces [30](#)
- UNIX Time-sharing (UTS) namespaces [30](#)
- user namespaces [30](#)
- native observability in Kubernetes [155](#)
 - events [157, 158](#)
 - logging [156, 157](#)
 - metrics and monitoring [155, 156](#)
- nerdctl utility [69](#)
- networking, Kubernetes [110](#)
 - ClusterIP [110, 111](#)
 - DNS [114, 115](#)
 - headless service [113, 114](#)
 - Ingress [115, 116](#)
 - load balancer [112, 113](#)
 - NodePort [111, 112](#)
 - service [110](#)
- north-south traffic [34](#)

O

- observability [154](#)
- Open Container Initiative (OCI) [41](#)
 - specifications [31, 41](#)
- OpenFaaS [139](#)
- OpenShift [38](#)
- OpenShift Container Platform (OCP) [38](#)
- open-source observability tools [158](#)
 - Elasticsearch [161](#)
 - Fluentd [161](#)
 - Grafana [160](#)
 - Jaeger [161](#)

Kibana [161](#)
Logstash [161](#)
OpenTelemetry [162](#)
Prometheus [158, 159](#)
Thanos [161](#)
open standards [40, 142, 143](#)
Open Systems Interconnection (OSI) model [116](#)
OpenTelemetry [162](#)
operations and troubleshooting
 events, viewing [200, 201](#)
 logs, viewing [200](#)
 namespaces, describing [198, 199](#)
 namespaces, working with [198](#)
 resources, displaying [198](#)
Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE) [47](#)
ordinal index [87](#)
Overlay [34](#)

P

persistent storage, in Kubernetes [201](#)
 data persistence, verifying [204, 205](#)
 deployment, updating [203, 204](#)
Persistent Volume Claim, creating [202, 203](#)
Persistent Volume, creating [201, 202](#)
PersistentVolumeClaims (PVCs) [120](#)
Platform-as-a-Service (PaaS) [8, 138](#)
Play with Docker [54](#)
Pods [84](#)
PodSpec [82](#)
private cloud [7](#)
probes [132, 156](#)

Prometheus [158](#), [159](#)

metric types [160](#)

Prometheus Expression Browser [159](#)

public cloud [7](#)

R

reconciliation [173](#)

resiliency in Kubernetes [131](#)

deployments [131](#)

network resiliency [132](#)

probes [132](#)

ReplicaSets [131](#)

StatefulSets [132](#)

storage resiliency [132](#)

resources in Kubernetes

creating [193](#)

deployments [195](#), [196](#)

Pod, creating [193](#), [194](#)

services [197](#)

role-based access control (RBAC) [46](#)

runc [31](#)

S

sandboxed container runtimes [32](#)

scale-down [134](#)

scale-up [134](#)

scheduling, in Kubernetes [93](#)

advanced scheduling [94](#)

affinity [95](#)

anti-affinity [95](#)

basic scheduling [94](#)

nodeName [95](#)
nodeSelector [94](#)
taints and tolerations [95](#)
security engineer [146](#)
security, in Kubernetes [104, 136, 137](#)
 4 Cs of Cloud Native Security [104, 105](#)
 cluster security [105, 106](#)
 ConfigMaps [108](#)
 network policies [107](#)
 role-based access control [106](#)
 secrets [108, 109](#)
 service accounts [106, 107](#)
serverless architectures [137](#)
 benefits [137](#)
 community and governance [140-142](#)
 FaaS [138](#)
 managed serverless platforms [140](#)
 managed, versus self-managed serverless [139](#)
 notes of caution [139](#)
 open source serverless solutions [139, 140](#)
service controller [82](#)
service mesh [116, 117](#)
 Consul [118, 119](#)
 Istio [117, 118](#)
 Linkerd [118](#)
 Traefik Mesh [119, 120](#)
Service Mesh Interface (SMI) [43](#)
service models [8](#)
 Infrastructure-as-a-Service (IaaS) [8](#)

Platform-as-a-Service (PaaS) [8](#)
Software-as-a-Service (SaaS) [8](#)
single sign-on (SSO) [36](#)
Site Reliability Engineer [145](#)
Software-as-a-Service (SaaS) [8, 138](#)
software defined infrastructure [5](#)
Software Development Kits (SDKs) [153](#)
StatefulSets [87, 132, 205-207](#)
storage, in Kubernetes [120](#)
 PersistentVolume [121](#)
 PersistentVolumeClaim [122](#)
 StorageClass [120](#)

T

Telemetry [152, 153](#)
Thanos [161](#)
trace [153](#)
Traefik Mesh [119, 120](#)
Twelve-Factor App [129](#)

U

UNIX Time-sharing (UTS) namespaces [30](#)

V

Vertical Pod Autoscaler (VPA) [134](#)
virtualization [5](#)
 benefits [6](#)
 versus, containers [13](#)
virtualized container runtimes [32](#)

W

worker node, Kubernetes
 container runtime [83](#)

kubelet [82](#)

kube-proxy [82](#)