Kubernetes for Jobseekers

DevOps and Kubernetes interview questions and answers for freshers and experienced professionals

Shivakumar Gopalakrishnan

DevOps and Kubernetes interview questions and answers for freshers and experienced professionals

Kubernetes for Jobseekers

DevOps and Kubernetes interview questions and answers for freshers and experienced professionals

Shivakumar Gopalakrishnan

www.bpbonline.com

Copyright © 2023 BPB Online

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

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

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

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-519

www.bpbonline.com

Dedicated to

My beloved Parents:

Mr. A. Gopalakrishnan

Mrs. G. Rajalakshmi

&

My wife ***Asha****, My Sons* ***Nikhil*** *and* ***Adil****, and My Brother* ***Vijay***

Special thanks to Waffles for keeping me grounded throughout

About the Author

Shivakumar Gopalakrishnan has over 25 years of experience in Software Development, DevOps, SRE, and Platform Engineering. He has worked in wide-ranging industries, from a healthcare enterprise to a consumer-facing web-scale company. He founded a startup, was a key architect within a Fortune 1000 company, and was a manager/engineer in a Fortune 100 company. He has also given multiple presentations at DevOps/SRE conferences. He is the co-author of Hands-on Kubernetes on Azure.

He is currently an Individual Contributor at LinkedIn.

About the Reviewer

Werner Dijkerman is a freelance cloud, Kubernetes (certified), and DevOps engineer. He's currently focused on, and working with, cloud-native solutions and tools including AWS, Ansible, Kubernetes, and Terraform. He is also focused on Infrastructure as Code and monitoring the correct “thing” with tools such as Zabbix, Prometheus, and the ELK Stack, with a passion for automating everything and avoiding doing anything that resembles manual work. He is an active reader of comics, non-fictional and IT-related books, where he is a Technical reviewer for various books about DevOps, CI/CD, and Kubernetes.

Acknowledgement

This book would not be possible without the unconditional love and support of my parents, family, friends, and colleagues. I would also like to thank LinkedIn for allowing me to pursue my passion for creating economic opportunities for every member of the global workforce.

Thanks BPB Publications for being patient with my deliverables.

Preface

DevOps/**Site Reliability Engineering (SRE)** is one of the hottest areas in the **information technology (IT)** job market. With an average salary of $100K, it is a very lucrative field to be in. If you are passionate about delivering value to customers safely, securely, and quickly, it is a field that is ripe for innovation and, frankly, lots of fun.

DevOps/SRE (IT in general) needs to make great strides in the inclusiveness of women, People Of Color (POC), and other underrepresented people. This book aims to be a very small part of addressing the huge diversity gap. The book starts with the importance of mindset over skills. It makes clear to beginners of any age that, with the right mindset, they can master technologies that can seem overwhelming at first. The book guides the reader on how to use the complexity to their advantage. Combining practice with the lessons in the book, you will be able to nail the Kubernetes job interview and thrive in your dream job.

Computing basics are introduced via comics and links provided for learning more in a beginner-friendly fashion. How the basics of computing apply to containers, the basic building block of Kubernetes, is then presented. Hands-on lessons continue with Kubernetes deployments, tools, and pointers to advanced technologies such as service mesh, policy engines, etc. All major public cloud vendors, Amazon AWS (EKS), Azure (AKS), Google Cloud (GKE) Kubernetes implementations, and the differences are covered. Methodical processes for debugging and troubleshooting are continuously presented in the book for you to handle any scenario in production.

This book is divided into **15 chapters**. They will cover computing basics, networking, containers, why Kubernetes is the de-facto standard in container orchestration, methodical processes for debugging and optimization, public cloud implementations of Kubernetes, and advanced tools in the Kubernetes ecosystem. The details are listed below.

Chapter 1 will cover why a career in Kubernetes is lucrative and whether it is the right one for you. Different career paths (Individual Contributor vs.

Manager) are covered to help you determine the right path for you and the steps you need to take now to get there.

Chapter 2 will cover the key challenges enterprises currently face when they are looking at Kubernetes for solutions. You will also learn how to make the complexity of the Kubernetes ecosystem your advantage. You will be able to gain the confidence of the interviewer that you understand their problems and are aware of methods to help them succeed using Kubernetes.

Chapter 3 will cover how you can demonstrate that you fit the team's culture. The chapter also covers the importance of company culture and the key tenets of DevOps/SRE culture, how your daily activities, and having the right mindset relate to DevOps culture.

Chapter 4 will cover the basics of computer architecture, operating systems, and networks and how having a solid understanding of them would help you ace the interview.

Chapter 5 will cover the benefits and limitations of containers. Hands-on lessons for building custom container images that package your application, running, and testing them. How to manage and debug containers and container images is also covered. You will be able to answer basic and some advanced questions on containers/Docker and their relevance to Kubernetes.

Chapter 6 will cover how to launch an application using Kubernetes. You will also expose the service over the network, make it available externally, and attach persistent storage to your applications.

Chapter 7 explains how to find the contributing factors (aka Root Cause) for Kubernetes deployment failures. You will be able to explain the different deployment strategies, including their pros and cons. The advantages Kubernetes gives you in implementing the deployment strategy are also covered.

Chapter 8 covers why networking plugins are required and how to apply a network policy to provide access control to applications. The different types of storage and which storage is best suited for what applications are also covered. What GitOps is and how GitOps uses application/node management features of Kubernetes for deployments using code will also be clear.

Chapter 9 summarizes all the questions and answers to all the interview questions we have seen so far. Brushing up on this chapter and chapter 15 will make you ready for the interview by putting the answers to frequently asked interview questions at the top of your mind.

Chapter 10 covers the advantages and limitations of Windows support in Kubernetes. Questions on differences between Azure, AWS, and GCP Kubernetes implementations are also covered. You will have hands-on experience deploying and performing essential monitoring on AKS, EKS, and GKE.

Chapter 11 covers a framework to analyze performance issues methodically. The methodology works for any computer performance diagnostics, not just Kubernetes. You will also know how to fix Kubernetes performance issues using the top 10 performance optimizations for Kubernetes

Chapter 12 covers how to address any Kubernetes issue methodically and, thus, demonstrate your mastery of Kubernetes principles to the interviewer. After understanding and practicing this chapter, you can answer questions on Kubernetes troubleshooting questions confidently.

Chapter 13 covers the basics, plus hands-on experience on the popular tools for app management so that you can confidently answer questions on Kubernetes tools. After reading this chapter, you will have hands-on experience with app management tools such as Helm and Kustomize and awareness of similar tools such as Helmsman and Helmfile.

Chapter 14 covers the main categories of Kubernetes plugins, examples of them, and when/why you would use them. You will also gain exposure to a wide variety of plugins, and you will be able to make an intentional choice on which plugin you would like to become an expert in.

Chapter 15 summarizes the Kubernetes-focused interview questions and answers that we have seen so far. You will be ready for the Kubernetes-focused interview questions by brushing up on this chapter.

Code Bundle and Coloured Images

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

https://rebrand.ly/ud182ay

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Kubernetes-for-Jobseekers**. 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**.

Table of Contents

1. Kubernetes/SRE/DevOps Career Map
- Introduction
- Structure
- Objectives
- Career paths in Kubernetes
  - *Individual contributor (IC) path*
  - *Management path*
- Choosing the right career path for you
- Kubernetes' inclusive community
  - *Women/PoC in DevOps/SRE*
  - *Inclusiveness in the Kubernetes community*
- Conclusion
- Points to remember
- Interview questions and answers

2. Kubernetes Adoption in the Industry
- Introduction
- Structure
- Objectives
- Hybrid-cloud
- Kubernetes ecosystem challenges
- Conclusion
- Points to remember
- Interview questions and answers

3. Introduction to DevOps/SRE Culture
- Introduction
- Structure
- Objectives
- DevOps/SRE culture
- Key elements of DevOps/SRE culture
- What does DevOps culture mean to you?

Key elements for culture fit
Learning
Continuous improvement
Collaborations
Intelligent risks
Relentless focus on customer
Conclusion
Demonstration
Points to remember
Interview questions and answers

4. Operating System Fundamentals
Introduction
Structure
Objectives
Key components of a computer
Basic components of a computer
Introducing Profesora
Storage
Workspace
Process
Computer architecture
Try it!
Need for an operating system
Operating system concepts
Network
Exercise on networking
Applicability to kubernetes
Conclusion
Next steps
Points to remember
Interview questions and answers

5. Containers/Docker
Introduction
Structure
Objectives

- Processes and their relevance to containers
- Cgroups
- Introduction to containers
- Container registry
- Container networking
- Container storage
- Running a container
- Runtime configuration of a running container
    - *ENV variables*
- Building your own container image
    - *Why do we need layers?*
    - *Corollary*
    - *Why is this information important for Kubernetes?*
    - *Multi-stage builds*
- Container images versus running container instances
- Stopping and removing running containers
- Container image management
- Push container
- Conclusion
- Points to remember
- Interview questions and answers

6. Kubernetes Basics

- Introduction
- Structure
- Objectives
- Need for container orchestration
- Mesos, Docker Swarm, and Kubernetes
- Kubernetes primitives
    - *Pods*
        - *Declarative specification*
        - *Pod specification*
        - *Try it*
    - *Replica sets*
    - *Deployments*
        - *Rollout strategy*
        - *Version deduction*

DaemonSets
Services
Ingress controllers
Try it
Health checks using liveness, startup, and readiness probes
Try it
Physical volumes
Persistent volume claims (PVC)
Try it
Conclusion
Points to remember
Interview questions and answers

7. Kubernetes Deployment
Introduction
Structure
Objectives
Consumer expectations
Cost of doing business
Calculating the availability, you need
Kubernetes and deployments
Blue-green deployment
Try it
Canary deployment
Try it
Comparison of canary and blue-green deployment
Conclusion
Points to remember
Interview questions and answers

8. Kubernetes Services
Introduction
Structure
Objectives
Network plugins
Container network interface (CNI)
Networking policy

Try it
Storage
Main limitations of block devices
Advantages of attachment
Network-attached storage
Main limitations of NAS
Object storage
Physical volume claims
Node management
Try it
Configuration maps and SShh… secrets
Gitops
Try it
Conclusion
Points to remember
Interview questions and answers

9. Section Summary and Interview Questions and Answers
Introduction
Structure
Objectives
Section summary
Operating system fundamentals
Containers/docker
Kubernetes basics and primitives
Deployments using Kubernetes
Kubernetes services/features
GitOps
Interview questions and answers

10. Kubernetes on Various Platforms
Introduction
Structure
Objectives
Windows support
Azure—Azure Kubernetes Service (AKS)
Try it

AWS—Elastic Kubernetes Service (EKS)
Try it
GCP—Google Kubernetes Engine (GKE)
Try it
Raspberry PI cluster—for fun and profit
Raspberry PI
Running Kubernetes on Raspberry PI
Try it
Conclusion
Points to remember
Interview questions and answers

11. Kubernetes Performance Optimizations
Introduction
Structure
Objectives
Computer performance
Utilization Saturation and Errors (USE) methodology by Brendan Gregg
Kubernetes-specific performance tools
kubectl top command
Kubernetes dashboard
Top 10 performance optimizations
Symptom #1
Optimization #1
Symptom #2
Optimization #2:
Optimization #3:
Optimization #4:
Optimization #5:
Symptom #3
Optimization #6:
Optimization #7:
Optimization #8:
Symptom #4
Optimization #9:
Symptom #5

Optimization #10:
Conclusion
Points to remember
Interview questions and answers

12. Kubernetes Troubleshooting Tips
Introduction
Structure
Objectives
Kubernetes troubleshooting mental model
YAML problems
Troubleshooting pods
Get pod information
Get Pod definition
Double-check by using events
Logs, logs, logs
Pod configuration
Storage configuration
Verify using exec into the Pod
Flow chart summary
Pod troubleshooting conclusion
Troubleshooting services
Get service information
Get the service definition
Make sure the labels match
Make sure that the container port and the target port match
Troubleshooting Ingress
Get Ingress information
Get Ingress definition
Confirm that the endpoints are working
Get the Ingress Pod
Get the Ingress logs
Troubleshooting flow chart
DNS issues
Worst case scenario
Flow chart
Conclusion

Points to remember
Interview questions and answers

13. Kubernetes Tools and Extensions
Introduction
Structure
Objectives
Helm
Need for Helm
Helm repo
Sample deployment: Install WordPress
Kustomize
Helm or Kustomize?
Tools for tools
Helmsman
Helmfile
K9s
Awesome Kubernetes
Conclusion
Points to remember
Interview questions and answers

14. Kubernetes Plugins
Introduction
Structure
Objectives
Non-goals
Plugins
Service mesh
Why do you need a service mesh?
Istio
Honorable mentions
Network plugins
Cilium eBPF-based network plugin
Example usage
Storage plugins
Security plugins

Authorization plugins
Open Policy Agent (OPA)
Gatekeeper
Custom operators
VM management from Kubernetes using KubeVirt
AWS Fargate
Conclusion
Points to remember
Interview questions and answers

15. Kubernetes Questions
Introduction
Structure
Objectives
Interview questions and answers
Conclusion

Index

CHAPTER 1
Kubernetes/SRE/DevOps Career Map

Introduction

DevOps/**Site Reliability Engineering** (**SRE**) is one of the hottest areas in the **information technology** (**IT**) job market. A quick search on LinkedIn for DevOps/SRE results in thousands of job openings with an average salary of $100K* in the USA and as high as $181K* in San Francisco (**https://www.builtinsf.com/salaries/dev-engineer/devops-engineer/san-francisco**). These salaries are among the highest in this area.

People with Kubernetes skills are highly sought after in this highly rewarding and lucrative market. Their salaries are at least 38% higher than the average Systems Administrator salary of $64K* (**https://www.payscale.com/research/US/Job=Systems_Administrator/Salary**). These are just the starting salaries, and this initial jump in the starting salary makes a huge difference if you consider earnings over your lifetime. In this chapter, we will learn about the various career paths available for you, how to choose the right career path for you, and, if you have the right mindset, why Kubernetes is an excellent choice.

This book aims to make Kubernetes accessible to anyone of any age to obtain and thrive as a Kubernetes engineer.

All salary numbers are current as of March 2021 and are expected to vary wildly depending on enterprises' demand, location, and priorities.

Structure

In this chapter, we will discuss the following topics:

- Choosing among different career paths with a specialization in Kubernetes.
- Determining your career path for a fulfilling job experience
- Inclusivity problem in the DevOps area

- Inclusiveness of the Kubernetes community
- Next steps

Objectives

After studying this chapter, you can determine whether a career in Kubernetes is right for you. Have an idea of the two main paths of career progression that are equally lucrative. If this is your career, you should be confident that you will do well because of the inclusive community of Kubernetes.

Career paths in Kubernetes

There are the following two main career paths in DevOps/SRE/Kubernetes:

1. Individual contributor (IC) path
2. Management path

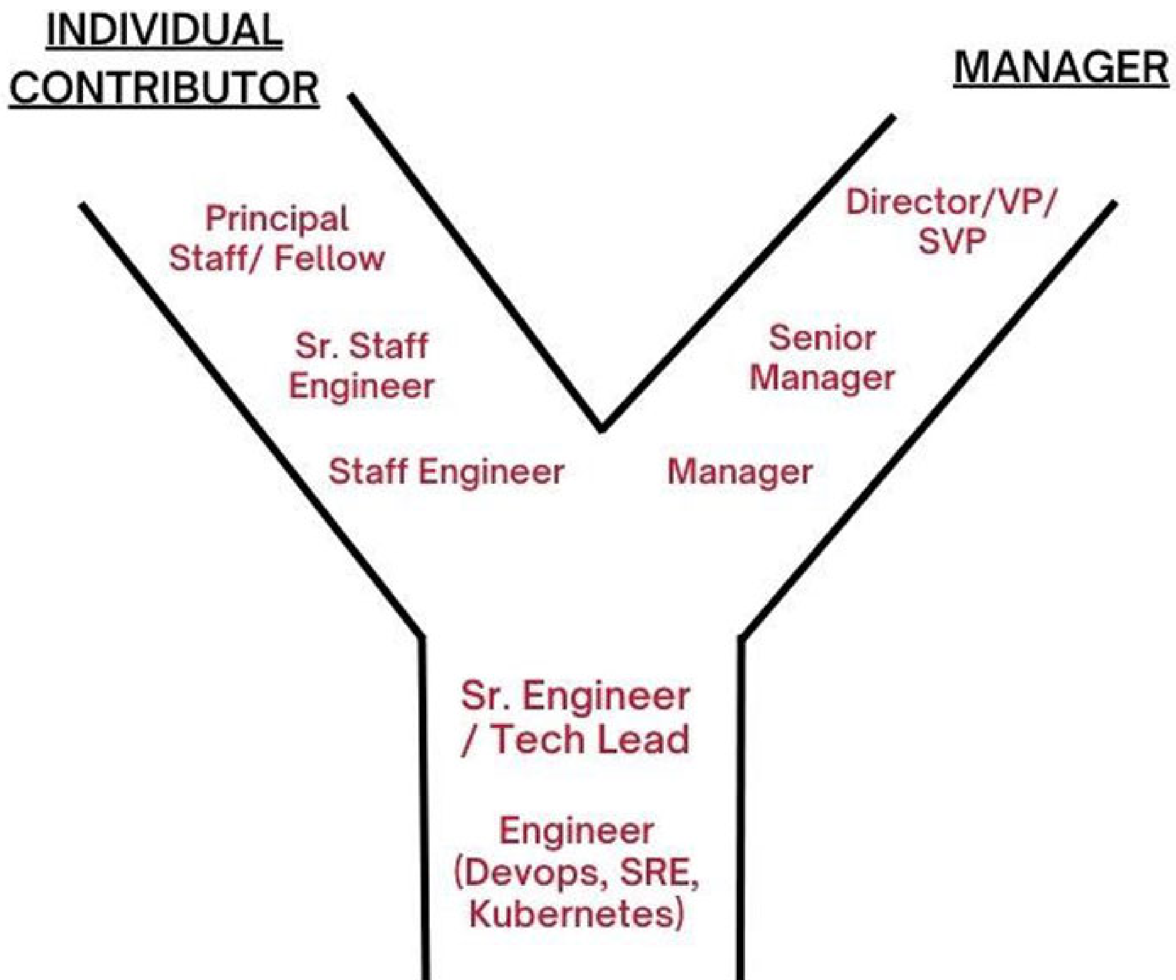


Figure 1.1: Two main paths of career progression

Individual contributor (IC) path

You focus on technical work, fixing things, and getting things done. While you are willing to coach and mentor your colleagues, you are not interested in managing people or project management. While eager to lead projects, you are not inclined to have people report to you directly.

The following table lists the roles for Kubernetes operation, maintenance, design, and implementation:

Role	Kubernetes Operator	Kubernetes Administrator	Kubernetes developer	Kubernetes architect	Kubernetes guru
Possible designation	DevOps/ SRE /Kubernetes Engineer	Sr. Engineer	Staff Engineer	Sr. Staff/ Sr. Architect	Principal Staff/ Fellow
Average salary (USA)	$100K	$180K	$220K	$250K	>>$300K
Key Responsibilities	Keep Kubernetes running. Be on-call for alerts. Participate in application deployments and upgrades, including Kubernetes upgrades.	Administration of Kubernetes. Ensure security, proper monitoring, regular upgrades, and educate teams on proper usage.	Understands the vision of the enterprise and implements it within a department by guiding a team of engineers.	A key contributor to setting the enterprise's vision and the plan for digital transformation. Helps in implementation within a location of an enterprise.	Setting the vision, evangelizing it across the entire enterprise, and getting the buy-in from all stakeholders (both management and tech leads).

Table 1.1: Possible individual contributor (IC) path

Management path

You are highly technical and, at the same time, want to help the company achieve more than you can do individually by leading teams. You like mentoring, helping people achieve their career goals, and interacting with other teams/leaders. You maintain a high standard of excellence and demand the same of those around you.

The following table lists the roles in the management path focusing on Kubernetes.

Role	Kubernetes Operator	Kubernetes	Kubernet	Kubernet	Director – Digital

		Administrat or	**es Manager**	**es Sr. Manager**	**transformation**
Possible designation	DevOps/SRE/Kubernetes Engineer	Sr. Engineer	Manager	Sr. Manager	Director/VP/SVP/CIO
Average salary (USA)	$100K	$180K	$220K	$250K	>>$300K
Key Responsibilities	Keep Kubernetes running. Be on-call for alerts. Participate in application deployments and upgrades, including Kubernetes upgrades.	Administration of Kubernetes. Ensure security, proper monitoring, regular upgrades, and educate teams on proper usage.	Sets goals for the team that is aligned with the overall vision. Create and assign tasks in JIRA/Project Management Tool. Performance reviews	Manage multiple teams. Set goals and communicate management strategy in technical terms to the teams. Present ideas from the team to management.	Implement enterprise-wide projects for the company in the Digital Transformation area.

Table 1.2: *Possible management path*

The roles and responsibilities of IC and Manager have many common traits, but the overall goal and experience are different, as shown below. IC works with a team, fixing issues, learning, mentoring, and ensuring that the team aligns with the manager and business goals. The manager acts as a servant leader leading by example, removing bottlenecks for the team, and ensuring that the team members work towards their personal and career goals:

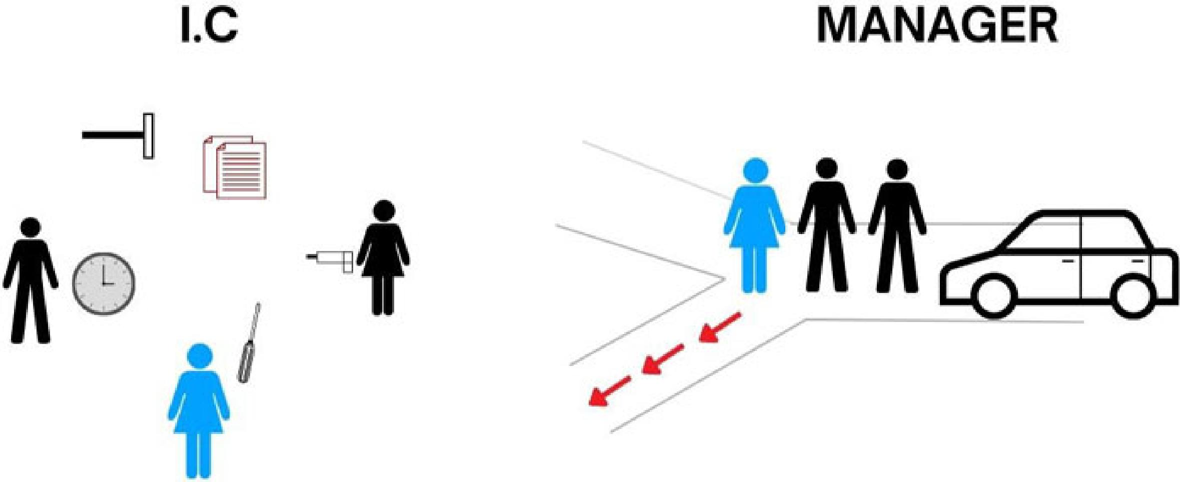


Figure 1.2: IC role versus manager role

I have worked as a Developer, CTO, Manager, Architect, and SRE at different career stages. I was lucky to have the choice of working in my areas of interest then (ranging from C++, RoR, networking, storage, and medical imaging to DevOps technologies such as Docker and Kubernetes). As I was working in my area of interest, I thoroughly enjoyed whatever role I was asked to perform, except when I was not performing to expectations as a manager. I had the opportunity to switch to an IC role with a lesser title, and I took it. I enjoyed the switch tremendously. Being flexible and focusing on the intersection of your strengths and the company/market needs can bring great satisfaction to your professional life. At a certain point in your career, fulfillment will matter more to you than titles.

Choosing the right career path for you

It is more important that you choose the right career path for yourself than choosing one that others think is right for you. You will spend most of your waking hours in your profession and must choose wisely. Another key factor that many forget is the opportunity cost. Imagine that you have 1,000 dollars. Using that money to throw a party means you have not invested that money in stocks that could have provided way better returns in the future. That is opportunity cost: investing in A means not investing in B. Although you can earn money back, you can never earn the time back. The opportunity cost for time lost is way higher than any other quantity.

To make the best use of your time, you must think like a startup where the product is you.

Figure 1.3: *Making your career choice*

The good news about Kubernetes is that people are willing to pay a lot. So, you have one concern down and two to go. Next is your interest/things you do well. As a reader of this book, that part is also covered. You like working with computers and are curious about what makes this whole internet thing tick. We are 2/3rd there.

The third part is crucial for success in this career path. Are you willing to persevere when the going gets tough? A crude way of looking at this but getting the point across effectively can be found in this article (**https://markmanson.net/life-purpose**) by Mark Manson, which can be summarized in a Safe For Work (SFW) fashion as "NOTHING IS EASY ALL THE TIME!"

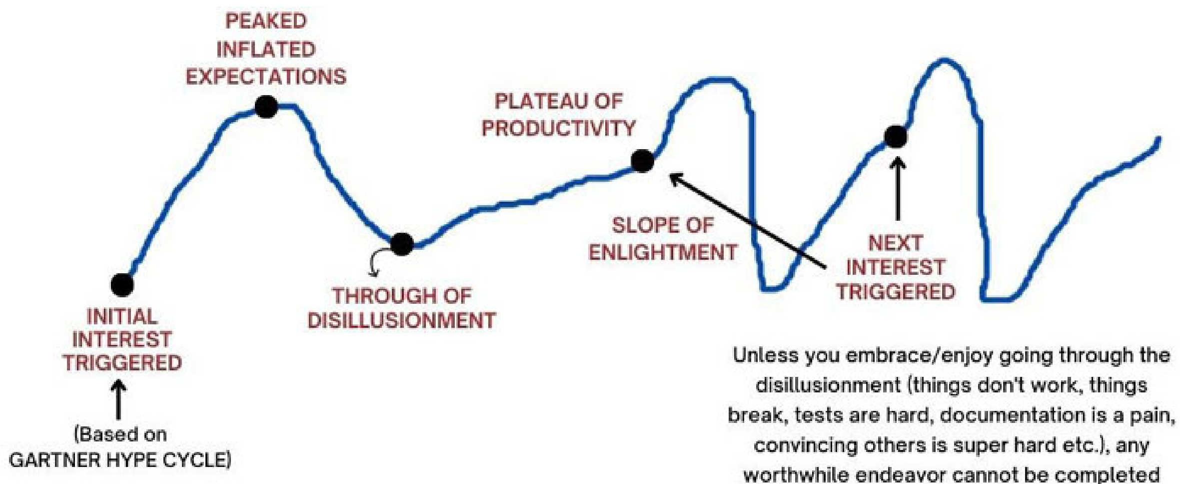


Figure 1.4: *Everything sucks, some of the time!*

To help you determine the following reasons why you *should not* choose Kubernetes are your career path. It may be surprising to see this in this book, though it is better for us in the long run. You would not be stuck in a job that you hate, and I will not feel guilty for pushing into a path that is not right for you.

- You do not like wasting time chasing errors down and getting frustrated when things do not work as they said they would.
- You do not like being up all-night fixing problems.
- You do not like reading or writing documentation.
- You do not like being in the hot spot when things go south.
- You do not like things failing and avoid them by taking no risks.
- You do not like frequent changes in technology.

Now that we have discussed why you should not choose a career path in Kubernetes, we can discuss why you should choose Kubernetes as your profession:

- You love the joy you experience when you see the features developed by your company reach its customer faster, safer, and better.
- You like learning about things you are not “good at” yet or areas that do not come easily to you.

- You treat failures as teaching/learning moments and are not afraid to take intelligent risks
- You love putting in the work to make systems resilient
- You get frustrated by manual/repetitive work and experience fulfillment by automating them.
- You like writing documentation/tests that make onboarding a smooth and joyful process.
- You do not like your team to be up all-night fixing issues and would rather have them work towards making systems that are self-healing.
- You do not like documentation that introduces the potential of manual errors and put in the work to eliminate the need for it.

In all cases, whatever path you choose will not be a straight path toward your goal. Be prepared and accept it to achieve success (whatever success means to you).

Every person's career path can be summarized as shown in the following:

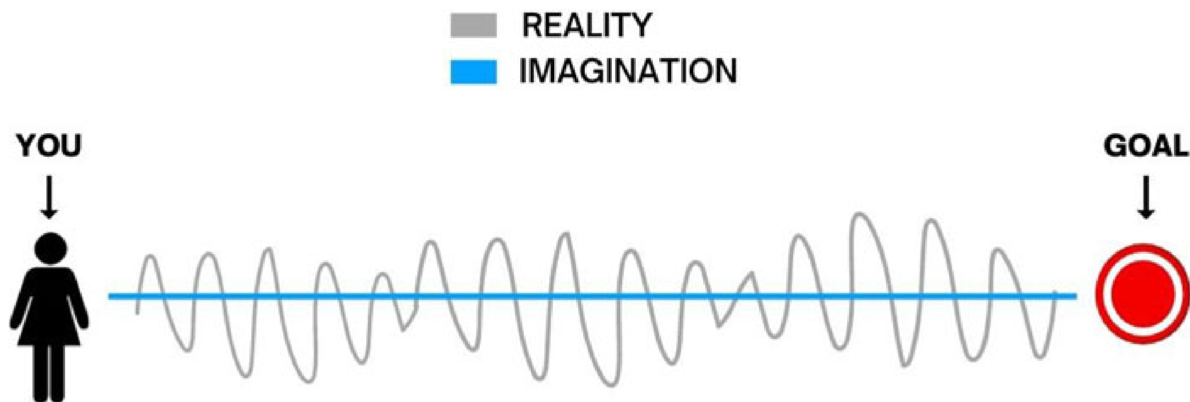


Figure 1.5: *Imagination versus reality*

Kubernetes' inclusive community

As you progress through the book or at any time during your career, you might feel overwhelmed or have impostor syndrome. The reassuring part about Kubernetes is that as a new technology, there are no *experts* in this field. Everyone, including me, is learning as we go along. We need solid fundamentals, a proper growth mindset, and grit to guide us. If you develop those, you are destined for success in this field.

Women/PoC in DevOps/SRE

Only 8% of DevOps/SRE engineers are women, resulting in the industry losing billions in revenue and intangible loss in creativity (**https://www.techrepublic.com/article/5-eye-opening-statistics-about-minorities-in-tech/**). The ratio is similar for Blacks and Hispanics in the DevOps world.

"Financially, a one percentage point move toward representative diversity leads to a three-point increase in revenue," Thibodeaux said during a keynote address at CompTIA's 2017 ChannelCon.

No one should feel excluded from thinking about a lucrative career in DevOps/SRE. The right mindset is all you need.

Inclusiveness in the Kubernetes community

Kubernetes was and is an inclusive community from the start. Kubernetes community has addressed the lack of diversity by acknowledging it head-on and dealing with it with humility that lots of work still needs to be done.

Excellent leaders like *Michelle Noorali*, *Diane Mueller*, *Divya Mohan*, *Eileen Teng*, *Ian Coldwater*, *Julia Han*, *Sabree Blackmon*, *Kelsey Hightower*, *Bryan Liles*, *Tameika Reed*, *Mutale Nikonde* (from **https://www.cncf.io/people/ambassadors/**) are few examples of pioneers who show that not only can you work in this industry, but you can also be leaders that inspire others to join this industry. These leaders have made Kubernetes the huge success it is today and continue to lead the way for enterprises.

You cannot find a better community than Kubernetes, where diversity in opinions is tolerated and encouraged.

Conclusion

This chapter helps you to convince yourself that a career in Kubernetes is what you would like to do and is a very lucrative and fulfilling profession.

In the upcoming chapter, we will see the tremendous adoption of Kubernetes in the industry. You will also learn why adoption has opened multiple opportunities for you.

Points to remember

- Kubernetes offers a lucrative path towards having a choice in the work you want to do and enjoy.
- Starting as a Kubernetes engineer, you can choose at the appropriate time whether to go into management (Manager, Director, VP, and CEO) or thrive as an Individual Contributor (Developer, Architect, Staff, Fellow, and Distinguished Scientist).
- A Kubernetes career requires—a growth mindset and the grit to persevere through failures—bias toward action and taking intelligent risks.
- It is a great career for you if you are a fan of automation and interested in making companies achieve their digital transformation goals.
- Choosing what career is right for you is more important than choosing what everyone thinks is the right career.
- Kubernetes promotes inclusivity at all levels and is an excellent choice for anyone of any age, including women, people of color, and any other underrepresented minorities.

Interview questions and answers

The answers are suggestions; unlike technical questions, there are no correct answers.

1. **Why do you want to become a Kubernetes engineer?**

   I like fixing issues and enjoy being part of the team that helps developers deliver features to our customers faster, safer, and better. Kubernetes enables developers by taking care of security, operability, and observability issues.

2. **Can you give me an example of where you fixed issues that improved the situation?**

   If you have experience in the IT field:

   Give an example where you helped find an issue with a software release or production issue. Interviewers also like to hear where you had a non-technical solution to a problem. One example of mine is where to solve customer issues quickly, and I suggested having a dedicated chat room where assigned developers (we called them Superheroes) would be watching. Anyone in the field could ask them questions.

   If you do not have experience in the IT field:

Interviewers are looking for people with the right attitude. Have an example ready with a challenging issue you helped solve. Preferably, it would be close to a technical area, but that is unnecessary. An example could be fixing the radio or a clock at home. Another could be helping a family member solve a tricky issue, like helping your mother talk to a person far away via video chat when your mother was very worried about meeting that person and could not meet them (very relatable during the COVID lockdown). The more authentic it is, the better.

3. **Where do you see yourself in five years?**

   Depending on the career path [manager or **Individual Contributor** (**IC**) role], imagine yourself coming to the office five years from now and describe the tasks that you would be doing ideally. For an Architect, it would be proposing architectural changes that would build on the success you were instrumental in achieving and incorporate the latest advancements in the field. You would also present this proposal to other teams to get their buy-in.

CHAPTER 2

Kubernetes Adoption in the Industry

Introduction

By 2030, there will be tens of millions of job openings, with a high percentage requiring advanced IT skills.

Kubernetes has become the de facto standard for container orchestration. It is widely adopted across enterprises from tech companies (for example, Airbnb, Alibaba, AWS, Apple, AT&T, CISCO, Fujitsu, Google, Microsoft, Reddit, Salesforce, Shopify, Spotify, Twitter, and VMware) to financial companies (for example, ANT financial, Bloomberg, CapitalOne, Citi, Fidelity, Goldman Sachs, Intuit, Morgan Stanley, and Nasdaq) to health care (for example, Babylon Health, United Health Group, and Siemens Healthineers). The market adoption rate is at an all-time high of 48% (2021) and is growing at 25% per year.

Kubernetes is an important tool in digital transformation. According to a 2021 study by PureStorage (**https://www.purestorage.com/content/dam/pdf/en/analyst-reports/ar-portworx-pure-storage-2021-kubernetes-adoption-survey.pdf**), 68% increased their usage of Kubernetes while they expect to reduce costs by 30%. Their survey also showed that people familiar with Kubernetes tend to earn higher salaries!

You are at the right time to join the Kubernetes movement. Kubernetes is only around seven years old (compare it to UNIX, which has been around 50 years, and Linux, which has been around 30 years). Large enterprises are confused and trying to figure out how to deliver their software using cloud-native computing, which they know is the future. Enterprises desperately seek to hire people like you who can provide expert guidance through this complex landscape.

The complexity comes from multiple dimensions, each providing you with an opportunity to become an expert in that area and, thus, highly sought after.

Structure

In this chapter, we will cover the following topics:

- Hybrid cloud—Challenges and path to success for enterprises.
- Kubernetes ecosystem challenges—What they are and how to overcome them.

Objectives

After reading this chapter, you will know the key challenges enterprises currently face when they are looking at Kubernetes for solutions. You will also learn how to make the complexity of the Kubernetes ecosystem your advantage. You will be able to gain the confidence of the interviewer that you understand their problems and are aware of methods to help them succeed using Kubernetes.

Hybrid-cloud

Enterprises have invested billions into data centers/on-prem computing with tons of traditional bare metal servers and virtual machines. Although moving the apps themselves is challenging, data is an entirely different animal. The data is their crown jewel, and it has its own gravity. Even though it can be more expensive than running your own data center in terms of pure acquisition and infrastructure costs (servers, power, and space), enterprises realize that they cannot keep up with the maintenance challenges. A simple security hack can bring their entire enterprise down. Having people with enough expertise is also a big challenge for them (which is, of course, good news for you).

What are the enterprises looking for in people about the hybrid cloud?

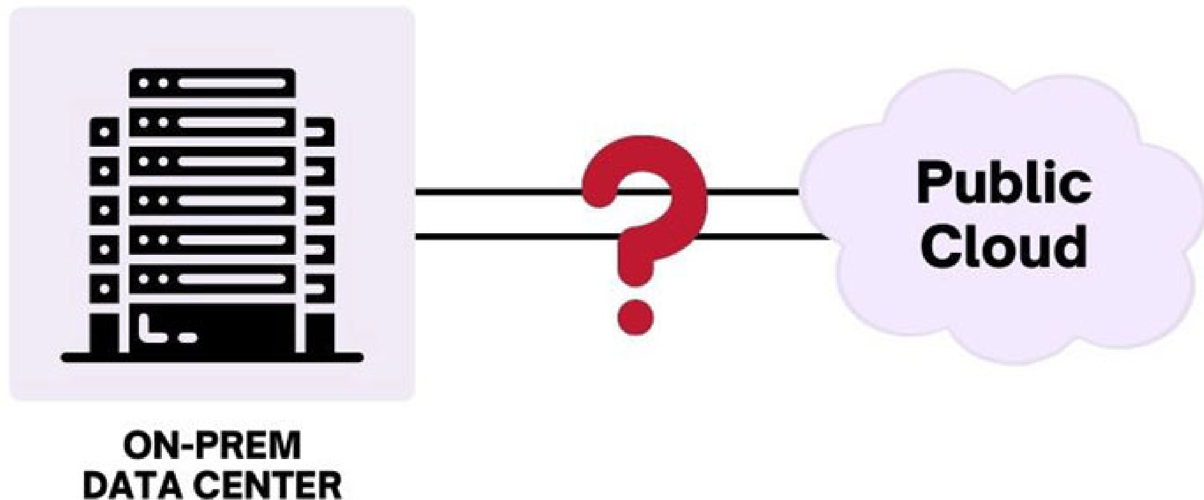


Figure 2.1: *Challenge #1: hybrid cloud*

Recognize that on-prem applications and data are the ones that are *paying the bills* for the *fancy* new kids on the block, such as Kubernetes:

- Knowledge of the different approaches for hybrid cloud implementation.
- Ideally, hands-on experience, but knowledge would also do on how to *Walk, Crawl, and Run* toward cloud-native applications.
- Knowledge of what technologies are stable and what are not in this rapidly evolving Kubernetes landscape.

Kubernetes ecosystem challenges

I remember attending the first KubeCon in San Francisco, with around 500 attendees. The last one I attended in 2019 in San Diego had 12,000. The 2020 virtual conference had approximately 23,000.

This tremendous growth has the unfortunate side-effect of having too many options. CNCF alone has 15 graduated projects (**https://landscape.cncf.io/?project=graduated**) (from just 1 when it started). As part of the Linux Foundation, Cloud Native Computing Foundation (CNCF) is the hub for open-source cloud native computing projects. They famously host Kubernetes and Prometheus, among others. They promote cloud-native solutions to allow all companies, including non-IT companies, to compete better, and deliver value faster.

CNCF also provides certifications such as **Certified Kubernetes Administrator** (**CKA**), **Certified Kubernetes Application Developer** (**CKAD**), and **Kubernetes Certified Service Provider** (**KCSP**). Having CKA on your LinkedIn profile and resume would put your application on top of the heap.

CNCF landscape (https://landscape.cncf.io/) has around 416 products (including proprietary and open source) at the time of writing and is expected to grow.

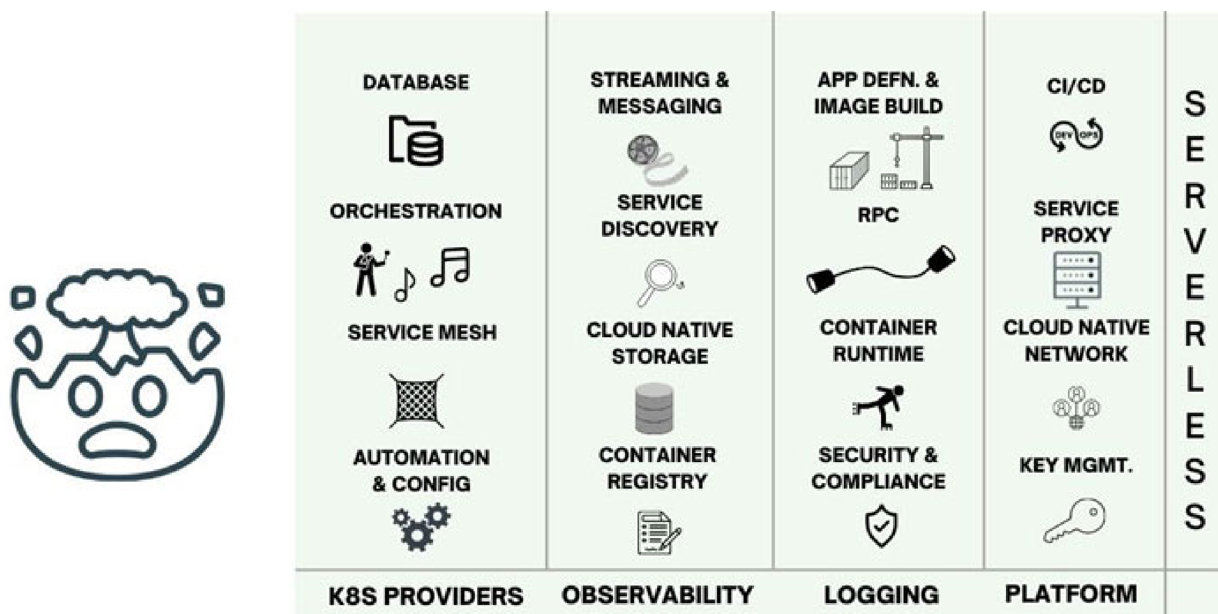


Figure 2.2: *Challenge #2: the difficulty of navigating Kubernetes due to its complex ecosystem*

This huge Kubernetes (useless Trivia: Kubernetes is Greek for helmsman—the person who holds the steering wheel of a ship/boat in layperson's terms) ecosystem has put enterprises in the unenviable position of figuring things out for themselves. It is like the early explorers trying to navigate to India without any maps or guides.

What are the enterprises looking for in people concerning the Kubernetes ecosystem? Let us have a look:

- Awareness that Kubernetes has a complex ecosystem.
- Awareness of hosted Kubernetes in a public cloud (Azure, AWS, and GKE) versus self-hosted.

- Awareness of the key areas of the Kubernetes ecosystem (CI/CD, storage, network, and provisioning).
- The knowledge that Kubernetes has widespread applications in machine learning (Kubeflow).
- Need to establish and follow a process for adopting Kubernetes that aligns with the "Crawl, Walk, Run" strategy for cloud-native applications.

Be aware that the overwhelming feeling you have right now is an incredible opportunity. Imagine enterprises with billions at stake if they make the wrong move or, worse, not move at all. They are looking for you to implement their path forward. The objective of this book is that by the end, you will have answers to all the preceding questions.

Conclusion

There is no better time than now to get into the Kubernetes ecosystem. A growing number of enterprises, currently at around 48% and growing, are looking for you. Kubernetes is only seven years old; it is rapidly evolving, and there are no *experts* with many years of experience in the field.

This need, coupled with the inclusive nature of the Kubernetes community, will generate huge wealth for you and the entire world. The complexities involved in digital transformation require people like you to navigate them to make their applications cloud-native through the crawl, walk and run approach.

By the end of this book, you will have the answers to the questions that enterprises have and be on your voyage to be an expert in this area. In the upcoming chapter, you will learn:

- What do they mean when they say, *Culture eats strategy for breakfast*?
- What is DevOps/SRE culture?
- DevOps culture—is it aligned with your values?
- If it is, how to demonstrate a culture fit with the company you are interviewing?

Points to remember

- Kubernetes is the de-facto standard for container orchestration.
- Used across various industries, from tech to finance to auto to health care.
- High demand for folks who know Kubernetes due to its newness, widespread adoption, and high complexity.
- A hybrid cloud, which uses both on-premises and public cloud data centers, is one big challenge for enterprises.
- The second challenge is managing applications across clusters with the right combination of tools available in the complex Kubernetes ecosystem.

Interview questions and answers

1. **What is the de-facto standard for container orchestration? Why?**

   Kubernetes is the de-facto standard for container orchestration. Although good solutions for container orchestration like Nomad and Mesos exist, the market has overwhelmingly voted with its feet and dollars toward Kubernetes. Having an organization like CNCF assures enterprises that they are not beholden to a single company and can confidently integrate with their in-house systems. All cloud providers providing managed Kubernetes as their offering enables enterprises of any size to dip their toes into Kubernetes without huge upfront investment.

2. **Is it possible to have ten years of hands-on experience in Kubernetes (as of 2023)?**

   Kubernetes was released to the public in 2014 and has gained stability and adoption only in the past five to six years. So, 10 years of hands-on experience in Kubernetes is not possible.

3. **What are the primary challenges that enterprises face when implementing Kubernetes?**

   The main challenges that enterprises face are as follows:

   - How to implement a hybrid cloud? It is not a good idea to migrate completely all on-premises applications to the public cloud.

- How to abstract Kubernetes complexity from developers by leveraging the multiple tools/methodologies available from the Kubernetes ecosystem? (aka *Day 2 problems*)

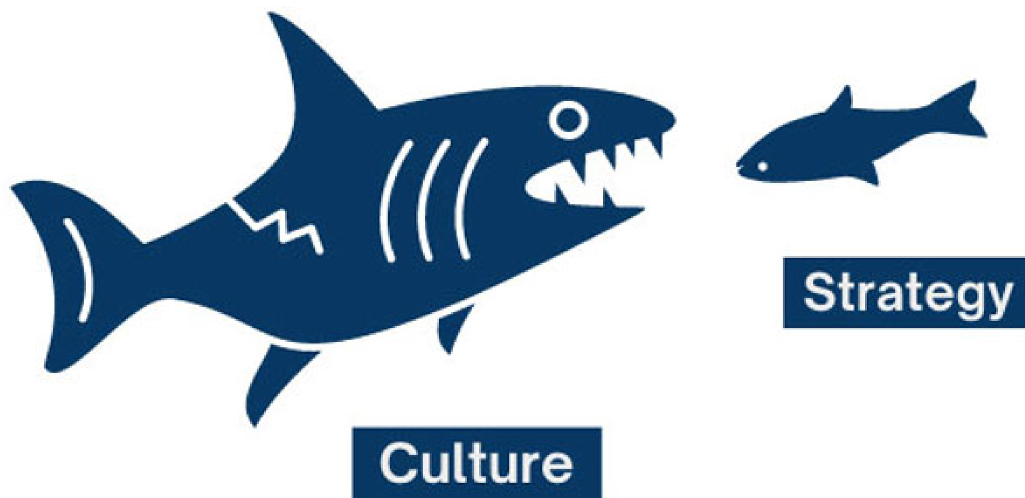
4. **They say that Kubernetes is the answer to all my deployment issues. What is your opinion on that?**

   Although Kubernetes is a critical tool in a company's digital transformation, it is not a panacea. It brings high complexity to deployments and may not be suitable for small organizations. Even with hosted Kubernetes solutions, it requires dedicated teams for day-to-day operations and customizing tools for improved developer experience and site reliability.

CHAPTER 3
Introduction to DevOps/SRE Culture

Introduction

Peter Drucker, attributed with the quote *Culture eats Strategy for Breakfast*, was a premier management consultant and is considered to have laid the philosophical and practical foundation of a modern enterprise. The *Culture eats Strategy for Breakfast* quote is illustrated in *Figure 3.1*:



Culture Eats Strategy

Figure 3.1: *"Culture eats strategy for breakfast."—Peter Drucker*

The quote is a warning to leaders that unless the organization's culture is aligned with the strategy, the execution of any plan is doomed to failure. The culture of an organization is the values that the employees consider important. If collaboration is not considered important, demonstrated by the organizations by promoting people who get things done but do not play well

with others, any strategy that requires collaboration between departments is bound to fail.

In this chapter, you will learn about how important an organization's culture is to your success and the enterprise's success. The key elements of DevOps/SRE culture (including what DevOps/SRE means) and how it needs to align with your own values.

Structure

In this chapter, we will cover the following topics:

- DevOps/SRE culture—the origin story and definition
- Key aspects of DevOps/SRE culture
- Is DevOps for you?
- How to demonstrate culture fit to the interviewer

Objectives

After reading this chapter, you will be able to demonstrate that you fit the team's culture. You will be able to do it by knowing the importance of company culture and the key tenets of DevOps/SRE culture. You will know how your daily activities and having the right mindset relate to DevOps culture.

DevOps/SRE culture

In the beginning, there were developers and operations people. Developers of the "cool" stuff then threw their developed product over the wall to operations. Any issue in deployment, maintenance, and day-to-day operations was considered an "operations" problem. *Figure 3.2* illustrates the throw-it-over-the-wall practice in product development*:*

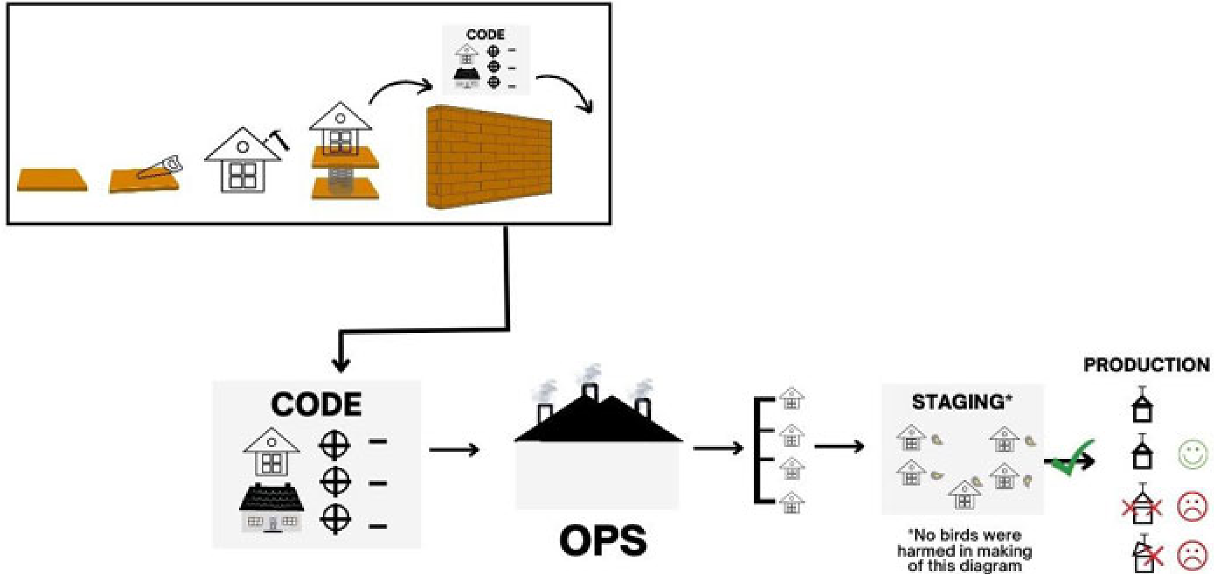


Figure 3.2: *It's Ops problem Now*

This way worked fine when developers were developing using the waterfall approach for development, where the releases were done only after a year or so. Although it introduced discipline in terms of listing the requirements and having an architecture and design before development, it came with a very long time (years) between releases.

Agile development methodology was adopted to address this challenge which put testing as the key part of developer responsibility. Collaborations between developers and testers became essential in improving product quality and customer experience tremendously.

While Agile methodology made software releases faster (months instead of years), it needed to scale up faster for Web-based companies.

There was a collective jaw drop when Flicker announced in 2009 that they were doing 10 deployments per day to production (totally eclipsed by 23,000 deployments per day at Amazon in 2020). Companies that can address customer needs as fast as necessary (sometimes slightly more) were destroying competing companies (think Blockbuster, K-mart).

The collection of cultural practices that help companies deliver at speed was grouped under the term DevOps. Dev + Ops means Developers and Operations People collaborating closely as *equal* partners. Figure 3.3 illustrates developers and operation engineers working together*:*

***Figure 3.3:** Dev and Ops*

Key elements of DevOps/SRE culture

What is the key element of DevOps? Although many might focus on tools such as Kubernetes and Docker, DevOps culture is key to success. What are the key elements of DevOps culture?

- **Learning**: Organizations assume good intent from all and treat all failures, big and small, as learning opportunities. The corollary is that organizations remove people who relish finger-pointing, shooting the messenger, and do not take responsibility when bad things happen.
- **Continuous improvement**: Organizations do not rest on their laurels and continuously, relentlessly improve the process of delivering value to their customers. It encourages feedback, even negative ones, as long as it comes with an idea for improvement. Suggestions are ranked on merit, with preference given to people closer to the problem. (This means the suggestions from the top will have the least weight by definition.)
- **Collaborations**: Organizations value and interest in establishing collaborative environments. Organizations reward people/teams who work with each other by increasing their funding. Conversely, teams who refuse to play along (with careful consideration given to teams who want to disrupt the status quo for the customers' benefit) are defunded.
- **Intelligent risks (courtesy: LinkedIn)**: Organizations encourage people who take intelligent risks with the potential for greater rewards. The "intelligent" part comes where risking customer/employee safety, or information security is a strict no/no. Skipping a long approval process to try an experiment should be treated as an intelligent risk and is not punished.

Employees' psychological safety is highly valued when team members express ideas freely without fearing ridicule, being put down, or being dismissed. Organizations encourage people who voice their unpopular opinion against groupthink without risking their careers.

The blameless retrospective is another key practice that successful DevOps organizations follow to make it a practice to learn from mistakes, especially when made while taking an intelligent risk.

- **Relentless focus on customers**: Delivery value to the customer is established as the top priority, and anything that hinders that is ruthlessly deprioritized.

  While the end customers might be the ones that most of us think of, it could be teams that depend on you and their most important employees. Employees' well-being and betterment result in an engaged employee who delivers tremendous value to the customer.

  Figure 3.4 illustrates DevOps culture:

Figure 3.4: *DevOps Culture*

What does DevOps culture mean to you?

How do the ideals for an organizational culture that wants to reap the benefits of DevOps relate to you? Culture can also be viewed as mindset, and strategy can be viewed as your plan/skills. The importance of mindset over skills is shown in *Figure 3.5*:

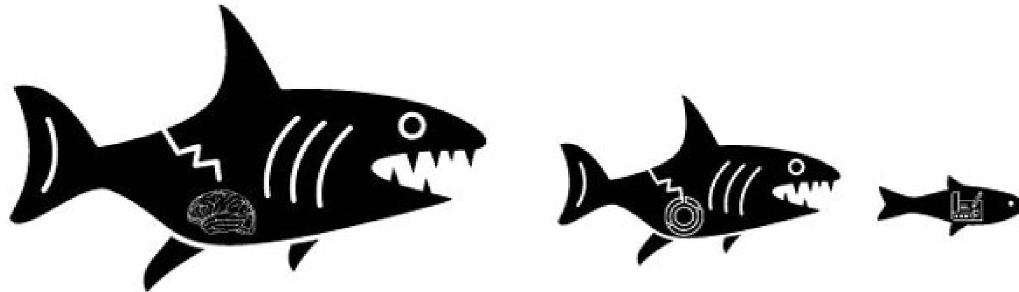


Figure 3.5: *Mindset > Skills > Plans*

To paraphrase Peter Drucker:

"Mindset eats skills and plans for breakfast."

Employers are looking for people with the right mindset so that the employee will not only survive but thrive in their company. Employers/hiring managers specifically reject people they think will not fit their company culture independent of their candidate's skills/experience in the field.

During the interview, if they see you as someone aware of the DevOps/ SRE cultural practices, it will give you a leg up over someone who might be more skilled.

Key elements for culture fit

A growth mindset (the belief that one can master any skill given interest and grit. One does not have to be *naturally gifted* to excel in an area) is key to demonstrate during the interview. Being open to learning constantly and continuously improving are key elements of the growth mindset.

Learning

The companies and the interviewers are looking for the demonstrated ability you focus on learning from your mistakes and as a person who is not afraid to fail. The evidence can come from courses you took, especially in fields that

are not in your comfort zone (public speaking is something I need to take). Evidence of curiosity can be shown by how your initiative found out about how the zipper worked.

Demonstrating that you have a growth mindset can be done by showing how you improved through continuous improvement. Referencing interesting books, YouTube videos, and blogs that promote a growth mindset is a plus.

Continuous improvement

Any process or product can be improved. When asked about your achievements, you could demonstrate your improvement-focused mindset on how you did not stop at achieving the goal but continued to fine-tune it by continuously improving it.

Collaborations

The person interviewing is not just focusing on your technical capability but whether they would enjoy sitting next to you while working on problems. They are also looking for whether you can help them with *boring* tasks such as setting up meetings with other teams, making presentations, and taking meeting notes. Demonstrating that you are more than willing to take one for the team goes a long way in getting to the next round of interviews.

Collaboration is a skill that takes on continuous practice and can be done by starting small, like sending weekly updates. Having a mentor would also greatly help for communicating with a larger audience than you are used to. The key skill is providing enough context in your communication with the right amount of information. (Goldilocks approach—not too much, not too little, just right)

Trusting that your coworkers have good intentions and are not there to bring you down is a very important attitude to have. The alternative way of thinking results in needless escalations and wasteful cycles spent on finger-pointing and blaming, which could have been spent on solving the actual problem.

Real-life experiences where you reached out to clarify a misunderstanding (when you thought clearly that the other person was on the wrong side), even in your personal life, would let the interviewer know you are a team player.

Intelligent risks

*The brick walls are there for a reason. They're not there to keep us out. The brick walls are there to give us a chance to show how badly we want something. (*Refer to *Figure 3.6)*

— *Randy Pausch of the Last Lecture fame* (**https://www.cmu.edu/randyslecture/**)

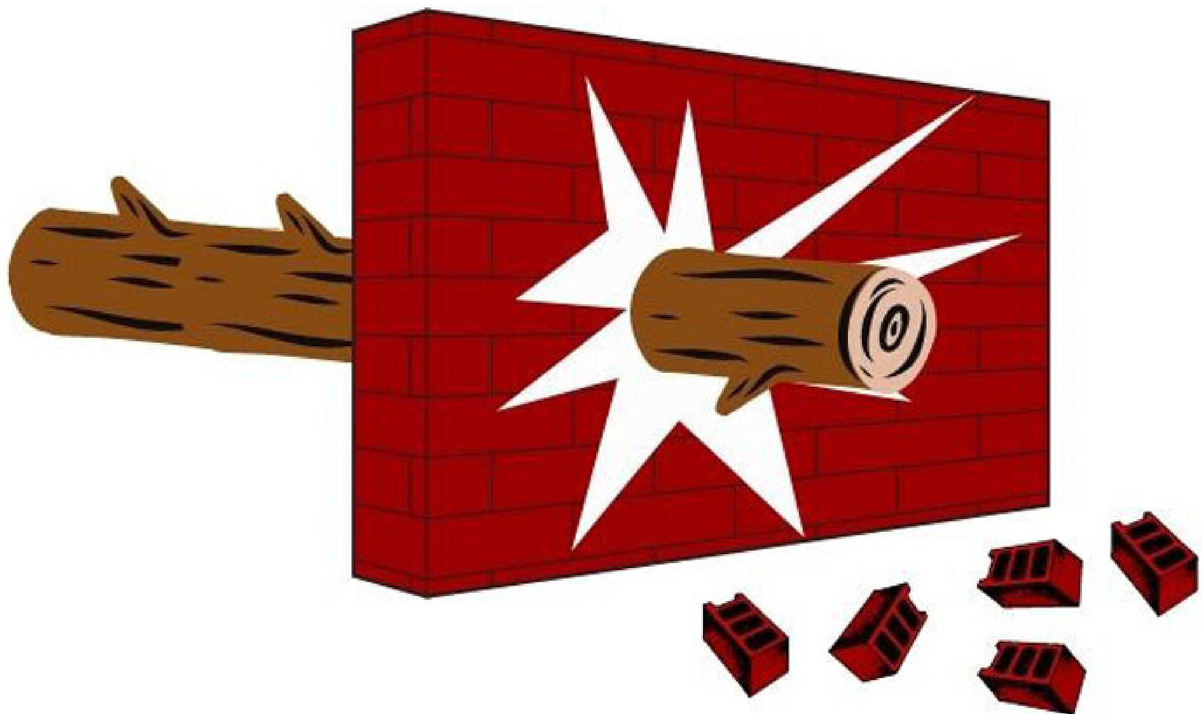


Figure 3.6: *Break the Wall of NO*

Demonstration of corrective ways you overcome red tape or things people said were "*impossible*" is what the interviewer is looking for in this area. Have some anecdotes ready for this question. I quote when I did the demo using my AWS servers to demonstrate a feature when getting a physical server at five different geographical locations would have been very hard.

Telling stories from your life where you provided cover for someone who wanted to try something new would also get your brownie points.

Relentless focus on customer

Corralling your team or your friends toward an objective of making your customer (whether they are your parents, employee, or community) happy is

very good evidence of you putting the customer first.

All the other mindset values are for delivering impactful value to customers safely and as quickly as possible. An empathetic person who can understand the customer's needs (not necessarily what they want) by putting themselves in their shoes is what my team would love to have.

https://hbr.org/2011/08/henry-ford-never-said-the-fast—Apparently misattributed a quote to Henry Ford, often used to differentiate between customer *needs* and *wants*. The quote is *If I had asked people what they wanted, they would have said faster horses*.

Side note: According to the article, improving the speed of delivery at a lower cost (DevOps anyone?) was the reason for the initial success of Ford.

Steve Jobs is also known for ignoring Apple's focus groups while bringing out innovative products. While most mp3 players of that time were going for more features (Radio, fancy display, and so on), Steve Jobs went for the simplest of features, 1,000 songs in your pocket with only one button and a dial as the interface (iPod). Then went totally in the opposite direction to bring the iPod.

The key, as in anything, is to strike a balance between listening to your customers and anticipating what they need. More importantly, keep experimenting to get quick feedback on what is accepted and what is not.

Conclusion

Culture eats strategy for breakfast. IT organizations are investing in a culture that prioritizes their customer experience and employees' well-being. The point was reinforced during the COVID pandemic. DevOps culture takes those key tenets as the key part of the implementation, and interviewers are looking more for culture fit along with skills in hiring.

In the upcoming chapter, you will learn computer, operating systems, and networking basics. This will set a solid foundation as we learn how advanced tools like Docker/Kubernetes work.

Demonstration

A growth mindset is critical to getting hired in a DevOps/SRE-focused organization. You can learn the growth mindset by practicing it in your daily life. These practices would provide evidence of your growth mindset to your interviewer.

In 8th grade, my dad asked me to carry a heavy desk up a carpeted stair. Independent of how hard I tried, I could not get the table up. Finally, totally exhausted, I told my dad that it was *impossible* for a single person to carry this table up the stairs. He asked me to think it over, and after some grunting, the answer was the same from me. *It is impossible*. He then suggested what if you turn the desk upside down. It was an Aha moment, and it was quite easy to push the desk up the stairs. The lesson I learned that day is that when a problem is tough to solve, see if you can reframe it by, say, inverting it. Refer to *Figure 3.7*:

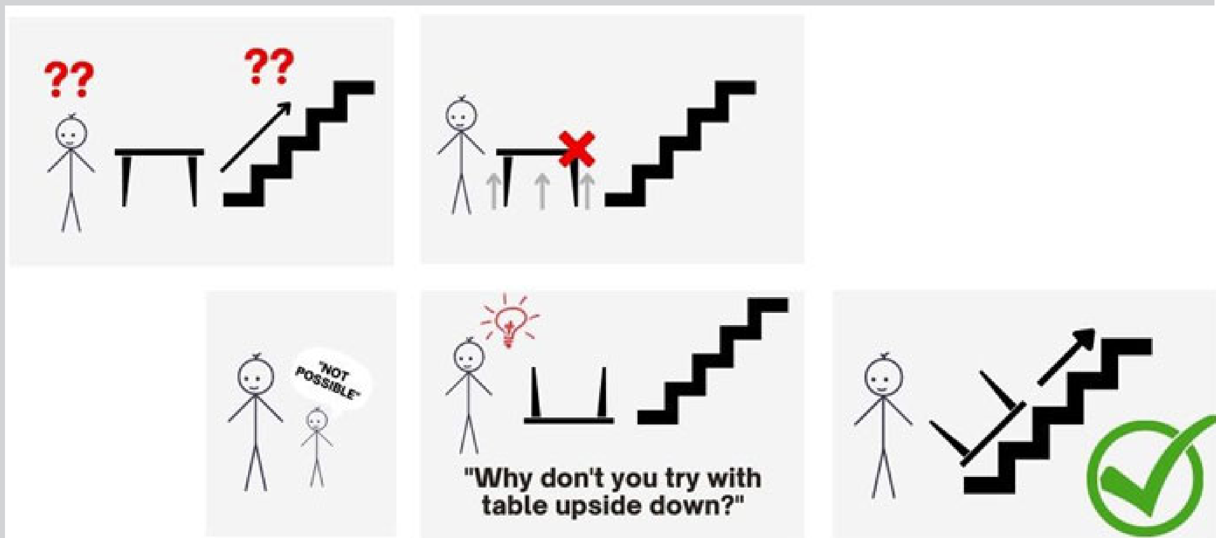


Figure 3.7: *Everyday DevOps Practice*

Another example was independent of how much I tried, and I could not get the network cable to reach a desktop server. My good friend Ronald suggested we turn the server around to be closer to the cable source. It worked!

I learn from these personal experiences to see if we can skip a hop to improve performance or use event-driven architecture instead of spending too much time getting good performance out of a database.

Learning to enjoy the journey is more important than focusing on the destination. While you will learn the skills in the coming chapters, practicing learning with curiosity on how Kubernetes works, continuously improving Kubernetes skills, and teaching others through open-source tutorials, LinkedIn posts, and so on will provide ample demonstratable evidence for you. Enjoying the time when you overcome the struggle of learning a new concept is what will keep you going and getting the job. Clicking through the pages and doing the exercises will get you to the end of the book. If you treat each page with a childlike curiosity, the journey of experiencing this book will be joyful and ultimately get you the job you desire.

Points to remember

- *Culture eats strategy for breakfast*—Peter Drucker: This is an important factor to be aware of in your job and job search.
- Culture fit keeps you engaged and thus ensures your well-being and thriving professionally at your workplace.
- DevOps/SRE culture brings developers and operators together to deliver customer value as safely, securely, and quickly as possible.
- The key elements of DevOps/SRE culture are learning, continuous improvement, collaboration, intelligent risks, and relentless focus on customers (including internal customers).
- DevOps mindset is more important than skills and tools. You can practice key elements of DevOps culture in your personal life by bringing curiosity into everything you do.

Interview questions and answers

1. **Why is the culture of a company important?**

   A company's culture is an excellent predictor of the people it attracts and how excited you will be to show up Monday morning. Having culture fit determines how engaged the employees would be and, in turn, the company's success.

2. **What is DevOps?**

   DevOps is a movement that brings developers and operations closer together to deliver value to the customer.

3. **I am hiring an engineer for my DevOps team. Is having a DevOps team sufficient to implement DevOps?**

   DevOps team focusing on deployment and site reliability issues to develop expertise and customize solutions to the enterprise/developer needs is essential but insufficient. Support for the shared responsibility and development of resilience in personnel and software is required from the top and the bottom.

4. **What are the key elements of DevOps/SRE culture?**

   The key elements are focus on learning, continuous improvement, collaboration, taking intelligent risks, and, most importantly, relentless focus on customers.

5. **A good interviewer would give more importance to your mindset or skills.**

   While demonstrated skillset would trump other factors in general, given that two candidates have a similar skillset, the interviewer would give more importance to the interviewee's mindset. The person with the growth mindset would be chosen.

6. **Do you need to be in a company to practice DevOps principles?**

   No. DevOps principles of having curiosity, continuously learning, and making improvements can be done when doing daily chores. You can also practice honoring commitments and solving problems without blaming interactions with your family.

7. **Can you give examples of when you practiced DevOps principles?**

   Be prepared with examples where you demonstrated curiosity and improvements to reducing toil. Although professional examples are ideal, you can use personal experiences also and might be remembered more than professional ones.

8. **What is the difference between DevOps/SRE/Platform Engineering?**

   Like this chapter, engineers, managers, and companies use DevOps/SRE/Platform Engineering interchangeably. The simple reason why we use it interchangeably is that the interviewer most likely uses it interchangeably also. The difference is documented well by Marco Schwarz in his post (**https://www.linkedin.com/posts/marco-schwarz-**

gcp_devops-infrastructure-sre-activity-7000704285139214336--oXb). The summary is:

- **DevOps** – Culture empowers development teams with more control over shipping code to production. The focus is on getting the code shipped.
- **SRE**– Practices that keep services up and running. It includes monitoring that gives feedback to developers on areas to focus on. The focus of SRE is keeping production software healthy.
- **Platform Engineering**– It is the underlying basis for both DevOps and SRE. It focuses on developing an ecosystem enabling Developers and SRE to work efficiently.

The entire chapter can be summarized as *Be Curious*, as shown in *Figure 3.8*.

Figure 3.8: *Be Curious*

CHAPTER 4
Operating System Fundamentals

> **"Do one thing. Do it well."**
>
> *—UNIX philosophy*

Introduction

In many interviews, I have seen many people with years of experience not knowing about the basic components of a computer. When they mix up **Random Access Memory** (**RAM**), measured in GB, and hard disk storage, also measured in GB (especially on phones), the interview is almost over at that point. They must climb up a steep hill from there to gain confidence that they can perform well as an engineer where details matter.

In this chapter, we will be learning about the following:

- Basic computer architecture
- Key computer components
- Operating system concepts

Structure

In this chapter, we will discuss the following topics:

- The key components of a computer.
- Why an Operating System (OS) is needed.
- What is an OS?
- The primitives on which the OS operates.

Objectives

At the end of this chapter, you will know the basics of computer architecture, operating systems, and networks and how having a solid understanding of

them would help you ace the interview.

Key components of a computer

When we talk about computers now, it is no longer only the stodgy mainframes that fill rooms, rows, and rows of servers in a data center or even your laptop or desktop computer. They are everywhere, from digital signboards, tablets, phones, kiosks, and POS (Point of Sale) machines to medical devices. Independent of their form factor, the basic components are the same.

Basic components of a computer

Computers exist to compute. We give the computer input, and it computes and gives an output. *Figure 4.1* shows the input being processed by the computer and generating output.

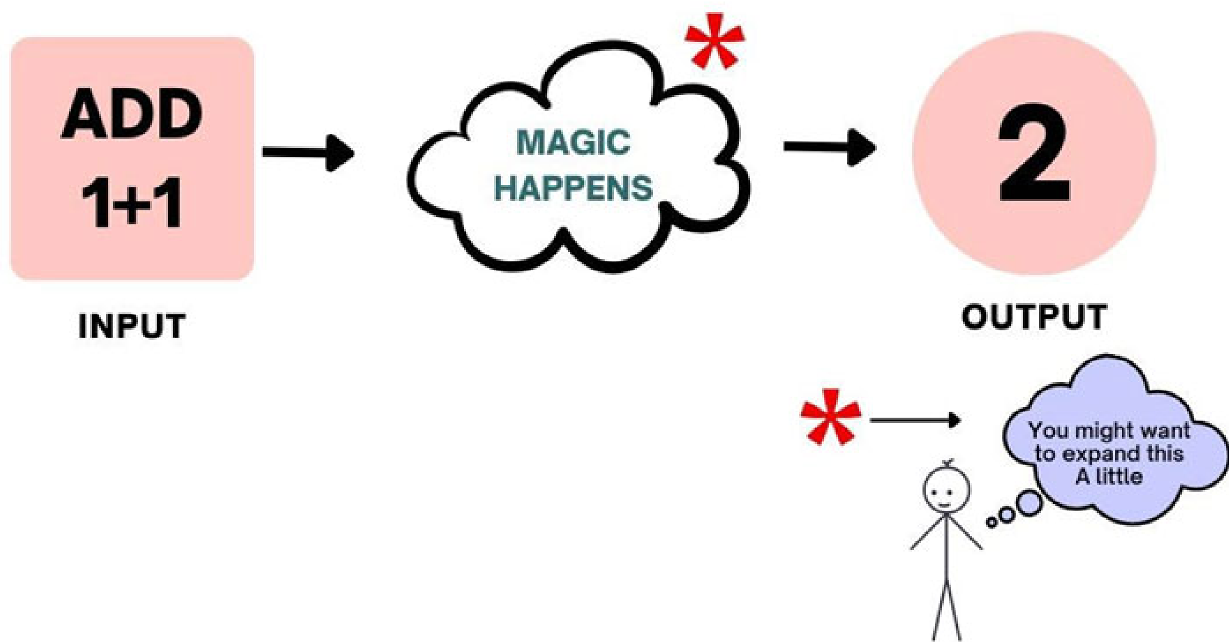


Figure 4.1: *Computing in a Nutshell (Input->Magic Happens->Output)*

Let us consider an example by taking it step-by-step.

Introducing Profesora

Figure 4.2: *Profesora—Master Planner, Guide, and Mentor*

Her goal is to get you a job as a Kubernetes engineer in *Dream Company, Inc.* She will get you in as a special chef inside their cafeteria (no relation to the Money Heist series).

Your code name is **Tokyo**.

Figure 4.3: *Tokyo: You—Aspiring Kubernetes Engineer*

The first mission is to follow this recipe for Bread Toast.

- Get two slices of bread from the loaf on the shelf.
- Put the slices of bread on the toaster.
- Keep the bread loaf back and mark how many slices are left in the bread.
- Get the jam bottle out.
- When the bread pops out, apply a layer of jam on one of the slices.
- Cover the bread slice with the other bread slice and place it on the serving plate.

You grumble about this meaningless task that has nothing to do with the *cool* stuff, such as docker, but read the instructions and execute it anyway to make the perfect bread toast.

Let us review what you just did:

Figure 4.4: *Input->Your Magic->Output (Bread Toast)*

Bread and Jam were inputs. Instructions were the process documented, also known as the program. Output was one delicious strawberry jam toast. *Figure 4.5* illustrates the process in the form of a flow chart:

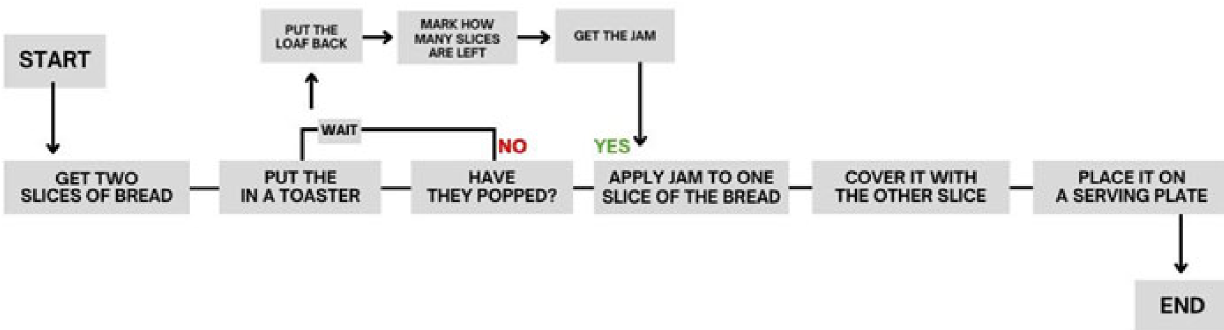


Figure 4.5: *The "Program"*

That is exactly what a computer does. Let us dig deeper into the requirements to make that magic toast happen.

Storage

Storage is where all the ingredients you need as a chef are kept. It is also where we keep an inventory record so that when we come the next morning, we know what to order next. It is a bigger place with lots of ingredients but getting things from there takes time. *Figure 4.5* shows a restaurant's physical pantry storage, illustrating its large capacity but difficulty in getting items quickly:

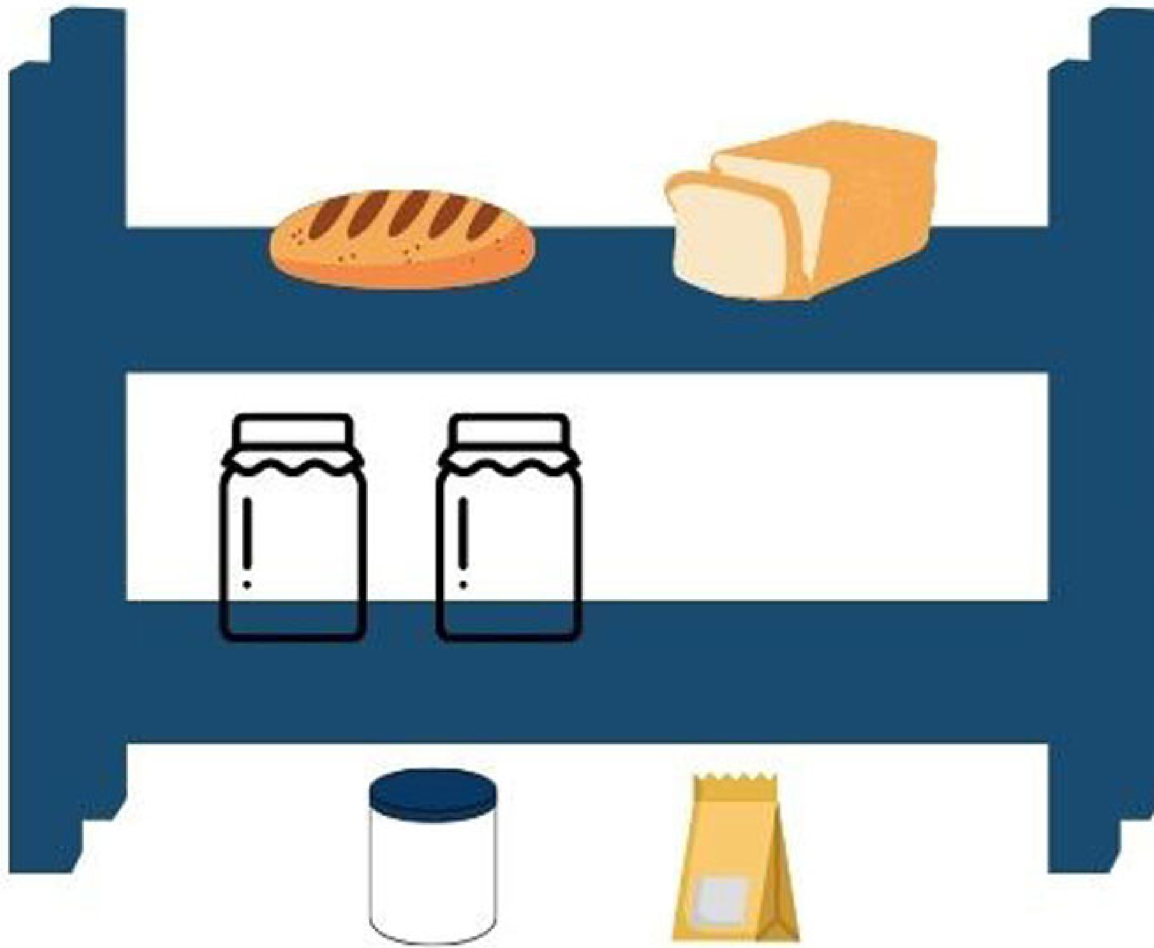


Figure 4.6: *Storage*

Workspace

Taking bread loaf and jam out of the pantry storage takes time, so we need a temporary workspace to make the sandwich. It has limited space, but you can access items randomly much faster than getting them from storage. Keep only what you need in the workspace because the space is limited. *Figure 4.7* shows the limited space available in the workspace.

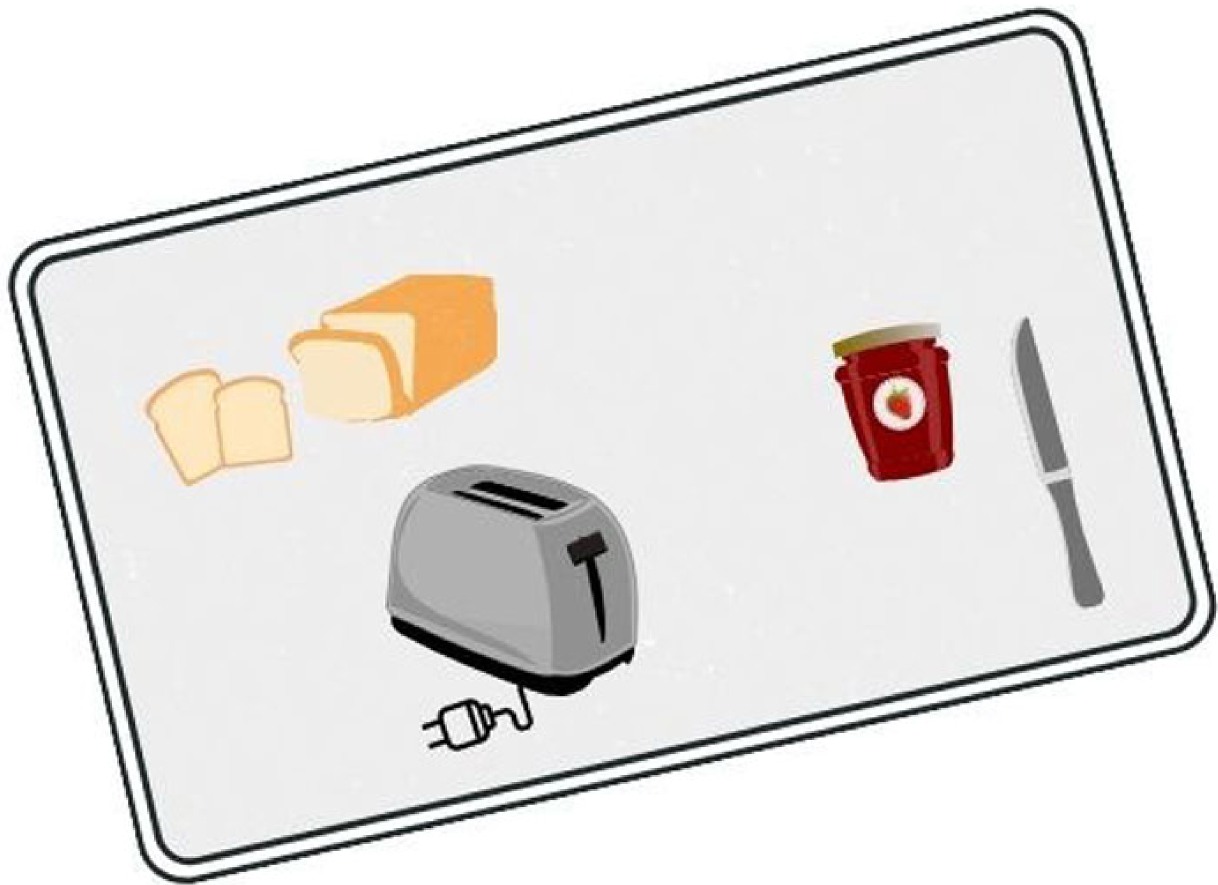


Figure 4.7: *Workspace*

Process

Finally, this is where the magic happens. You are following the instructions and processing the ingredients to make the sandwich. You also update the inventory that is in the storage based on how much you took out. *Figure 4.8* shows you making the toast while accessing the storage and the workspace:

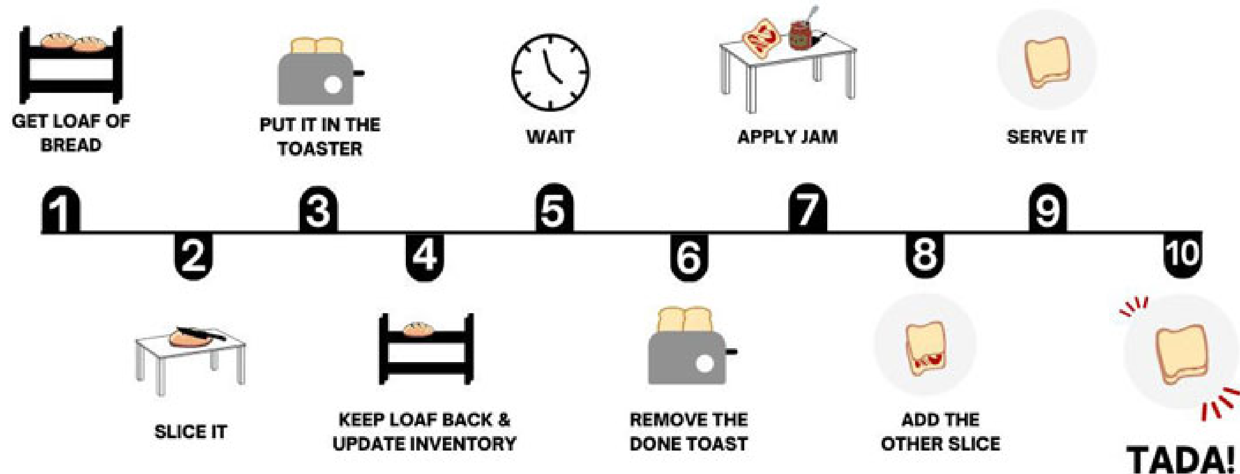


Figure 4.8: *Processing following the "Program"*

Computer architecture

A computer takes inputs using Input/Output devices (I/O or simply IO) and processes the instructions. It uses its **central processing unit** (**CPU**), leveraging **random access memory** (**RAM**) for fast access of temporary data and persistent storage devices such as hard disks or media for storing and retrieving data that needs to be persistent.

As you can see, the faster the CPU is faster it can process the instructions and is limited only by the speed at which it can access the hard disk or RAM.

RAM is measured in GB (nowadays, 1–2 GB is common in phones, and 8–64 GB is normal on desktops). Even though they are quite large from what they were a few years ago, it is still not as large as the hard disk. Hard disk size is generally measured in hundreds of gigabytes and even terabytes (1 Terabyte [TB] =1,000 gigabytes). RAM loses all its information when the power is cut off, whereas persistent storage devices do not. RAM is the *workspace* for the computer where it places the things it needs to be handy.

Storage is measured in hundreds of gigabytes to terabytes and has lots of space to store data but is slower than RAM in access speed. The access speed of a hard disk (even with much faster **Solid-State Disks** [**SSDs**]) is measured in milliseconds versus nanoseconds for RAM. *Figure 4.9* shows

the computer components and the CPU’s interaction with I/O, RAM, and storage.

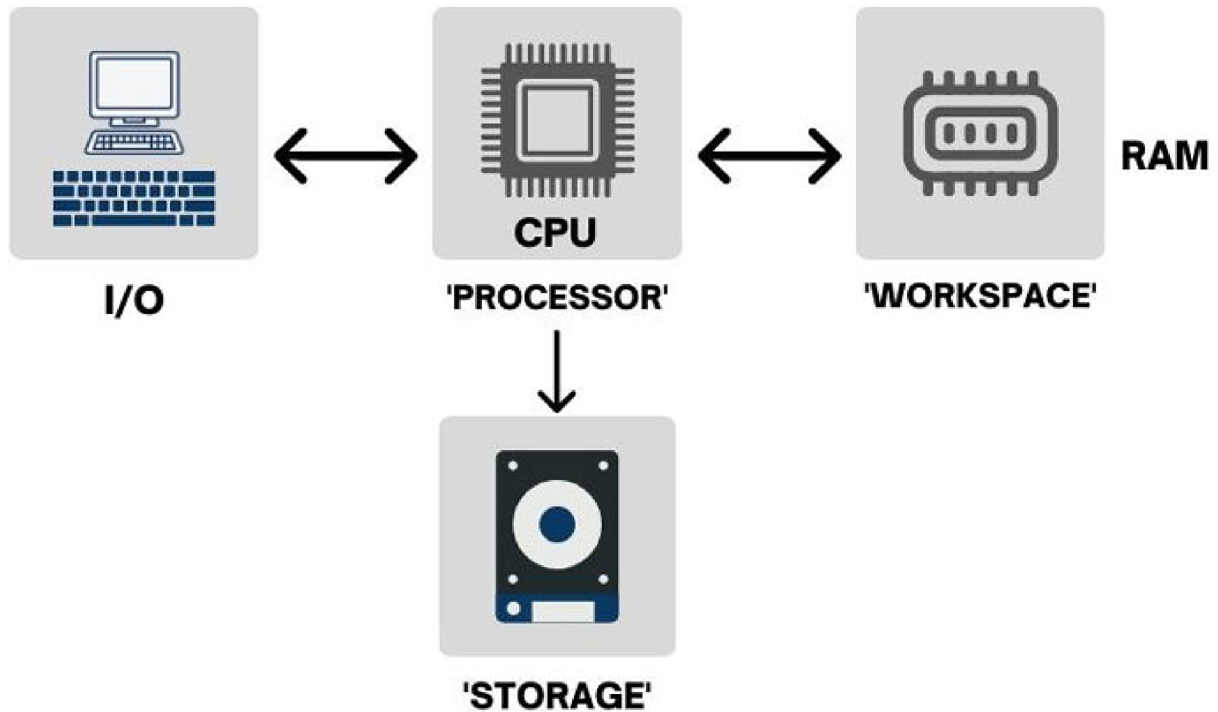


Figure 4.9: *Computer components*

Profesora’s notes:

- The monitor or screen is just an output device. It is never the computer.
- A phone with 64 GB specification has 64 GB built-in (disk storage), not RAM, which will be in the 2–4 GB range.
- CPUs have cache memory (in megabytes), so they can access data faster than RAM.

Try it!

Open a desktop computer if you can or look it up on the Web and identify the storage, RAM, and CPU.

With more and more use of phones/laptops, few have seen the insides of a computer. When there is an issue with computers, it generally involves these

three: CPU/RAM/Storage (with one more coming up soon). Touching the hardware physically gives you a better connection/understanding of how things work inside a computer.

Getting a cheap computer like a Raspberry PI (https://www.raspberrypi.org/) lets you get a great feel for a *real* computer where you can see all the components.

Need for an operating system

How the requirements for making a toast and the requirements of running a program match are illustrated as follows:

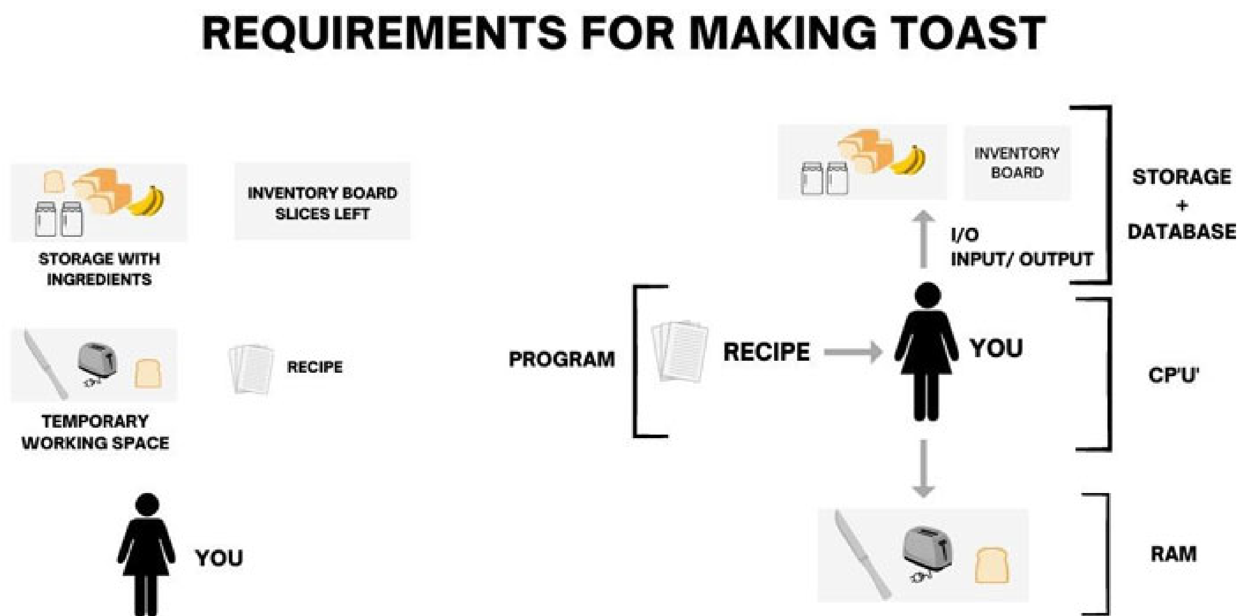


Figure 4.10: *Mapping computer components/process of making a toast*

Profesora has a new assignment for you. She wants you to make French Toast with butter and a thin coating of brown sugar on top of the jam. She also says she would ask for plain toast or a toast with jam whenever she feels like it and wants you to deliver fast.

You try making the toast but get interrupted by her next order, which asks for a toast with different ingredients. With your previous training, you know how to build a single sandwich quickly, but handling the requests and fetching ingredients takes too much time. You get stuck in getting the ingredients while you could be putting the brown sugar topping resulting in the toast becoming cold by the time you get to it.

You ask for an assistant. She gives you **Oslo**:

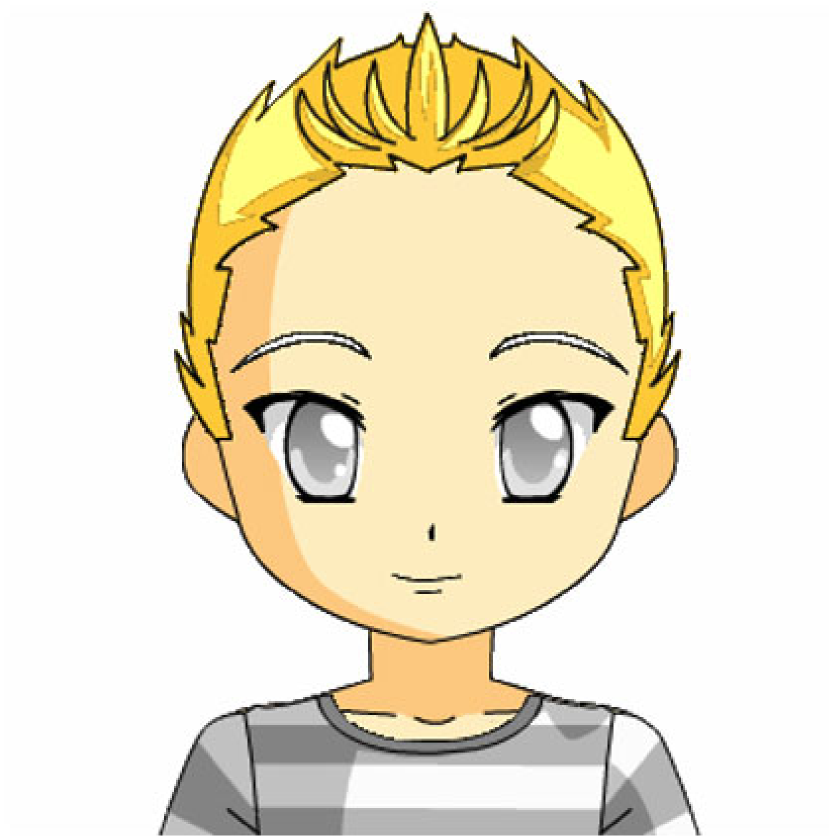


Figure 4.11: *Oslo—Your assistant*

Oslo is the right assistant for you. He puts all Profesora's requests in a queue and fetches the ingredients when you ask for them at the right time. He gives you the instructions for the next and sets the plates out with the finished toast. *Figure 4.12* shows Oslo at work:

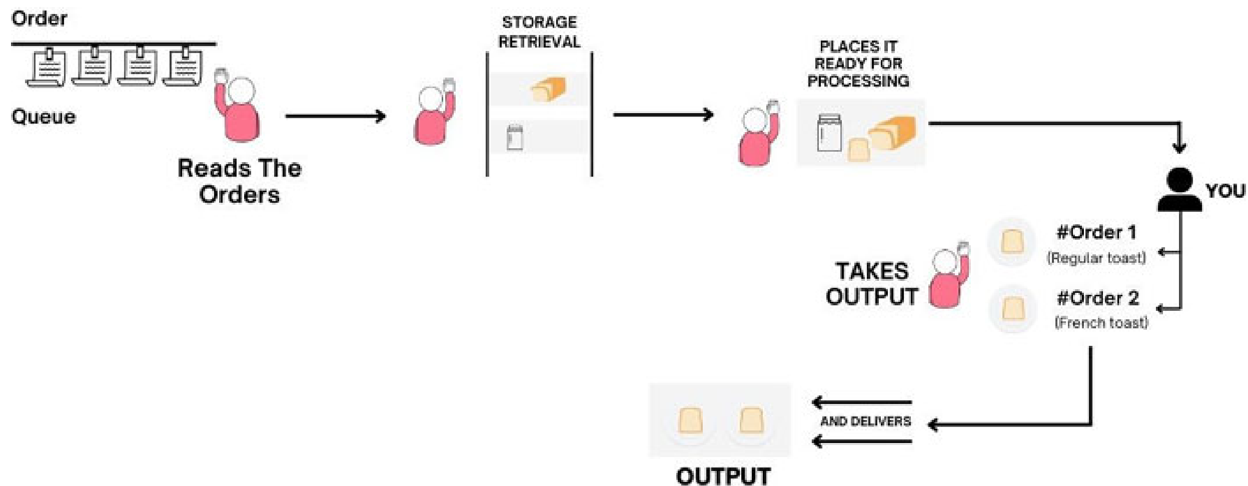


Figure 4.12: *Oslo at work*

Profesora is getting her toasts just how she likes it and as quickly as possible. Let us see what Oslo is helping you with:

- Let you focus on processing each order.
- Fetches the ingredients from the storage.
- Puts the finished toasts out for consumption.
- Queue all the orders so you can focus on processing the ingredients.

Without Oslo, you cannot process the orders efficiently. Oslo takes care of all the logistics making you the best chef you can ever be. You can even say Oslo is operating the entire system☺.

Operating system concepts

An operating system consists of software that orchestrates all hardware functions to work together as a cohesive system. The basic unit of work, just like Tokyo processing the order, is called (no points for creativity here) the **process**. The process takes the instructions in the program and is scheduled to be run by the operating system, just like the orders for the chef.

And as the program runs, operating systems abstract away the exact details of the hardware to fetch data from storage, manipulate data in RAM, and finally, send the data to the output.

Oslo takes care if they use a different shelving system or a different set of plates. Tokyo remains oblivious to the changes in the backend and can focus on the task at hand. Tokyo and Oslo use well-established way of communication that has been used for decades so that Tokyo can talk to Oslo (fetch me two slices of toast) anytime, even though the restaurant could use a completely different storage system. You might even say Tokyo and Oslo talk using a portable system interface. The official term in OS is **Portable Operating System Interface** (**POSIX**—established in 1988). *Figure 4.13* shows how OS acts as a glue between various hardware components that make up a computer:

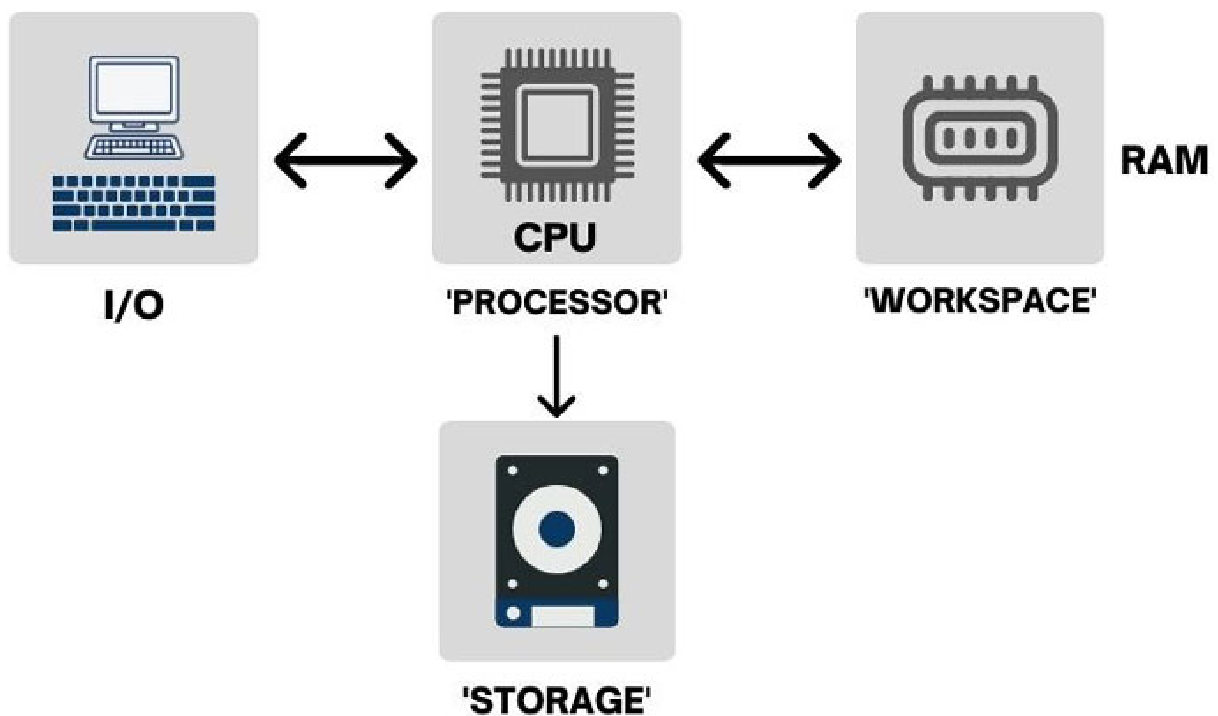


Figure 4.13: *Operating System (OS) interacting with I/O, RAM, and Storage*

Operating systems deal with storage using a file system. The file system abstracts the underlying storage mechanism on how and where the data is stored. OS says, "tell me where the `inventory.txt` is," and this file system gives the handle to the OS that the OS can use to read and update the contents. *Figure 4.14* shows how OS interacts with the file system (FS):

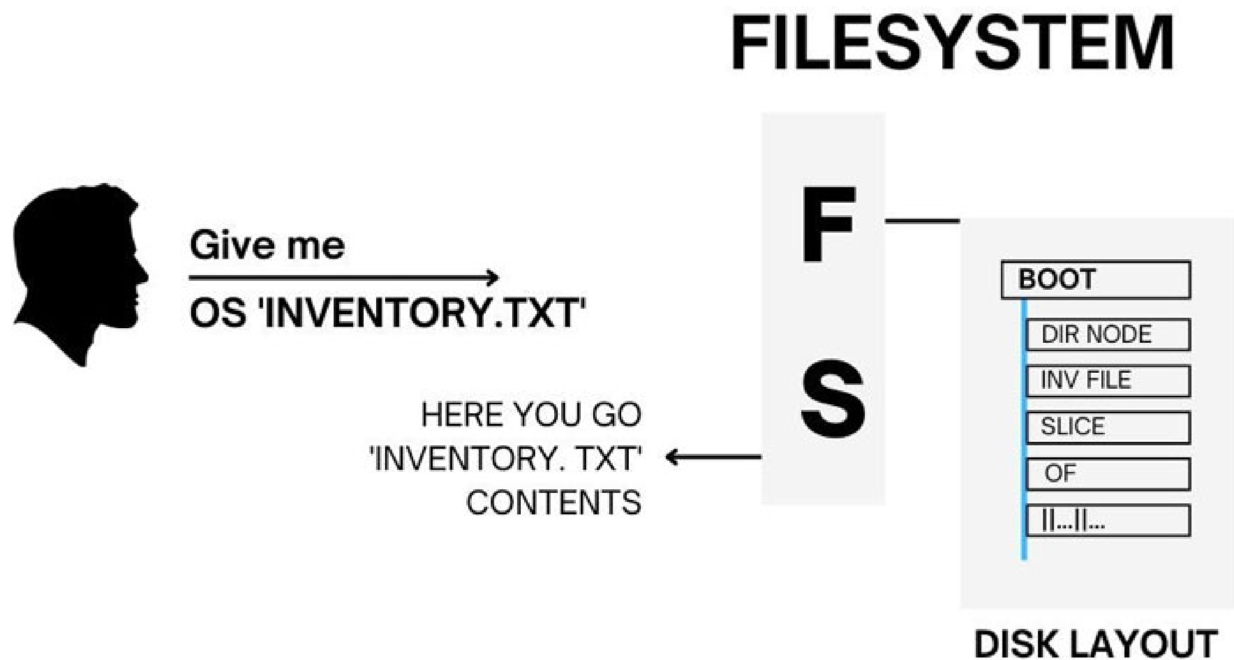


Figure 4.14: *OS <-> Filesystem<->HDD interaction*

OS orchestrates the running of processes that require CPU, memory, and storage to complete the task.

Network

“Network is the computer.”

Profesora is pleased with your toasts and wants to serve them in her popular restaurant. You also realize that while making toast is good, they are piling up with no one to taste its deliciousness.

As Chef Tokyo, you are stuck and need someone to work with Oslo to receive orders from multiple customers and deliver the finished orders to them.

Enter Nagpur:

Figure 4.15: *Nagpur—network guru*

Nagpur is well connected and can snake through any obstacles to deliver the order. He delivers the input orders to Oslo, who then adds the orders to the input queue, and once you are done, Oslo lets Nagpur know the table number to where the order should be delivered.

You feel incredible that many customers are enjoying your toasts, and your French toast with blueberries is a super hit. *Figure 4.16* shows how the network communicates across servers using Network Interface Card (NIC):

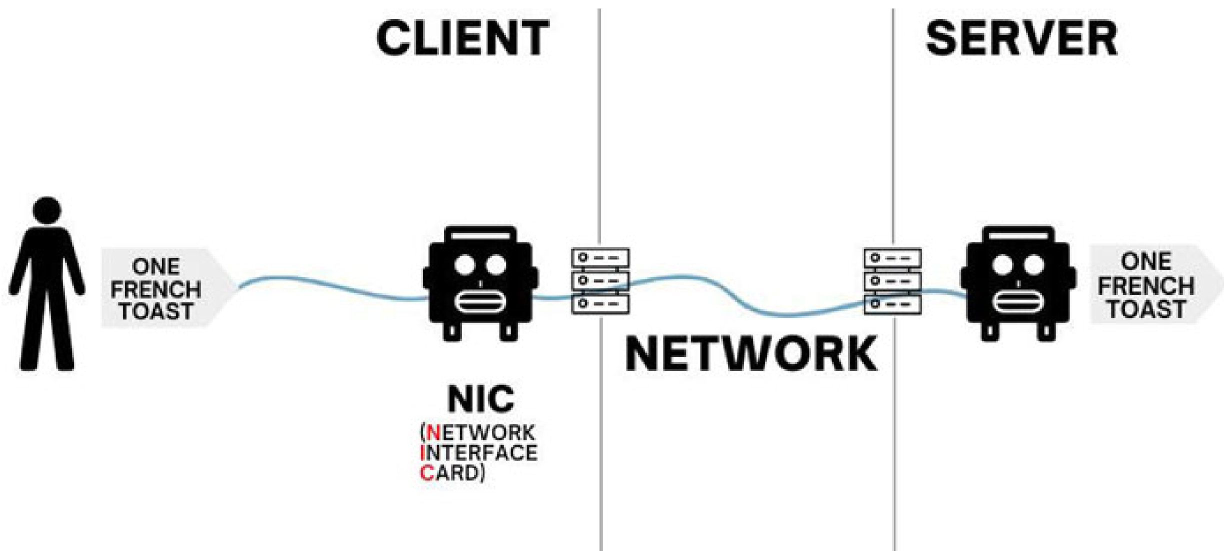


Figure 4.16: *Network communication across servers*

Networking enables computers to process information from other computers. When you click on that link on your phone (or your computer), you are talking via networks to a server (another computer) to deliver the information about the Mars Rover's spectacular landing.

For a process to be more effective, it needs to be part of a network. Physically, networking is implemented using network interface cards. NICs that use radio as a medium of communication is what enable Wi-Fi.

Now, we have a complete picture of what an OS does:

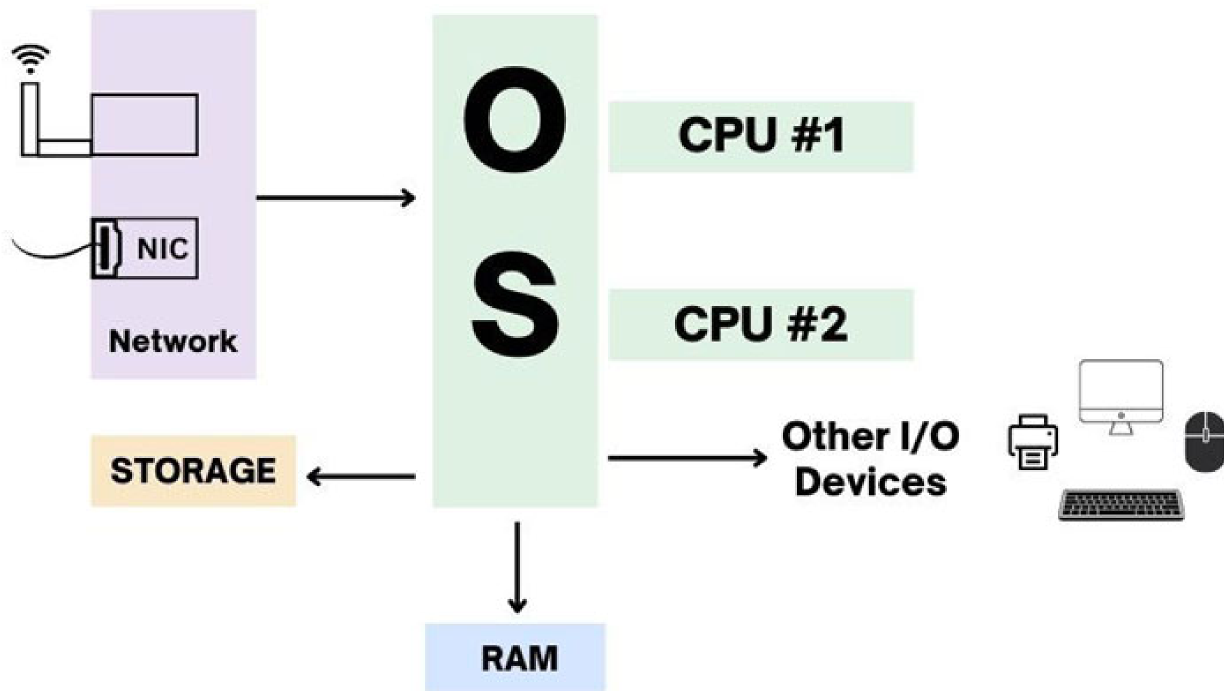


Figure 4.17: *OS: Complete picture*

The operating system manages processes. The CPU power determines the max processing speed. The process has a temporary workspace called RAM to store frequently accessed orders or other data. It opens file handles to read and write files. The process communicates with other servers using NIC, as shown in the following:

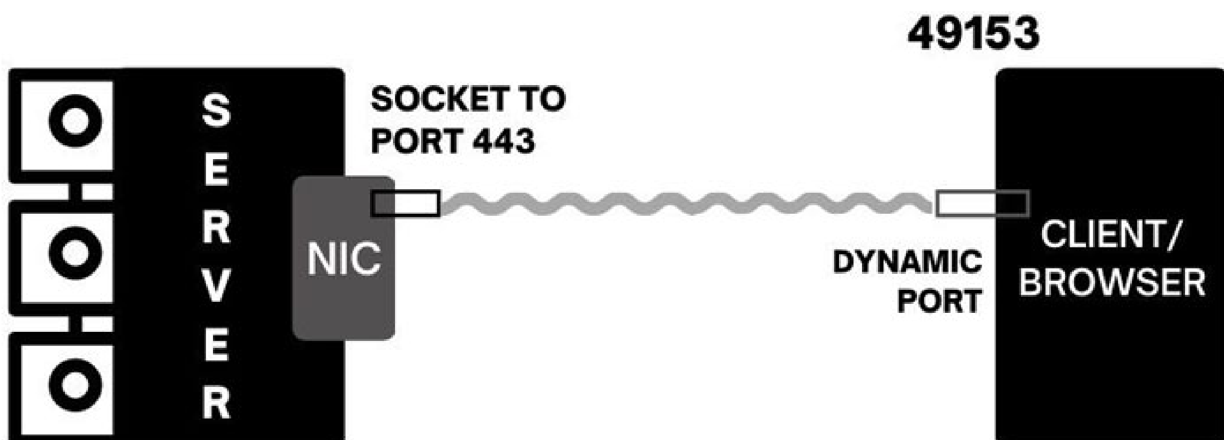


Figure 4.18: *Network ports/sockets communication*

Nowadays, all networking is done by using IP Internet Protocol networks. For communication through IP, the computer needs an address called an IP address. The process can transmit or listen to IP packets on a port. There are some which are famous:

Port number	Protocol
80	HTTP—Hyper Text Transfer Protocol
443	HTTPS—Hyper Text Transfer Protocol Secure
3389	RDP—Remote Desktop Protocol (Used mostly for Windows Remote Access)
22	SSH—Secure Shell (Used for Linux/Unix command line remote access)

Table 4.1: *Well-known ports*

The total number of usable 1–65534 ports in a server is divided into well-known, registered, and dynamic, shown as follows:

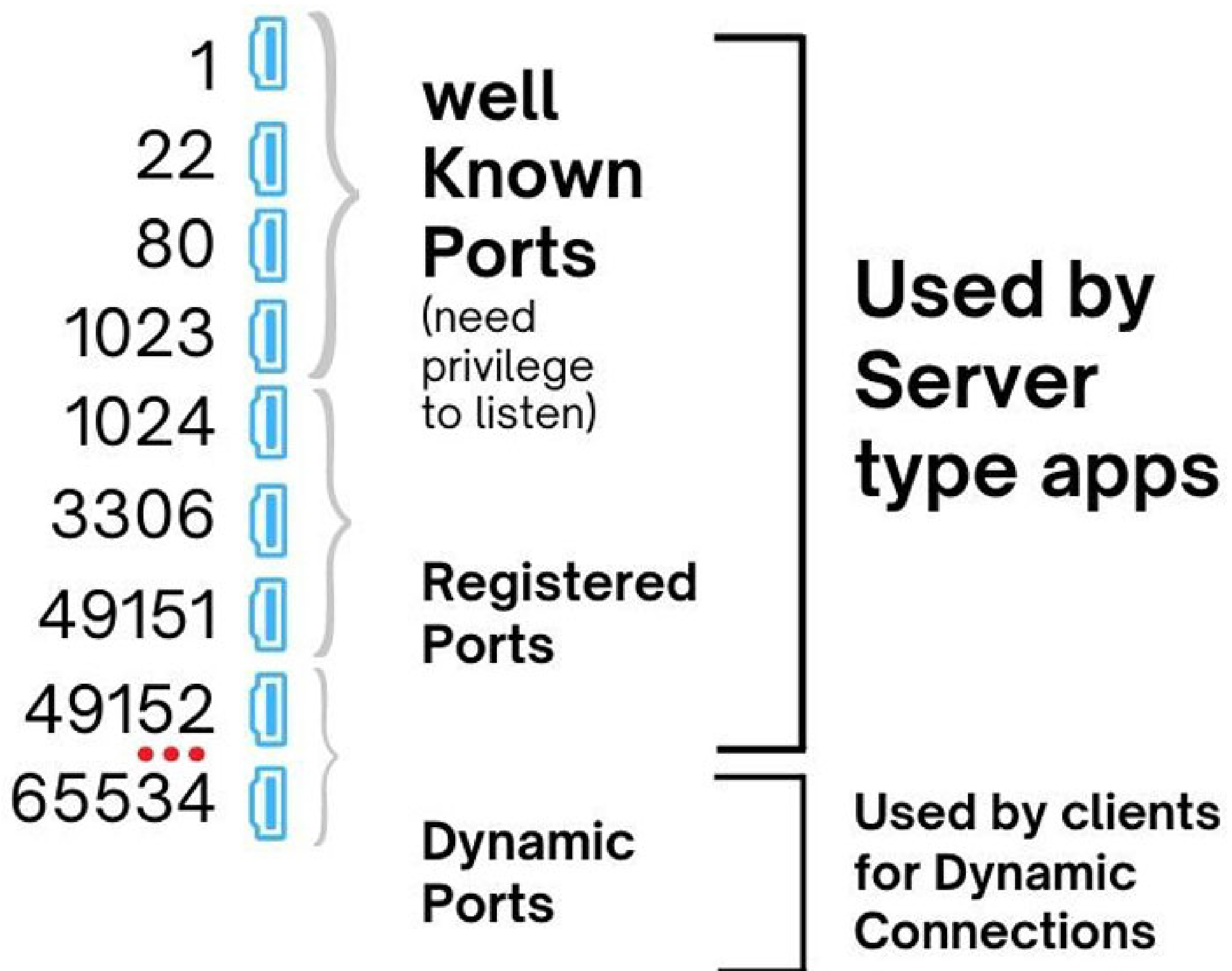


Figure 4.19: *Well-known, registered, and dynamic ports*

For reliable communications, the TCP/IP protocol ensures that IP packets are received in the correct order and are complete. UDP is another protocol used mostly in real-time services such as video and audio streaming, where packet delivery is not guaranteed. It operates in fire-and-forget mode. It keeps firing off without worrying about whether it reaches the target.

Exercise on networking

We are going to spend a little bit more time on networking. The main reason being most of the interview questions are asked in this area, as understanding networking is key for internet companies. They deliver through networks and need people who understand how the network works.

Networking examples: Running your own Web server.

“When I do. I understand”

For Windows:

1. Install the Git terminal from **https://git-scm.com/download/win**. See **https://phoenixnap.com/kb/how-to-install-git-windows** for detailed instructions for installing Git on Windows.
2. Go to **https://python.org/**, and see instructions for installing Python at **https://phoenixnap.com/kb/how-to-install-python-3-windows**.
3. Launch the terminal using Git bash.

   For Mac/Linux, use the regular terminal, and Python should already be installed. You can also see why Linux is the OS that runs the entire Web ecosystem.
4. Run

   ```
   python -m SimpleHTTPServer 8000
   ```

   Or the following if you have python3

   ```
   python3 -m http.server 8000
   ```
5. Go to your browser. Type **http://localhost:8000/**.

   If the error is something like:

   ```
   /usr/bin/python: No module named SimpleHTTPServer
   ```

   Install the HttpServer module using:

   ```
   pip install HttpServer
   ```

   You might have to use pip3 instead of pip. You might also have to install pip3 using the following on an Ubuntu terminal:

   ```
   sudo apt-get update && sudo apt install -y python3-pip
   ```

Even though the page is simple, the mechanism through which it is delivered is the same for every website you visit, independent of how fancy or complicated the site looks.

Let us dig deeper into how it worked. (Very popular interview question, BTW.)

When you type localhost, the browser contacts your computer’s **Domain Name System** (**DNS**) resolver, giving back the IP address `127.0.0.1`. It is a special IP address that is given to the local computer. On any computer, you type `ping localhost`, and you will get `127.0.0.1`. It is the same when you

are typing www.google.com. Your computer tries to see if the local DNS cache has the address for **www.google.com**.

If you see the ping results printed out continuously, like:

```
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.043 ms
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.043 ms
…
```

Exit out of the command by typing *Ctrl* + *C*:

```
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.043 ms
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.043 ms
^C
```

The cache is temporary storage for frequently accessed information that is expensive to obtain. To obtain the IP address of **www.google.com**, your computer must make a DNS request to one of the authoritative servers over the public network each time (if the local cache is not present). This takes too long and overloads the DNS server. Instead, your computer stores the **www.google.com** -> IP address mapping in its local storage (which is called the cache) till the **Time to Live** (**TTL**) specified in the DNS response expires. Then, it makes the request again.

Distributed DNS (DDNS) attacks exploit the above by abusing the DNS request mechanism. They command thousands of machines to repeatedly make the same DNS request on the domain they want to attack so that the DNS servers are overwhelmed.

We will assume the local cache does not have the IP address for **www.google.com**. The computer then asks the DNS server specified in your IP configuration whether it knows what **www.google.com** is. Most likely, they do, so let us again assume that it does not or that the **time to live** (**TTL**) has expired. The DNS server will ask its DNS servers higher up the chain until they finally hit *authoritative* DNS servers. Authoritative DNS servers are the servers that hold the DNS records for just the domains that they serve. For example, GoDaddy and Cloudflare are all authoritative DNS servers for many registered domains.

You can find out who the registrar is by typing the domain name you are interested in: https://whois.com/.

Applicability to kubernetes

Kubernetes orchestrates the deployments of the applications packaged into containers. Your simple Web server application is packaged into a container, and Kubernetes deploys it to serve requests. Inside Kubernetes, it follows the exact mechanisms we went through in this chapter.

Any issue with the application (any application, for that matter) can be troubleshot if the fundamentals explained in this chapter are understood and practiced regularly.

Conclusion

The computer is made of the following four key components:

- Processor (CPU), which does the actual computing.
- Memory (RAM) is used to store temporary data.
- Storage (HDD/SDD) is used for storing data permanently.
- A network (NIC) is used to communicate with other computers.

The operating system helps coordinate the four key components so that computations are done as efficiently as possible. It queues the work for the processor, fetches data from RAM and HDD/SDD, and processes the network packets, among other things.

Understanding how the network works is key to troubleshooting Web applications. Typing **www.google.com** and the display of the page on your browser exercises the entire network stack in your computer. From obtaining the IP address of **www.google.com** via DNS servers to establishing port-to-port connections via NAT in your modem to finally displaying the data in your browser, displaying a Web page on your computer requires an intricate dance between the preceding four components in your computer and hundreds of servers in between.

Running your own Web server is easy and mostly a one-liner that you can simulate most of the preceding interactions on your computer.

In the upcoming chapter, you will learn:

- Introduction to containers/Docker
- Docker ecosystem

- Running your own container
- Building your own container image
- Pushing your custom container to the Docker hub
- Applicability of containers to Kubernetes

Next steps

Operating systems is a hot area in research and in practical day-to-day operations at scale. The operating system issues are notoriously hard to solve as it is very hard to replicate (for some, it is a myth, though). Persons investing time in understanding the fundamentals will reap great rewards when they troubleshoot issues that nobody else in the team can. This leads to bigger responsibilities and, thus, more freedom to work on their areas of interest. To practice more

- Read and buy zines from **https://wizardzines.com/zines/networking/**
- Get Raspberry PI and install your own Linux distribution.
- Experiment
- Play with applications.
- Play by breaking applications intentionally (pulling a network cable while your app is running is always fun).

Points to remember

- The operating system enables the computer to do its tasks as efficiently as possible. It coordinates the activities between CPU/memory/network/storage/peripherals.
- Most modern OSes are multi-tasking (Linux/Windows).
- OS accomplishes its work by launching processes.

Interview questions and answers

1. **My phone has only 16 GB. What am I complaining about?**

   The complaint is about limited storage on the phone. The phone runs out of storage very quickly when taking photos or videos. This amount has nothing to do with the phone's RAM. RAM is also measured in

GB. This can be confusing. An engineer should have a very clear understanding of the components of a computer.

2. **What are the main components of a computer?**

   The main components are the CPU, RAM, Network, and Storage. GPUs are becoming an important component in servers due to the high demand for video processing and machine learning.

3. **Why do we need an operating system?**

   An operating system is needed to provide an abstraction layer to the CPU, RAM, Storage, Network, and other peripherals. A multi-tasking operating system like Linux/Windows ensures optimal resource utilization so that a single application waiting for input does not block other applications from running. Advanced features of an operating system provide better security and isolation between applications such as cgroups and namespaces, which are used for containerization.

4. **Describe what happens when I type http://www.google.com on my browser.**

   Ask the interviewer what depth they want the answer on. Start with the basics:

   - The browser checks the type of URL (HTTPS vs. HTTP vs. file, and so on) and launches a request accordingly. Here is the request for the HTTP protocol. The browser asks the operating system to find the IP address for the domain name **www.google.com**.
   - **www.google.com** is resolved into an IP address using DNS servers. (DNS prefers UDP but will use TCP if required). DNS uses well-known port 53.
   - Your computer establishes a TCP/IP connection between your computer and the IP address returned from the previous step.
   - Using the TCP/IP connection, it sends an HTTP GET request to the IP address of **www.google.com**.
   - The server listening on port `80` returns an HTML page with a search box.
   - The browser renders the HTML page obtained in the previous step.

The details required by the interviewer can be arbitrary and most likely involve the interviewer's area of expertise. It is unlikely but possible that a hardware engineer would like to know whether you know about key bounce handling that needs to occur to make sure that when you press `w` on your keyboard, only one `w` and not `wwwww` shows up on the browser. A network engineer might be interested to know about the NAT protocol being used between your router and Google's servers. A backend server person might be interested to know about geo-caching. The *simple* task of viewing the Google landing page is a result of years of hardware, software, and design improvements. Just remember that one answer will not satisfy all interviewers.

CHAPTER 5
Containers/Docker

Introduction

We are now at the most exciting part of this book in this chapter, namely, containers. Containers form the foundation of Kubernetes, and a thorough understanding of their capabilities and limitations is crucial for having a good experience with Kubernetes. We will learn the similarities between processes and containers and how to build, run, and manage containers.

Structure

In this chapter, we will discuss the following topics:

- Processes and their relevance to containers.
- A brief history of containers.
- Building, running, and managing containers.
- Relevance to Kubernetes.

Objectives

After reading this chapter, you will be able to understand the benefits and limitations of containers. You will be able to build custom container images that package your application, run, and test them. You will be able to manage and debug containers and container images. You will be able to answer basic and some advanced questions on containers/docker and their relevance to Kubernetes.

Processes and their relevance to containers

Processes are the higher level of work managed by the operating system. They require four resources to operate (in general):

- **CPU**: Central Processing Unit and runs the compiled instructions of a program (measured in GHz).
- **Memory**: Place to store interim program results or cache data from network storage.
- **Storage**: Permanent place to store data used by the process to perform its function.
- **Network**: Allows the process to save or receive data to/from other computers that connect to it.

Containers have the exact requirements as a process to run; they need CPU, memory, storage, and network access.

What is the difference between containers and processes, then? To understand the difference, a slight detour into the history of virtualization. As computer servers became more and more powerful, enterprises found that they were running idle most of the time. The underutilization of servers increased the **total cost of ownership** (**TCO**) for running these servers. On top of that, these servers had manual configuration changes and libraries/applications manually installed. This meant that the mission-critical applications running on these servers would run on only those particular servers. The administrators cannot replicate the required configuration in another (even more powerful server) as no one knows what necessary changes need to be made to run the mission-critical applications.

VMWare introduced its virtualization platform that revolutionized how servers were managed. With their virtualization technology, it was easy to divide the CPU, memory, storage, and network into a single server to be used by multiple instances of the operating systems.

Each OS ran independently and had full CPU, memory, storage, and network access. VMWare virtualization allowed multiple instances of OS to run on the same server. Tremendous savings and better management of servers resulted.

The gains were not without costs, especially for developers. It was tough to share working images of the OS as they would be at least multiple GBs. They were not reproducible and required mighty servers to run. They were slow to start and required expensive licensing for advanced (sometimes essential) features.

Docker built upon Linux container technology (**LinuX Container runtime** (**LXC**)) provided an easy-to-use interface for developers. The images were constructed from code, so these systems were reproducible anywhere (this avoids the manual configuration/manual installation of libraries problem).

That concludes our brief interlude into virtualization/container history.

Coming back to docker and processes, what makes containers a fast and lightweight technology is that a container is just another process from an OS perspective. Let me repeat that, from an OS perspective launching a process (which is the primary purpose of an operating system) is not different from launching a Docker container.

Let us see what advantages it provides us:

- **Lightweight virtualization**: Like OS can handle multiple processes simultaneously, and OS has no issues running multiple Docker containers.
- Multiple processes share the server resources very efficiently, and as Docker containers are the same as processes, they also run very efficiently.
- **Efficient resource usage**: VMs require dedicated CPU, memory, and storage. Docker containers, just like processes, share the CPU, memory, and storage.
- Due to efficient sharing, many more docker containers can run on the same server.

What are the disadvantages of docker-based versus VM-based virtualization?

- **Isolation**: Imagine sharing your house with ten families in a shared bathroom and kitchen configuration. You will run into resource contention. This conflict over resources would be the case, especially if you have misbehaving roommates staying in the bathroom for too long. Although there are ways to make containers behave well, using container limits, the OS cannot guarantee complete isolation.
- **Security**: Since the kernel is shared, any bug in the kernel affects all the containers running on the system. In VM-based virtualization, it will affect only the VM running the insecure kernel. A rogue container can take over the entire host if proper security settings are not in place.

You might have heard of crypto mining hacks that take advantage of this.

Security Tip: Allow containers to run only as non-root users.

- **Availability**: If a VM goes down, only the application running on that server becomes unavailable. If a docker host goes down, it takes down all the containers (applications) running on it.
- **Complexity**: Docker hosts can run hundreds of containers on a single host, each with its IP and storage needs. Licensing that depends on the network MAC address may not work. Routing network traffic to hundreds of containers is complex and not as simple as one VM couple of IPs.

VM technology is like townhomes. You and your family get your dedicated place with their own bathroom, kitchen, and TV. You do share a common wall and other resources. Isolation from misbehaving neighbors is way better. You do lose the efficiency of living in a shared house, however. *Figure 5.1* illustrates the isolation present in containers, VMs, and bare metal servers:

Figure 5.1: *Containers versus VMs versus bare metal servers*

To complete the housing analogy, a base-metal server is like living in an independent home. There is no sharing with anybody else, and all the resources in the house are available to you and only to you. You are entirely isolated from the activities of your neighbors.

The single takeaway from this chapter is shown in *Figure 5.2*:

Figure 5.2: *Running containers == running processes*

Cgroups

Cgroups is the short form for Control groups and is a feature of Linux that enables the isolation of resource usage by processes. Containers exploit this feature to limit their access to CPU, memory, network, and storage resources.

Containers combine Cgroups with a related feature of Linux called namespaces to provide isolation between containers. Linux namespaces should not be confused with Kubernetes namespaces, which is a completely different concept. Namespaces in the Linux kernel prevent the containers from seeing each other's resources.

Cgroups *limit* access to resources for containers, whereas namespaces, prevent containers from *seeing* each other. Combined, we get the isolation and resource allocation capabilities required to make containers useful.

Introduction to containers

We have gone through the building blocks that enable containers. To summarize:

- Operating system schedules processes and coordinates the access to system resources (CPU, memory, network, and storage).
- Processes shared the entire system resources before Cgroups and namespaces were introduced into the Linux kernel.
- Cgroups provide a quota mechanism for processes to limit the system resources they use.
- Linux namespaces provide an isolation mechanism for processes so they cannot see other resources.
- Containers leverage Cgroups and Linux namespaces to launch processes, making them lightweight and quick.
- Containers share computer resources; thus, many can be packed into a single server.

Now, we can officially introduce containers.

A container is a set of one or more applications/processes that run isolated from the host system. They are packaged with a root filesystem, dependencies, and binaries to run as an independent unit. They are built using layers of data, and the layers and the container image have a checksum associated with data integrity. The container images can be pushed and retrieved from a central place (called a **registry**) and run on any Linux system*. The container image is used to launch container instances that run the application packaged within the image.

*** - *Any Linux system* statement comes with the generalization that most of the servers are running AMD/Intel's x64 architecture. As we will see in future chapters, containers can run in ARM systems like Raspberry PI. Caveat: Containers built for x64 architecture will not run in ARM-based systems and vice versa.**

Container registry

Deep thoughts: If you have a nice meal in a fancy restaurant but did not share it on social media, have you eaten?

Knowledge is not worth much in life if you cannot share it with others. Docker revolutionized software development with the introduction of DockerHub. Like GitHub, where you can push and pull code, Docker hub, a container registry, allows developers to push and pull containers' images. DockerHub made it super easy to distribute applications.

A container registry like DockerHub is a repository of container images. Like GitHub, where you can push and pull code, container registries allow you to push and pull container images.

DockerHub, when it was introduced, was revolutionary in two ways:

- Developers can share applications written in any language with anyone without requiring them to have the compiler or special software required to build the application. This contrasts with GitHub, where it is complex to build the end application even though the code is available.
- With just docker installed, you can run the application just like the developer who built it in the first place (no more "*works for me!*" problem)

Docker registry also introduced the concept of layers. Layers allow us to download only what is not present on the server already, like getting only the newer commits once you have downloaded a git repository.

Container networking

On a server, the number of ports is $2^{16} = 65{,}536$. As with any computer address scheme, the count starts with zero. The computer can use any port between 0 and 65,535. Of these ports, 0–1023 (<1024) are privileged and normal users cannot run servers on them.

Some of the ports one is expected to know in an interview are as follows:

Port	Application
`20,21`	FTP
`22`	SSH (Secure Shell)
`23`	Telnet

25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name System (DNS)
80	HTTP (Hyper Text Transfer Protocol)
443	HTTPS (HTTP – Secure)

Table 5.1: *Ports*

Beyond the 0–1023 ports are 1024–49,151 (<49,152), the registered ports available for running servers. Some well-known ports in this range are as follows:

Port	Application
3000	Ruby on Rails Dev Port
3306	MySQL
3389	RDP (Windows Remote Desktop)
5000	Python Flask Dev Port

Table 5.2: *Some well-known ports*

The following diagram illustrates the well-known, registered, and dynamic ports (again from *Chapter 4: Operating System Fundamentals*):

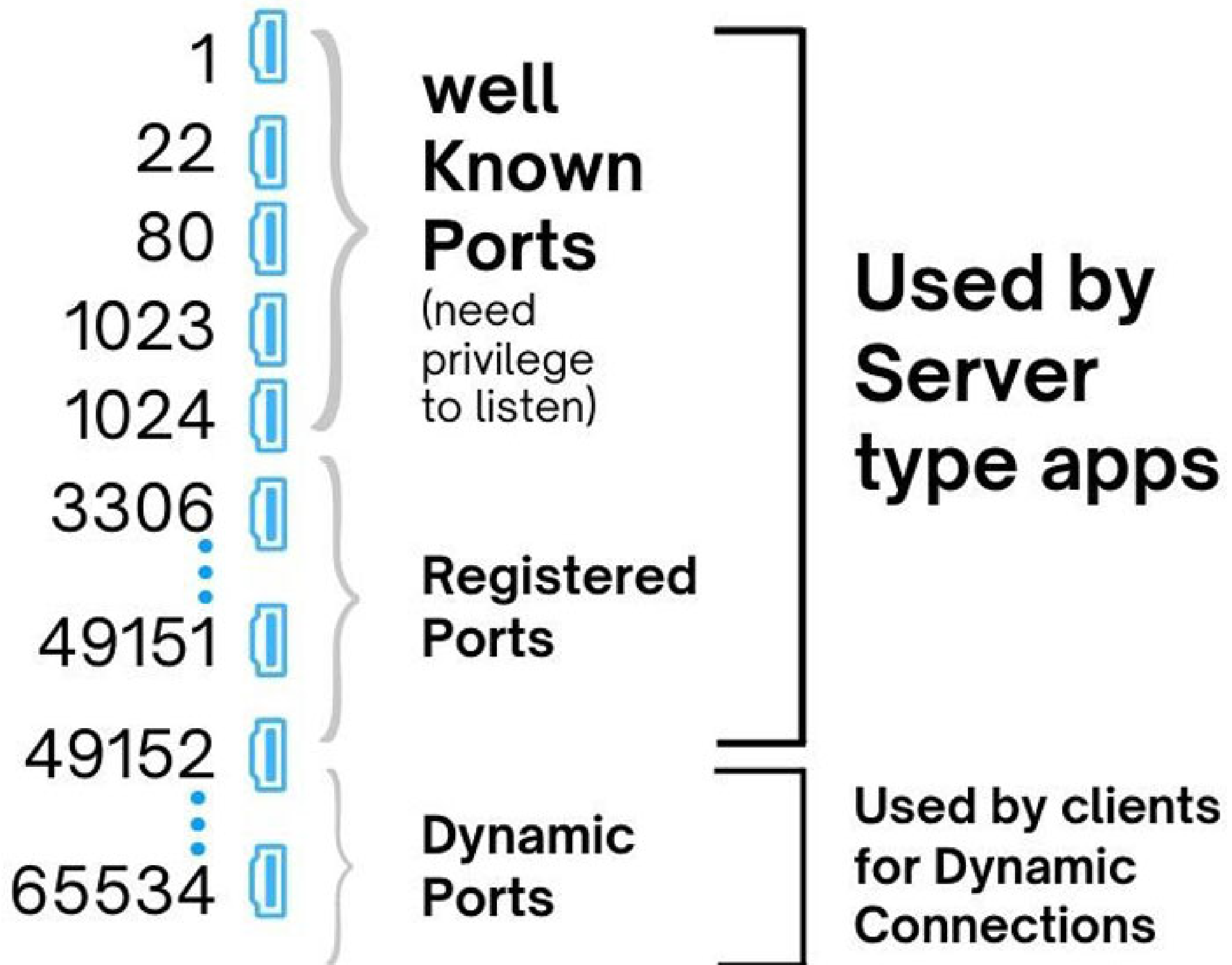


Figure 5.3: *Well-known, registered, and dynamic ports*

When a server has a service listening on a port, no other service can listen on the same port. The mechanism of assigning a port to a service is called `binding`.

When a service starts up, it asks OS that it wants to listen on, say port `80`; the OS then binds it to port `80`. When another service makes the same request, the OS will refuse the request with an error message like:

```
Failed to bind to port 80, address already in use.
```

So, only one service can listen on a port in a server.

One port per application creates a problem packing as many docker instances as possible on a server. **Team A** has a Web service, and **Team B** has a Web service. Both listen to port `80`. Deploying on the same server will let only the first service that binds to Port `80` come up. The problem with two servers listening on the same port is shown in *Figure 5.4*:

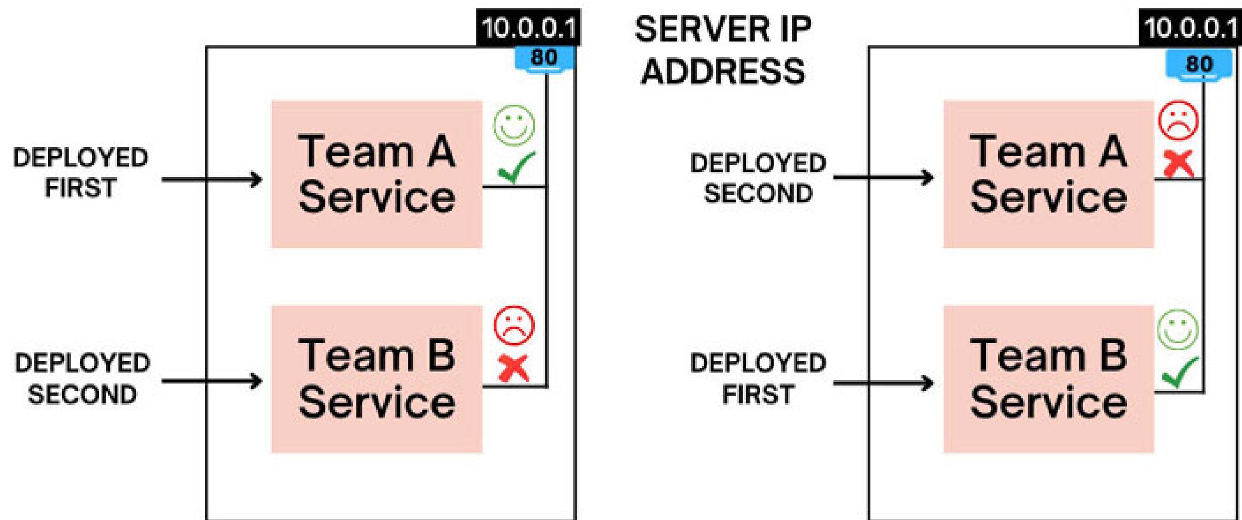


Figure 5.4: *First in, first out*

Docker introduces a virtual networking layer between the real server ports and the docker container ports to overcome this problem.

Each container thinks it is a server on its own and has all the `65536` ports available. Heck, it is even allocated its own IP by the docker virtual networking layer (something like `172.10.121.1`). From a networking perspective, docker containers deployed with docker virtual networking look like the following:

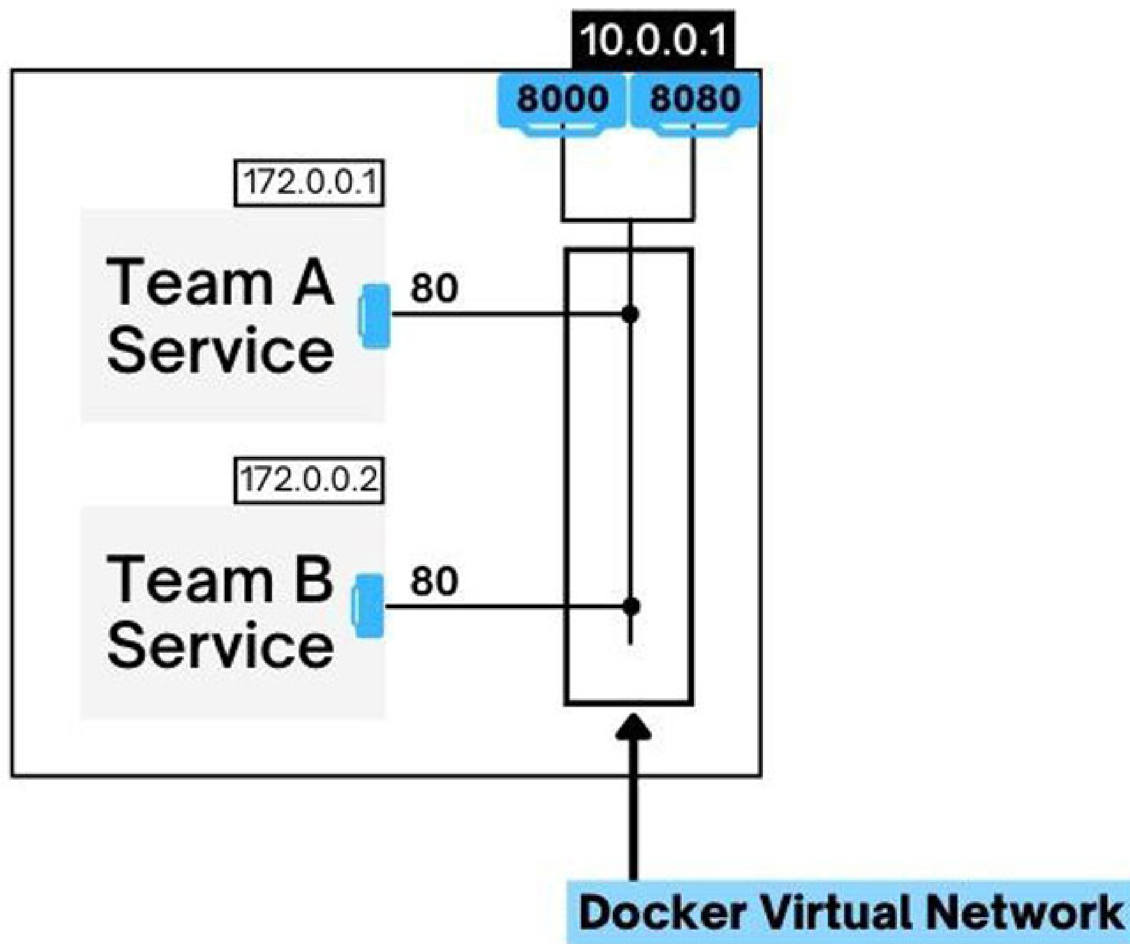


Figure 5.5: *Docker virtual networking layer*

Container storage

Did you know you can run Fedora on an Ubuntu server using docker containers?

It seems like magic that you can run multiple Linux operating system distributions on a server that is running a completely different operating

system. How is that even possible? The *magic* lies in how storage is mounted in the container.

Back to "how it works on a regular server":

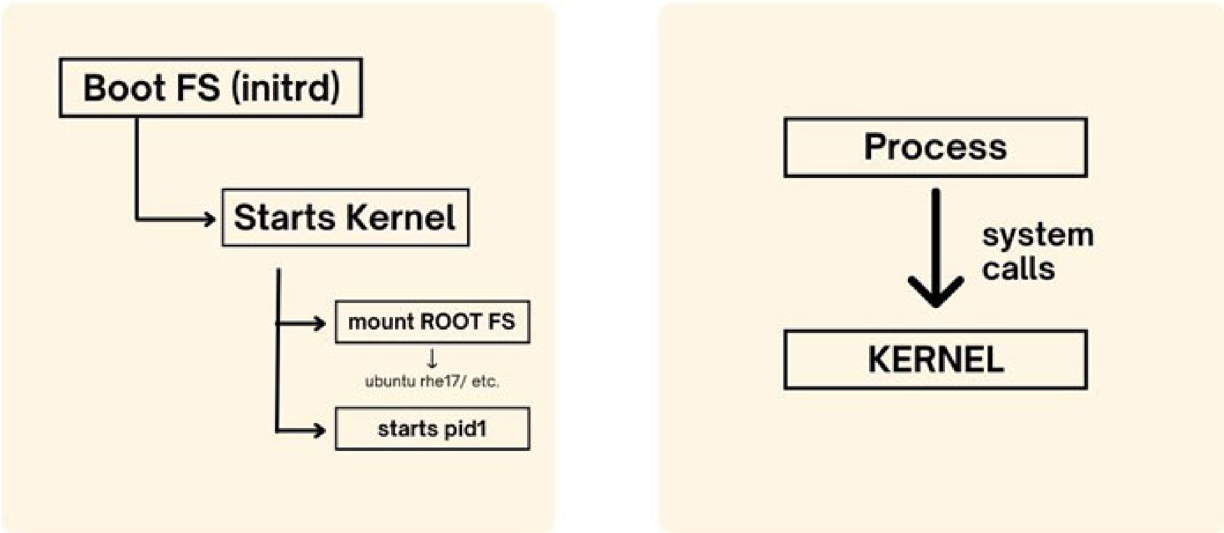


Figure 5.6: *Boot process and process accessing kernel on a regular server*

Remember running a container is still a process as far as the kernel is concerned.

What container runtime does is mount the rootfs from the provided image as root. The applications run as usual, and when system calls are made to the kernel by the application, they are passed on to the server host kernel via Cgroups and namespaces. The mount process for containers is shown in *Figure 5.7*:

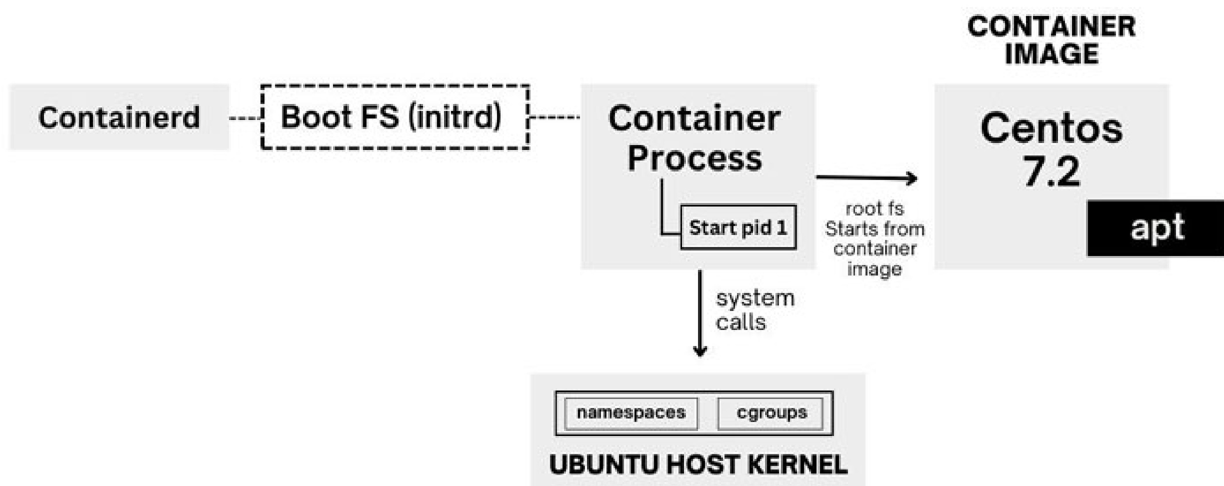


Figure 5.7: Container process mounting its rootfs and accessing the kernel

The mounting of the different rootfs makes it possible to package all dependencies for an application in the image, including any distribution-specific commands, say yum (available only in Redhat distribution), and make it run on a Debian-based system like Ubuntu (which does not have yum package manager installed).

A container's file system is separated from the host's file system by using a technology called a union file system. A union file system allows multiple file systems to be "stacked" on top of each other, with the top-most file system taking precedence. In the case of a container, the container's file system is "stacked" on top of the host's file system.

During runtime, when the container starts, the container engine (such as Docker) takes the container image and creates a new file system for the container by "mounting" the image's file system onto a new location in the host's file system. This new location is called a "container root filesystem" and it's where the container operates. The container engine also sets up a virtual network interface and other resources that the container needs to run.

When the container accesses files or directories, it's actually accessing the files in the container's root filesystem and not the host's filesystem. The container engine routes these access requests to the appropriate file system and returns the results to the container.

In summary, a container's file system is separated from the host's file system by using a union file system. During runtime, the container engine takes the container image, creates a new file system for the container and "mounts" the image's file system onto a new location in the host's file system which is called container root filesystem and it's where the container operates.

This section went into the deep end to explain the *magic* of running Ubuntu on RedHat Enterprise Linux 7 (RHEL 7 host). How can we answer this if this question gets asked in an interview?

Can you run Ubuntu applications, like apt and so on*, on an RHEL 7 Linux host?*

Yes, you can by using containers. Containers mount the Ubuntu binaries and use the shared kernel whenever the binary requires to make a system call.

If you do not fully understand this, it is ok; come back to this section after running some examples.

Running a container

Let us get our hands dirty. Ideally, you will have a docker installed on your computer.

Docker can be installed on:

- Linux
- Windows
- Mac

If you do not, do not worry. Thanks to docker, you can try all the examples using just your browser. Go to https://abs.play-with-docker.com, obtain a user account and click on **`Add instance`**. You should see a shell prompt. Notice that the underlying host OS is an *alpine* distribution.

Open a terminal (Power shell or command prompt in Windows) for people who have installed docker.

Run the following command:

```
docker run --name nginx-web -d -p 8080:80 nginx
```

You can now go to **http://localhost:8080/**, and you should see the Nginx welcome page on your browser.

Let us review the accomplishments you have done by running a single command. You are running the latest version of the Nginx Web server without:

- Searching on the internet for where to download the latest version of Nginx.
- Figuring out whether the newest version is available in the software repository for your particular distribution.
- Reading articles on how to set the basic configuration of nginx to get it running.
- Poring over articles on how to run it as a background service.

The need to compile from source using a complex set of tools and to ensure an even more complicated set of required dependencies exists can be very frustrating even to an experienced engineer. Even if you get all those correct, you will run into the correct binary/library for your particular

distribution/version of Linux not being present; hence, your application will not run. (Trust me on this, it is excruciating.).

The reason docker got popular was precisely due to the preceding reasons. As a software producer, you no longer need to figure out how to generate different distributions or worry about the special compiler tools and settings required to create the binary.

You know your consumer has an image that will run on their system the first time they try it. As a consumer, you download the image and run it. If the Docker file generated from the image is available, you can also build the same image or modify it according to your needs. Docker solves the perennial problem between developers and consumers: the *works for me* problem.

Runtime configuration of a running container

Now that we have solved the software binary generation and distribution problem, we run smack into the still unsolved (at scale) runtime configuration problem.

After billions of dollars in cost and effort using multiple frameworks, the runtime configuration of services is a complex problem. You know that when you hear about the entire internet has gone down because someone set the wrong configuration value in one of the routers or DNS.

This section will not deal with the more significant software configuration problem at scale. We will focus on configuring a single service running on a single container.

ENV variables

Software like Nginx, developed over many years, has multiple settings that can be configured. For example, you might want to set the hostname for the container instead of the random set of characters that docker sets it to.

To set it to run, use the following command:

```
docker run -e HOSTNAME=my-cool-server -d -p 8080:80
```

We can pass multiple values like `HOSTNAME` to arguments that the software accepts to change the runtime behavior of the software.

Here is an example of setting one more value to a server that connects to a Postgres Database server:

```
docker run -e POSTGRES_USER='postgres' -e
POSTGRES_PASSWORD='password' myappthatusespostgres
```

`myappthatusespostgres` is a fake container image that does not exist. It is listed here to illustrate the concept.

To pass the sensitive variables in a slightly more secure fashion, you can pass them via a file using the **`--env-file`** parameter.

Building your own container image

It is cool to run any of the thousands of images available on **https://dockerhub.com/**, like MySQL, Python, PostgreSQL, etc.

It is cooler to build your own. As with any starting exercise on computers, we will build a container that prints `hello world`. Here is the Docker file, which has the build instructions for building such an image:

```
FROM alpine
CMD echo "hello world."
```

Save the preceding contents to a file named Dockerfile.

Run the following command:

```
docker build . --tag my-hello-world:v0.0
```

You should have an output something like as follows:

```
=> [internal] load build definition from Dockerfile        0.2s
=> => transferring dockerfile:
80B                          0.0s
=> [internal] load
.dockerignore                              0.2s
=> => transferring context: 2B
0.0s
=> [internal] load metadata for docker.io/library/alpine:latest
0.0s
=> [1/1] FROM
docker.io/library/alpine                              0.3s
```

```
=> exporting to
image                                                      0.0s
=> => exporting
layers                                                     0.0s
=> => writing image sha256:<xyz>
0.0s
=> => naming to docker.io/library/my-hello-world:v0.0     0.0s
```

Here, image `<xyz>` is the image id docker created when building the image. The last output line shows that the image `<xyz>` is named `my-hello-world:v0.0`. You can also read it as it is *tagged* with the name `my-hello-world:v0.0`.

Congratulations, you are a proud parent of a container image. You could run the image `<xyz>` directly, but it is unwieldy. As we tagged it with a human-friendly name, you can run the image using a human-friendly name, `my-hello-world:v0.0`.

Run your container the same way as any other container.

```
docker run my-hello-world:v0.0
```

You should see the output.

```
"hello world."
```

When building images locally, it gets stored in a local docker registry. Normally, we build the images to be pushed to a registry like DockerHub. If no prefix is given, then the DockerHub registry is assumed. The repository path is assumed to exist in DockerHub. Later, when we see how we push your image to DockerHub, the path becomes important. It will be in the form of `<myaccountname>/<reponame>`. For example, mine is `gshiva`. You can see the `my-hello-world` repository with the `v0.0` tag there.

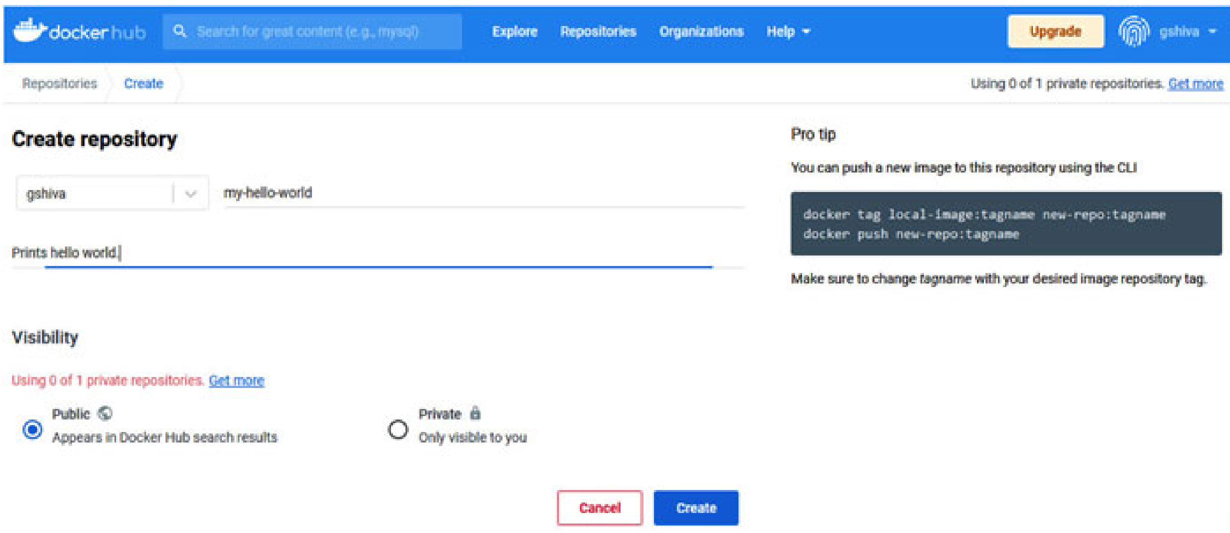


Figure 5.8: *Create a repository in DockerHub*

Along with tags, labels can be attached to a docker image. Labels are key-value pairs used to attach metadata to container images. These are very useful for CI/CD, where decisions can be made using the data in the labels. Because the labels can be specified during build time, they cannot be changed after the image is built. This makes it more reliable than tags which can be changed at any time.

```
# Add this to the Dockerfile
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

Let us review the Dockerfile to understand key concepts of Docker Images, namely, layers. Layers allow docker images to be very efficient in downloading minimal information when building/downloading container images.

```
FROM alpine
```

The first line tells the docker the build command to choose the **`alpine`** image as its starting point. The **`build`** command does not download the **`alpine`** image every time. It checks its local cache for the image with the same SHA sum as the **`alpine`** image. The flow is listed as follows:

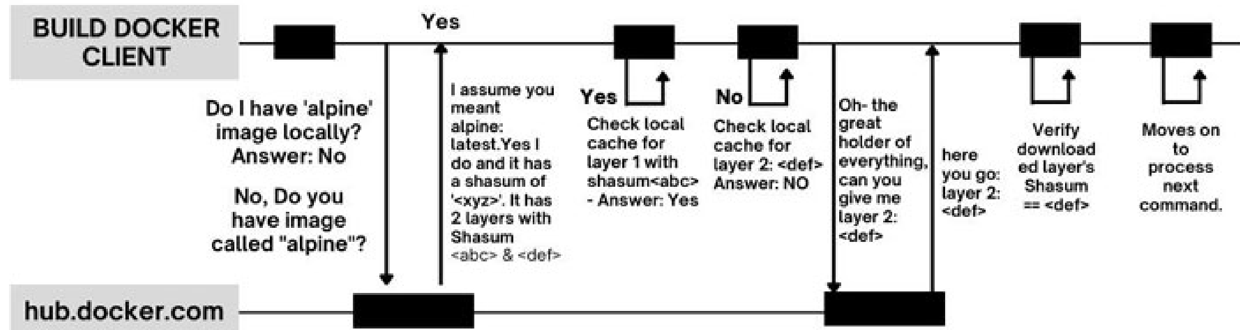
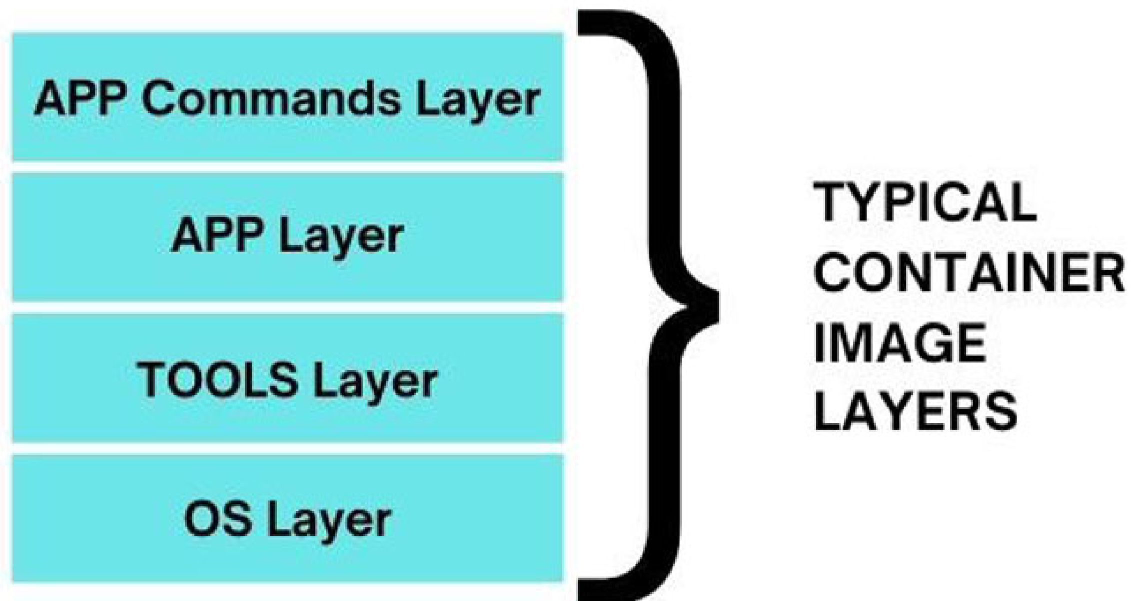


Figure 5.9: *Sequence/flow diagram for the docker build command*

Why do we need layers?

You can think of layers like git commits. An image is built layer upon layer, and each command is in a docker file that acts as a separate layer:



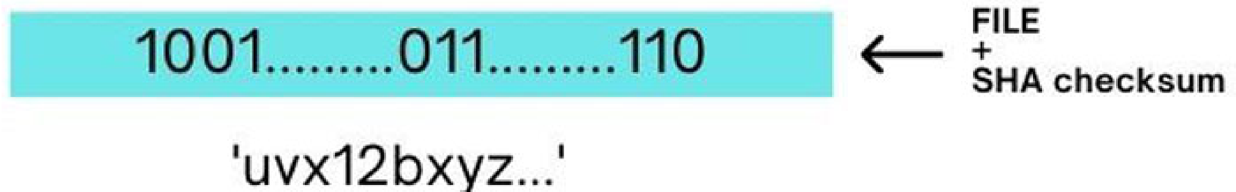


Figure 5.10: *Docker layers and layer information*

1. **You can calculate the SHA sum on any file.**

```
echo "abc" >SHA-test.file
sha256sum SHA-test.file ---> abcd
```

   It will be the same for the same file contents.

   SHA sum is a fixed length of 65 characters independent of the file size.

2. **This immutable property is used to verify the integrity of any file, including docker layers.**

You can see the SHA sums commonly listed along with downloadable packages as a preventive measure against a man-in-the-middle attack, where a malicious actor sends you a malicious file with changed contents.

SHA sum is also used to detect corrupted downloads as a checksum. You can inspect the layers created for your image by running:

```
docker image history <image>
```

Corollary

As each line in Dockerfile creates a separate layer, people try to minimize the number of commands to the bare minimum.

The docker build command tries to see if each command will change the layer contents. If it does not, it does not run that command again while building and skips to the following line.

This considerably improves the build speed, provided the commands have been constructed, considering this knowledge.

Why is this information important for Kubernetes?

Kubernetes uses layers of data to download the minimum data for launching a container.

The less data it needs to download, the faster the image will come up. For multiple reasons, keeping the image sizes used in production as small as possible is critical.

Kubernetes launches containers, tons of them. They all use the same mechanisms as your local Docker install. Understanding how to build, run, and debug containers is essential to becoming an effective Kubernetes engineer.

Multi-stage builds

Dockerfile build commands are limited but very powerful. This chapter touches on the bare minimum to get you through an interview. You are strongly encouraged to spend time trying out various commands by building your own images.

In one of the corollaries, we saw the need to keep the production image size as small as possible. It is done using smaller base images and minimizing the number of layers. At the same time, we want the developer to have complete freedom in using their favorite tools for building their application. We have Dev versus Ops problem. If you want to make Devs happy, you have images that run into GBs that cause operational and security issues. If you make Ops happy with small-size images, it becomes extremely difficult for Devs to build their applications.

Docker came up with an innovative solution for competing concerns called multi-stage builds. It separates the build process into multiple stages. The most used pattern is one stage for building and using the output in the second stage (there are no limits on the number of stages you can have).

It is important to note that the base image for different stages need not be identical. So, we get to eat our cake also. The developer can use an image with all the compilers' developer tools installed for building their application. The tools can be as heavy as they want (for example, JDK, GCC, go compiler, and so on). Even though not recommended, they can have as many dockers build commands as they want.

The production/security team can supply minimal distribution on the base image. The production team will be happy if the binary/output can run on the smaller base image.

The process is illustrated as follows:

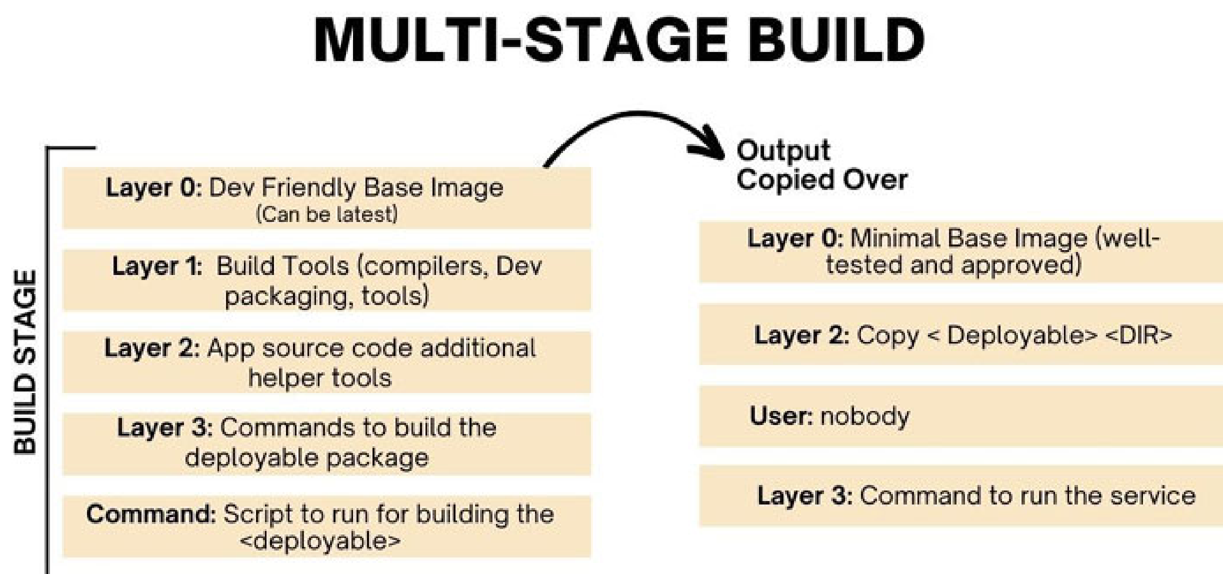


Figure 5.11: *Multi-stage build process*

Using multi-stage builds; we have reduced an image that could be in GBs to maybe a few MBs. There is also less attack-surface; by restricting the production image, we have improved security.

Note the **USER** command, which sets the user id that will be set for the running application. Using the default root user is a serious security violation (and mostly an unnecessary one.)

During an interview, you should insist that all containers in production should be run as a non-root user (whether it is practical in an organization is something you must figure out—once you have got the job). It is a warning if the developers insist that running as a root is good. Run, do not walk away from such organizations.

We will use a contrived example to try out the multi-stage builds.

In our case, our "**dev tool**" is curl. Our deployable aims to print popular baby names in the USA:

```
# Layer 0: Dev Base Image: Ubuntu image size is around 200 MB
FROM ubuntu
# Layer 1: Dev Tools Install: The following line will install
all the compilers and tools. This can be in GBs
RUN apt update && apt install -y curl
# Layer 2: Build the "application"
# The "application" downloads the HTML page containing
# the top 10 baby names from
# US Social Security Administration and outputs them to a
# more machine-friendly format.
RUN curl https://www.ssa.gov/oact/babynames/index.html | grep -
E '[[:alpha:]].*</td>' | sed 's/^[ \t]*//;s/[
\t]*$;//;s/<td>//;s/<\/td>//' > /deployable.txt
# Build the release image
# alpine image is around 5MB (40 times less than regular Ubuntu
image)
FROM alpine:latest
# copy the deployable from the previous build (Build #0) to the
release image
COPY --from=0 /deployable.txt ./
# run the command
```

```
CMD ["cat", "./deployable.txt"]
```

Run the docker build command.

```
docker build . -t last-year-top-10-baby-names:v2022
```

Run the built image.

```
docker run last-year-top-10-baby-names:v2022
```

You should see a list of names. Note that your built image does not need any binaries like **curl** installed. It also does not reach out to the internet for the names increasing the security of the production image.

Container images versus running container instances

Docker runtime uses shasums to identify both container images and running instances uniquely. This can cause confusion.

To list the images in your system, run the following:

```
docker images
```

That shows the list of images in your system.

```
last-year-top-10-baby-names v2022 0c26ddb855b5 About an hour
ago  5.58MB
gshiva/my-hello-world    v0.0 3b53f59de44c  6 months
ago    5.57MB
my-hello-world       v0.0 3b53f59de44c  6 months ago    5.57MB
```

Similarly

```
docker ps
```

The preceding command shows the list of running containers on your system:

```
CONTAINER ID  IMAGE   COMMAND          CREATED     STATUS
PORTS   NAMES
8725eb3c4d9a  nginx   "/docker-entrypoint.…"  3 seconds ago  Up
2 seconds  80/tcp  quizzical_gates
```

One way to think of it is that the images are equivalent to classes in object-oriented programming, and running containers are class instances:

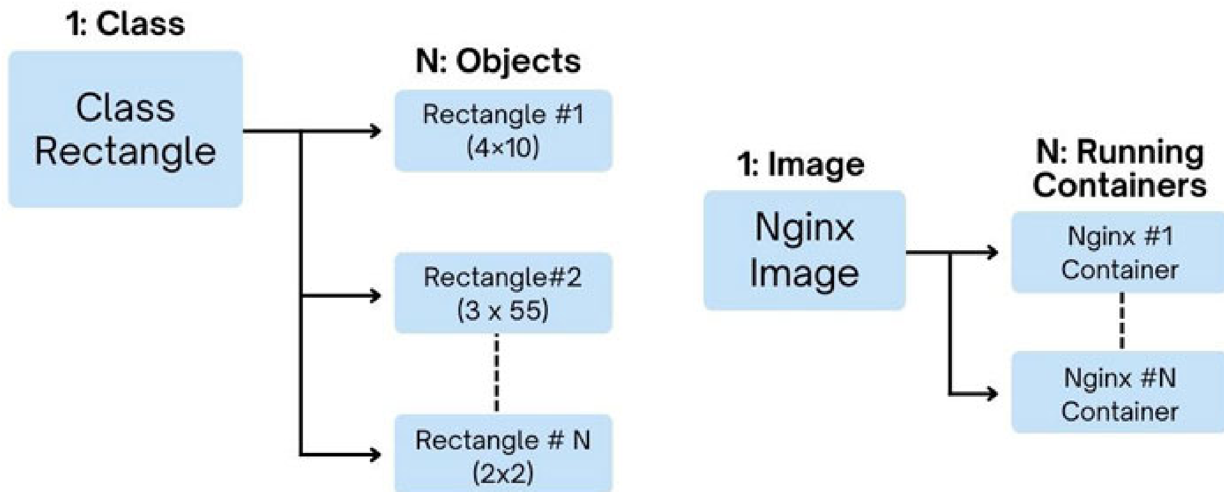


Figure 5.12: *Class: objects are analogous to image: running containers*

Stopping and removing running containers

To stop a running container, run the following command:

```
docker stop <container-instance-id>
```

Or

```
docker stop <container-instance-name>
```

And to remove a running container, run the following command:

```
docker rm <container-instances-id>
```

Or

```
docker rm <container-instance-names>
```

Docker uses the **copy-on-write** (**COW**) file system to store the changes (only) from the base image when running a container. Thus, you can have a thousand instances of an image; the actual disk usage is a fraction of the calculated *N* containers *x* image size. The exercise to try this out using du is left to the reader.

The state information for each container instance is stored in a separate location, usually under (`/var/lib/docker`). Stopping a running container is the same as stopping a running process (*running containers = processes*).

Typing the following command resumes the process. The process continues mainly as if nothing happened.

```
docker start <container-instance-id>
```

Docker removes all the state information associated with the container when you run **`docker rm <container-instances-id>`** (it can be run only on stopped containers).

If you want to store the current state permanently (note that this is not repeatable in CI/CD pipelines), you need to run the following:

```
docker commit <instance-id>
```

Docker generates a new image from the information stored in the running instance. This could be useful for debugging where the issue is seen on a production container and you do not want to change anything in the running instance.

To create an image, the running instance need not be stopped. However, any information in memory or file system caches will not be stored.

Container image management

When you run the following command, the latest tag is added automatically by docker:

```
docker run … nginx
```

As in any software application with multiple versions available, you would like to fix the version you run in production to minimize the unknowns. Docker allows this by setting tags. Docker also enables you to attach multiple labels to an image (a tag can be only one per image).

You can see the versions available generally on **hub.docker.com**.

When you build images, you can also tag them.

```
docker build . tag=mywonderful-app:v1
```

Insist on running images with specified tags in production. Using the *latest* tag can lead to unpredictable environments, especially under Kubernetes, where containers are run on multiple nodes.

Push container

You would like to share your image with the entire world. To do that, you need an account at hub.docker.com. Choosing a free option should be good enough for us. After registering, run **`docker login`** and enter your username and password.

Push your container using the following command:

```
docker push username/mywonderful-app:v1
```

Then verify that you have pushed it; visit your page at https://hub.docker.com/.

Congratulations, you have taken the first steps in mastering containers.

Conclusion

We have covered much ground in this chapter and set the stage for us to be successful Kubernetes engineers. We have seen how running container instances are normal processes that are restricted and isolated using Cgroups and Linux namespaces. You learned how container images are built, how to keep their sizes small, and tips on reducing their size using multi-stage builds to provide hints to developers on how to keep their production images small. How to manage container images and push them to the docker registry was illustrated.

In the upcoming chapter, we will learn about the basics of Kubernetes, the need, and how to run an application in a cluster. We will learn about the most used primitives in Kubernetes, such as pods, deployments, and ingress.

Points to remember

- Running containers is just a process.
- Processes require server resources such as CPU/RAM/storage and network.
- Containers require the same resources as processes such as CPU/RAM/storage and network.
- Containers start with the same speed as a process, with very little overhead.
- Containers are the latest iteration of the virtualization of servers.
- Containers can be seen as a lightweight virtualization mechanism.
- Containers share the server's resources as opposed to a traditional VM's dedicated RAM/CPU/storage.
- Container registries like Docker Hub make sharing images super easy.
- Dockerfile enables the rebuilding of containers anywhere.

- Container networking enables multiple containers to listen on a standard port and routes traffic to the container by exposing host network ports.
- Docker image has a root file system mounted at runtime to the container. This enables the Centos container to run on an Ubuntu host.
- Build a container using instructions specified in Dockerfile.
- Using multi-stage builds, you can keep the container image as small as possible.
- You can run, stop, and delete containers.
- You can push multiple versions of your container to DockerHub for sharing.

Interview questions and answers

1. **What are the differences between containers and virtual machines?**

   The underlying technology between virtual machines and containers is completely different from each other. Virtual machines provide complete isolation of resources such as CPU, RAM, and storage. Due to such isolation, it is possible to run Windows, Solaris, and Linux VMs. Because they are images of an entire server, they are large, making it very difficult to share them with others. Also, it is impossible to share resources across VMs running on the same server due to isolation.

   Containers, on the other hand, are isolated processes launched by the operating system. They share the kernel, CPU, and RAM. Although different distributions of Linux OS can be launched on the same server. It is impossible to launch a Windows container on a Linux Host due to the sharing of the kernel. Containers can be shared easily due to their relatively small size. The code to build a container is even smaller and thus can be shared using code repositories such as GitHub.

2. **How come containers are so fast?**

   Containers are regular processes in an operating system and thus have very little overhead to launch them.

3. **What can slow down the launching of containers?**

Launching of containers can be slow due to startup code that needs to run within the container and if it requires information from external systems to start.

4. **How do you share a container image?**

   Container images are shared by pushing them to a registry.

5. **What makes the building of containers reproducible?**

   Containers are built using Dockerfile specifications. This makes them reproducible.

6. **How do you run a container and expose a port?**

   You use `docker run -d -p 8080:80/tcp nginx`. This runs a Nginx container that listens on port `80` and is exposed via host network interfaces at port `8080`.

7. **Can you launch multiple containers listening to port 80 on the same host? How and when does it work?**

   Yes. Docker virtual network layer allocates unique IP to each container. From a container perspective, it is a complete server that does not share any resource with any other container. Multiple containers can listen to the same internal port without any conflicts. They cannot be exposed using the same host port.

8. **How are labels used in docker images? How are they different from tags?**

   Labels are key-value pairs used to attach metadata to container images. These are very useful for CI/CD, where decisions can be made using the data in the labels. Since labels can be specified during build time, they cannot be changed after the image is built. This makes it more reliable than tags that can be changed anytime.

CHAPTER 6
Kubernetes Basics

Introduction

Kubernetes can get complicated quickly once you add all the technological alphabet soup required for a production-grade setup. Although an individual technology (certificate management, configuration management, security, compliance, networking, service mesh) deserves multiple books on its own, no one ever expects you to have deep knowledge of all of them. In an interview, they want to make sure of the following two things:

1. You have a solid grasp of fundamentals.
2. You have the passion and potential to understand their complex environment quickly.

The good news is that for basic Kubernetes knowledge if you have understood the previous chapters and practiced well, you are set for the rest of the book! All you need to remember, in addition, is the commands through which Kubernetes does container orchestration.

All Kubernetes does on the nodes is pretty much what you do with docker containers: downloading them, passing configuration params/secrets, and running them. Just like you did manually, it also monitors them.

A solid understanding of CPU, ram, storage, processes, and networking will make the rest of the chapters a breeze.

Structure

In this chapter, we will discuss the following topics:

- Brief history of container orchestration solutions
- Getting started with Kubernetes
- Hands-on Kubernetes primitives:
  - Pods

- Replica sets
- Deployments
- Services
- Ingress controllers
- Persistent volume claims

Objectives

After reading this chapter, you will be able to launch an application using Kubernetes. You will also expose the service over the network, make it available externally, and attach persistent storage to your applications.

Need for container orchestration

Containers in production do not run as a single instance, and almost all run multiple instances on multiple nodes for scalability and reliability. When running Docker on your machine, there was no choice but to run the applications on your machine only with no redundancy.

For simplicity, let us take a prevalent scenario and assume that all the instances are stateless:

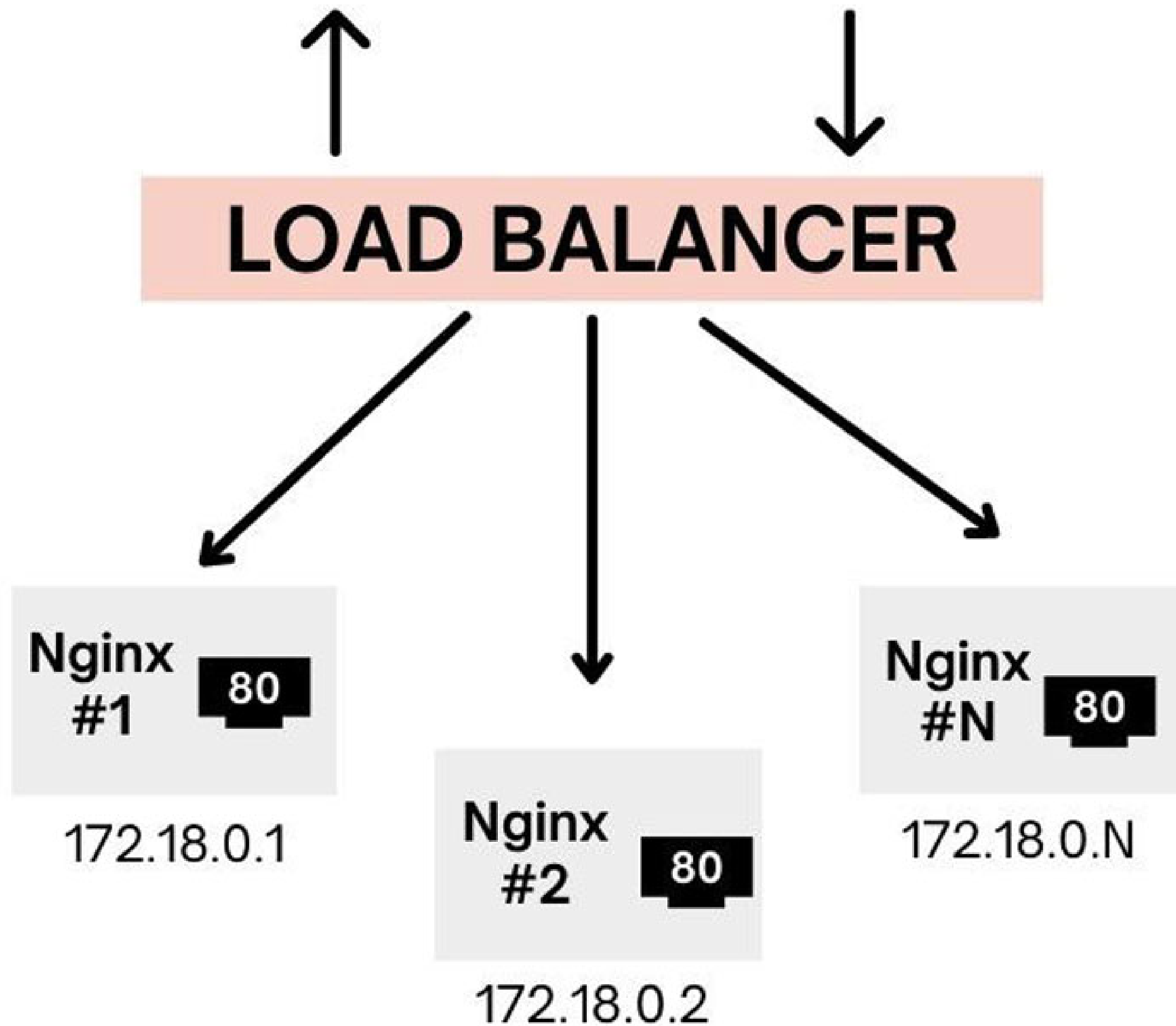


Figure 6.1: *An app with the load balancer*

We will see how you will do this on your machine *the hard way*:

1. Start the Nginx container and expose its Port `80`.
2. Inspect the container and note down its IP address.
3. Start another container, expose its Port `80`, and identify its IP address.
4. Repeat Step 3 till you think you have enough Nginx instances.
5. Start another Nginx that is configured as a reverse proxy. In its configuration, add the IP addresses of containers you have launched and ports as the backend.
6. Expose the load balancer Nginx container's `80` as the hosts' Port `8080`.

7. When you type `http://localhost:8080/`, it first hits the load balancer Nginx container and then forwards the request to one of its backend containers.

To review the steps you have done *the hard way*:

- Determine how many Nginx instances you need.
- Keep starting those instances till you have hit the target.
- Keep track of the internal IP addresses and exposed ports of each instance of the container.
- Configure the load balancer container with the backend container IP and port information.

Remember the preceding steps when running the `IngressController` example and see how easily Kubernetes accomplishes the same goal.

In life, real friends are those who have your back and are there for you when things go bad. Similarly, tools and technology's robustness and reliability are based on how they handle availability when things go wrong.

In this instance, let us list the scenarios where things do not go according to plan:

- Your machine does not have enough resources (CPU and memory) to launch the required number of instances.
- Even if they can launch, they crash, needing you somehow to figure out the IP address of the crashed instance and fix the load balancer config.
- You might kill an instance, and you must repeat the same thing as the previous one.
- If all the instances are at capacity processing data, you must scale up the number of instances to handle the load.
- Once the load has subsided, you must reduce the number of instances.

All the previous steps are just for orchestrating a few instances on a single machine. Imagine the need to do it for thousands of containers across thousands of machines.

Now you know why Kubernetes is required. Kubernetes performs the deployment and takes care of all the details of launching containers, updating iptables, and handling scaling and failures across multiple nodes.

Mesos, Docker Swarm, and Kubernetes

A brief history of container orchestration solutions. Container orchestration became a requirement immediately after Docker made containers popular among developers. Developers wrote their own custom scripts for orchestration while asking the community for reliable, scalable solutions.

The community responded with three main solutions: Apache Mesos, Docker Swarm, and Kubernetes. Mesos was excellent in operations; Docker Swarm excelled in keeping things simple; Kubernetes had the Google pedigree and a huge community backing. After a couple of years of going head-to-head, Kubernetes emerged as the de-facto standard.

Kubernetes primitives

To perform orchestration, Kubernetes has various primitives described in the following sections.

Pods

Pods are the smallest unit of deployment in Kubernetes. Conceptually, you can think of them as individual containers. There are differences between a container and a pod; the main difference is that a pod spec can have multiple containers.

Declarative specification

Kubernetes can be viewed as a declarative engine that takes a specification as input and makes it happen. It differs from the imperative model, where you give exact instructions to an engine on what to do. Python, Java, and C are all imperative programming languages.

Imperative languages are like children, where you must tell them about the exact steps on how to make tea when you want it. If you mess up in the instructions, you get terrible tea at best or a huge mess to clean up at worst.

Declarative engines such as Kubernetes take a different approach. They are more like a restaurant where you specify what you want, tea with or without milk, and professionally made hot masala chai is delivered to you.

Imperative versus the declarative way of getting tea:

Imperative model of getting a tea	Declarative model of getting a tea
Ask a person to make tea using the following instructions: 1. Find tea 2. Boil water 3. Once hot, add two teaspoons of tea powder. 4. After 5 minutes of boiling, add a little milk. 5. Filter tea using a strainer. 6. Pour into a teacup. 7. Bring it to me.	Go to a restaurant. Place an order • Item: tea • Milk: Yes • Count: 1
Get the tea on your table.	Get the tea on your table.
Drink tea.	Drink tea.

The declarative model is more straightforward; however, it has limitations on customization and cost in terms of implementation complexity.

Pod specification

An example of Pod spec is shown as follows:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   labels:
5     app: nginx
6   name: my-nginx-pod
7   namespace: default
8 spec:
9   containers:
10   - image: nginx:1.14.2
11     imagePullPolicy: IfNotPresent
12     name: nginx
13     ports:
14     - containerPort: 80
```

Please see Kubernetes documentation for all the fields you can specify on a Pod. We are listing only those relevant for an interview. This goes for all the sections in this chapter. The key fields are as follows:

```
2 kind: Pod
```

This line tells Kubernetes that you want an instance of a Pod. Because Kubernetes can launch multiple kinds of objects (Pods, ReplicaSets, and Deployments), it needs to be told what kind of an object you want. This differs from Docker, where you mostly run one kind of object, namely, a container. So, the **`kind: Container`** is implicitly assumed.

```
6  name: my-nginx-pod
```

It is your Pod; you get to name it. In Docker, it is optional and mandatory in Kubernetes Pod Spec.

```
10   - image: nginx:1.14.2
```

Under the **`spec/containers`** section, specify the container image you want to launch in the Pod. It is containers plural because Pods can contain more than one container. Here, we use one Nginx image, version 1.14.2

```
13     ports:
14     - containerPort: 80
```

We list the container ports we want to expose under the **`ports`** section. Here it is just one port, namely, Port **`80`**.

It is almost the same minimum fields that you would specify when running a docker container:

```
docker run --name my-nginx --image nginx --port 80
```

You can have multiple containers specified in the spec, and Kubernetes will ensure that either all of them are launched or none. The cool part, you can specify the max CPU and memory this Pod can use. This is super useful in production as now you can predict the maximum number of containers you can stuff on a single node.

It also lets Kubernetes let you know whether you have enough resources to launch your Pod. Pod Spec with CPU and memory limitations are shown as follows:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   labels:
5     app: nginx
6   name: my-nginx-pod
7   namespace: default
8 spec:
```

```
9   containers:
10   - image: nginx:1.14.2
11     imagePullPolicy: IfNotPresent
12     name: nginx
13     ports:
14     - containerPort: 80
15     resources:
16       limits:
17         cpu: 100m
18         memory: 100Mi
```

The highlighted section sets the CPU limit to 100 millicpu (100*1/1000= 0.1). This container is allowed to use only a max of one-tenth of a CPU core. 100Mi means that this container can use a max memory of 100MiB (Mebibytes).

With Pods, we benefit from launching a set of containers automatically. The placement of the Pod is also taken care of by Kubernetes, depending on the resources available on its nodes.

If a node fails, Pod automatically moves to another worker node if possible.

Try it

Register an account at **https://labs.play-with-k8s.com/**, and you can have your Kubernetes cluster with nothing to install on your computer. The instructions in **https://medium.com/swlh/kubernetes-101-play-with-kubernetes-labs-64d1fcbde2f3** give you step-by-step instructions.

For local installation, I would recommend K3sup (pronounced as ketchup) for ease of installation (**https://github.com/alexellis/k3sup#download-k3sup-tldr**) for a Linux host like Raspberry Pi. On Windows or Mac, Docker for Desktop with Kubernetes enabled is easy to get going. Once set up, run the following command to launch your first Pod.

```
kubectl run nginx --image=nginx --port=80
```

The code for this book will be made available in a GitHub at **https://github.com/bpbpublications/Kubernetes-for-Jobseekers. Check out the repo and run.**

```
kubectl apply -f chap-6-pod-defn.yaml
```

Replica sets

This section is presented only to introduce the Replica concept in Kubernetes. ReplicaSets are not generally used anymore; Kubernetes recommends Deployment, which is introduced in the following section.

You would want multiple instances of your Pod for resiliency and scaling purposes. Kubernetes lets you ask for that by specifying a ReplicaSet. The spec for ReplicaSet is given as follows:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: frontend
spec:
# number of replicas you want
replicas: 3
selector:
  matchLabels:
    tier: frontend
template:
  metadata:
    labels:
      tier: frontend
  spec:
    containers:
    - name: my-nginx-with-replicas
      image: nginx:1.14.2
```

The replicas field lets Kubernetes know many Pods Kubernetes should launch. That is it. Launch it using the following commands should show the status:

```
kubectl apply -f chap-6-replicaset.yaml
```

Verify that three pods were launched by typing the following command:

```
$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
frontend-bbq7d              1/1     Running   0          5m13s
frontend-lnflh              1/1     Running   0          5m13s
frontend-qzsb4              1/1     Running   0          5m13s
```

Now, let us have some fun. Let us kill a pod by running the following command:

```
kubectl delete pod/frontend-<pod-id>
```

I chose **frontend-lnflh** as the unfortunate Pod that is going to be terminated:

```
kubectl delete pod/frontend-lnflh
```

Watch as Kubernetes automatically launches a new pod to replace it; when you type **kubectl get pods**, you will get an output similar to the following.

```
NAME                        READY    STATUS     RESTARTS    AGE
frontend-bbq7d              1/1      Running    0           14m
frontend-qzsb4              1/1      Running    0           14m
frontend-t92hl              1/1      Running    0           5m45s
```

Much easier than watching the screen all the time and relaunching the container. Kubernetes shines in orchestrating containers this way.

Deployments

While doing container orchestration the *hard way* using Docker, we did not perform a key use case for container orchestration. The use case is performing upgrades.

Deployment object lets you manage your rollouts, and it has almost all the fields like a replica set with two additional features.

Rollout strategy

You can specify whether you want gradual replacement of existing Pods (default) or shut down all the existing Pods and launch new ones.

Version deduction

This is common for all specs in Kubernetes but is critical for deployment. Kubernetes detect the changes and perform only the difference.

Try it:

Initial deployment specification.

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

```
metadata:
name: nginx-deployment
spec:
replicas: 3
selector:
 matchLabels:
  app: frontend
template:
 metadata:
  labels:
   app: frontend
 spec:
  containers:
  - name: nginx
   image: nginx:1.14.2
   resources:
    limits:
     memory: "100Mi"
     cpu: "100m"
   ports:
    - containerPort: 80
```

It is almost the same as the ReplicaSet definition, except the kind is Deployment. The following command performs the deployment:

```
kubectl apply -f chap-6-deployment-defn.yaml
```

Let us perform an update to the Deployment by using a newer version of Nginx. You can update the deployment using the following command.

```
kubectl set image deployment/nginx-deployment
nginx=nginx:1.20.2
```

Or update the file and then run:

```
kubectl apply -f chap-6-deployment-update-defn.yaml
```

You can see that the new Pods have new versions.

DaemonSets

Deployments are good when you want a set of replicas of your application. What if you wish to run an application on all the worker nodes and only one

instance of it? As Kubernetes is a dynamic environment, the number of nodes cannot be predicted, and ReplicaSets could not guarantee that the pods would be placed on each node even if you did. Kubernetes offers **DaemonSet** as a primitive to launch such applications. Primary use cases are for node monitoring, log collection, and storage management applications that need to be present on all nodes. The following spec launches **DaemonSet** for Nginx:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
name: nginx-daemonset
spec:
selector:
 matchLabels:
  app: lb
template:
 metadata:
  labels:
   app: lb
 spec:
  containers:
  - name: nginx
   image: nginx:1.20.2
   resources:
    limits:
     memory: "100Mi"
     cpu: "100m"
   ports:
    - containerPort: 80
```

You can deploy it by running

```
kubectl apply -f chap-6-daemonset-defn.yaml
```

Here is the output from my session using my local docker-desktop Kubernetes instance, which has only one node.

```
$ kubectl get nodes
NAME             STATUS   ROLES                  AGE   VERSION
docker-desktop   Ready    control-plane, master   20h   v1.22.5
$ kubectl apply -f chap-6-daemonset-defn.yaml
```

```
daemonset.apps/nginx-daemonset created
$ kubectl get daemonsets
NAME                 DESIRED   CURRENT   READY   UP-TO-DATE
AVAILABLE   NODE SELECTOR   AGE
nginx-daemonset   1         1         1
1            1           <none>          6s
$ kubectl get pods
NAME                                 READY
STATUS     RESTARTS   AGE
nginx-daemonset-brfgt                1/1     Running
0          11s
```

Services

While we have all the Pods running, we have yet to launch the *load balancer* equivalent, which we did in orchestration management the *hard way*.

Services act as a *load balancer* for Deployments/ReplicaSets/StatefulSets. Here is the spec for a **`Service`** that does round-robin load balancing for our deployment:

```
apiVersion: v1
kind: Service
metadata:
name: frontend
spec:
selector:
 app: frontend
ports:
- port: 8080
 targetPort: 80
type: NodePort
```

Services find any pods/replica sets deployment that matches the selector labels **`app`**`:` frontend and performs load balancing. Port **`8080`** is the port that the service would serve using the cluster's IP. Any pod in the Kubernetes cluster can access this service using **`ClusterIP:8080`**.

For my docker-desktop, I declare the type as **`NodePort`** so that I can test the service by using the following:

curl localhost:<Node Port shown when I type kubectl get svc/frontend>

A sample session is shown as follows:

```
$ kubectl apply -f chap-6-service-defn.yaml
service/frontend created
$ kubectl get svc/frontend
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE
frontend    NodePort    10.102.187.11
<none>        8080:30245/TCP   8s
$ curl localhost:30245
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
 body {
   width: 35em;
   margin: 0 auto;
   font-family: Tahoma, Verdana, Arial, sans-serif;
 }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the Nginx web server is successfully
installed and working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Ingress controllers

While services are good, they are not enough to provide advanced routing, such as:

- Virtual hosts
- TLS (required for HTTPS)
- Integration with load balancers (both cloud and on-prem such as MetalLB)

This is an absolute requirement when you want to expose your service outside your cluster. Ingress controllers provide that service. The spec is as follows:

```
1 # kubectl apply -f chap-6-ingress-defn.yaml
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: nginx-ingress
6   labels:
7     name: nginx-ingress
8 spec:
9   ingressClassName: nginx
10   rules:
11   - host: nginx.localdev.me
12     http:
13       paths:
14       - pathType: Prefix
15         path: "/"
16         backend:
17           service:
18             name: frontend
19             port:
20               number: 8080
```

```
9   ingressClassName: nginx
```

This tells Kubernetes which ingress controller to use, which in this case is Nginx.

```
10   rules:
```

This is the important section where we add the rules for the Nginx server to use when an HTTP request arrives.

```
11 - host: nginx.localdev.me
12     http:
13       paths:
14       - pathType: Prefix
15         path: "/"
```

This is the meat of the spec. It says Nginx; watch for an HTTP request, which has nginx.localdev.me as the host it is trying to reach. The protocol is HTTP (as opposed to HTTPS). You can have the same Nginx controller serving different paths. `nginx.localdev.me/frontend/` can go to the frontend server, and `nginx.localdev.me/api/` can go to the API server. In this case, we are setting the rules for the root path `/`:

```
16         backend:
17           service:
18             name: frontend
19             port:
20               number: 8080
```

The backend does the actual work and rarely gets the glory, but I digress. This tells the Nginx controller when the request to **http:**/// arrives; it should forward the request to the IP of the Kubernetes service called frontend at Port `8080`.

Try it

First, you must launch the Nginx Ingress controller that can handle traffic routing for the entire Kubernetes cluster, which means it can route traffic to services in any network.

Namespaces allow the segregation of Kubernetes objects such as Pods, ReplicaSets, and so on. It also allows one to set permissions and limits to the namespace and anything running in that namespace. In the examples we have used so far, we have used the namespace `default` for convenience.

I will use the bare metal version of the ingress controller so that the example will work in any environment:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-
nginx/controller-
v1.0.0/deploy/static/provider/baremetal/deploy.yaml
```

Verify that it is running and then launch the following spec:

```
kubectl apply -f chap-6-ingress-defn.yaml
```

You can verify it is working by running:

```
kubectl get svc --namespace=ingress-nginx
NAME                       TYPE        CLUSTER-IP       EXTERNAL-
IP    PORT(S)                          AGE
ingress-nginx-controller
NodePort     10.106.167.167
<none>         80:31699/TCP,443:31806/TCP
```

Note down the port number in the output. **31699** for HTTP and **31806** for HTTPS in my instance. With that port number, we can now see it working by running the following:

```
curl http://localhost:31699 -H 'Host: nginx.localdev.me'
<!DOCTYPE html>
<html>
```

For the HTTPS version:

```
curl --insecure https://localhost:31806 -H 'Host:
nginx.localdev.me'
<!DOCTYPE html>
<html>
<head>
```

As you can see, even though the underlying service has no support for HTTPS, the Nginx **IngressController** provides that service for free.

You are not expected to know how this works (if you do, that is amazing!). You can read **https://docs.k0sproject.io/head/examples/nginx-ingress/** to understand better how this all comes together.

Health checks using liveness, startup, and readiness probes

We saw that Kubernetes services and, by proxy, Ingress Controllers route traffic to the correct pods using selectors. How do they know whether the Pod is ready to serve traffic and whether the application is healthy? They do this by checking the Kubernetes pod **READY** status:

```
kubectl get pods
```

```
NAME                                    READY
STATUS     RESTARTS   AGE
nginx-deployment-5f89495678-927sb   1/1     Running
0          3h49m
```

They direct traffic only to the Pods that are in the **READY** state. For simple applications like ours, which come up fast, the pod launch and start time are close; it is sufficient to use the default Kubernetes state handling. Most real-world applications take some time to start up. How do we tell Kubernetes to wait till the application is live before switching the pod status to **READY**? We tell it by adding the Liveness Probe section in the Deployment/Pod spec:

```
livenessProbe:
 httpGet:
  path: /
  port: 80
 initialDelaySeconds: 30
 periodSeconds: 10
 failureThreshold: 3
 successThreshold: 1
 timeoutSeconds: 1
```

httpGet lets you specify the relative URL to want to check and the port. **initialDelaySeconds** asks Kubernetes to wait for the specified time before running the liveness checks. **periodSeconds** is the period Kubernetes will wait between running the liveness command. **failureThreshold** allows the check to fail the number of times specified in the field before marking the Pod as **NOT READY**. **successThreshold** is the number of times Kubernetes will ensure the check succeeds before marking the Pod as alive. The probe will automatically timeout based on the wait time specified in **timeoutSeconds**.

The readiness probe uses the same parameters as the Liveness probe. The probes can be shell commands also, as follows:

```
readinessProbe:
 exec:
  command:
  - ls
  - /bin/sh
```

Try it

You can see that it works by running the following command.

```
kubectl apply -f chap-6-deployment-readiness-probe-fail.yaml
```

Look at the pod status:

```
kubectl get pods
```

Look at the pod description:

```
kubectl describe pod/nginx-ever-not-ready-6564484d58-p8nxn
```

My session is shown in the following for reference:

```
$ kubectl apply -f chap-6-deployment-readiness-probe-fail.yaml
deployment.apps/nginx-ever-not-ready created
$ kubectl get pods
NAME                                    READY
STATUS    RESTARTS   AGE
nginx-ever-not-ready-6564484d58-p8nxn   0/1     Running
0          4s
$ kubectl describe pod/nginx-ever-not-ready-6564484d58-p8nxn
Name:          nginx-ever-not-ready-6564484d58-p8nxn
Namespace:     default
Priority:      0
Node:          docker-desktop/192.168.65.4
…
Events:
  Type     Reason     Age                 From
Message
  ----     ------     ----                ----                 ----
---
  Normal   Scheduled  43s                 default-
scheduler  Successfully assigned default/nginx-ever-not-ready-
6564484d58-p8nxn to docker-desktop
  Normal   Pulled     42s
kubelet            Container image "nginx:1.20.2" already
present on machine
  Normal   Created    42s
kubelet            Created container nginx
```

```
  Normal   Started    42s
kubelet            Started container nginx
  Warning  Unhealthy  3s (x6 over
41s)  kubelet            Readiness probe failed: cat:
/tmp/nonexistent-file: No such file or directory
```

Physical volumes

Physical volumes let you manage storage in your cluster in a cloud-agnostic manner.

Persistent volume claims (PVC)

Persistent volume claims (**PVC**) gives you an abstraction over the underlying storage provider (Azure, AWS, NetApp, Minio, ceph, NFS, and so on). Your code does not need to change if you change the underlying storage provider. For further details about all the settings for persistent storage, see https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

Try it

Persistent Volume specification that uses the host node storage. Remember, this will work only with single node clusters such as the one using Docker Desktop:

```
# kubectl apply -f chap-6-pvc-defn.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
name: example-pv
spec:
capacity:
  storage: 1Gi
volumeMode: Filesystem
accessModes:
  - ReadWriteOnce
persistentVolumeReclaimPolicy: Delete
storageClassName: manual
```

```
hostPath:
  path: "/tmp/pvc-data"
```

On the host, this creates a folder **/tmp/pvc-data** and limits the storage use to 1Gi.

```
hostPath:
  path: "/tmp/pvc-data"
```

PVC abstracts the persistent volume from the Pod. The following code will still work independently of the underlying persistent volume provider. This powerful abstraction lets your deployment of YAML files be as cloud agnostic as possible.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: example-local-claim
spec:
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 1Gi
storageClassName: manual
```

The **PeristentVolumeClaim** requests 1Gi of storage from the **storageClassName** manual.

Run a deployment that stores custom HTML in this persistent storage which is used by the Nginx Web server using the following command:

```
kubectl apply -f chap-6-pvc-defn.yaml
```

You can verify the results by looking up the service **NodePort** and using **curl localhost:<nodeport>**:

```
$ kubectl apply -f chap-6-pvc-defn.yaml

persistentvolume/example-pv created
persistentvolumeclaim/example-local-claim created
deployment.apps/pvc-deployment created
service/pvc-deployment created
$ kubectl get svc
```

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)             AGE
frontend
NodePort    10.110.95.223    <none>        8080:32575/TCP   9h
kubernetes          ClusterIP
10.96.0.1           <none>        443/TCP          5d19h
pvc-deployment   NodePort    10.100.239.172
<none>         8080:30160/TCP   4s
$ curl localhost:30160
<!DOCTYPE html><html lang="en"><head><meta charset="utf-8">
<title>k8s for job seekers</title></head><body>You got this!
</body></html>
```

PVCs are a powerful abstraction that Kubernetes provides to make your deployments portable across various providers.

Conclusion

We have covered the most used primitives in Kubernetes. Combined with the fundamental knowledge of the operating system, processes, compute, memory, network, and storage containers, we know precisely how Kubernetes executes our wish. The declarative spec is powerful and makes Kubernetes work at scale. Readers are encouraged to read further on the Kubernetes scheduler. Hands-on exercises were done on Kubernetes on the most frequently used primitives for the application deployment lifecycle.

In the upcoming chapter, we dive deep into application deployments. With the need for high availability, the various options that Kubernetes offers to manage deployments will be tried using hands-on exercises.

Points to remember

- A solid understanding of the fundamentals of processes and their resource usage (CPU/RAM/Storage/Network) makes working with the complex ecosystem of Kubernetes a breeze.
- Kubernetes emerged as the winner of the container orchestration battle between Apache Mesos and Docker Swarm.
- Kubernetes does container orchestration across multiple hosts in an automated fashion of what you could do manually.

- Free labs such as **https://labs.play-with-k8s.com/** make Kubernetes accessible to anyone with a browser.
- Kubernetes uses declarative specifications to launch applications and services.
- Declarative code specifies the end state and leaves the implementation to the underlying provider. Imperative code directs the provider on what should be done and in what order.
- The primitives of Kubernetes are Pods, ReplicaSets, Deployments, DaemonSets, Services, and IngressControllers.
- For persistent storage, Kubernetes uses physical volumes and PVC.

Interview questions and answers

1. **Is Kubernetes simple?**

   No. Kubernetes is an extremely complex ecosystem with tools built on top of tools to handle the hard problem of deploying applications at scale.

2. **If Kubernetes is so complex, how does one go about learning it?**

   A solid grasp of the fundamentals of processes, storage, and networking makes understanding Kubernetes a matter of learning the syntax. Learning the syntax and practicing with Kubernetes is how one goes about mastering Kubernetes.

3. **What are the primary primitives of Kubernetes?**

   The primitives of Kubernetes are Pods, ReplicaSets, Deployments, DaemonSets, Services, and IngressControllers.

4. **What are Pods?**

   Pods are the smallest unit of deployment in Kubernetes.

5. **What are ReplicaSets?**

   Replica Sets are a collection of Pods where the number of pod replicas is specified.

6. **What are Deployments?**

   Deployments are used for deploying applications and upgrading them. The upgrade strategy can be specified when replacing an existing deployment with a newer application version.

7. **What are DaemonSets?**

   DaemonSets are used when you want only one instance of an application running on every Kubernetes node.

8. **What are Services?**

   Service act as a proxy routing network traffic between healthy pods of a deployed application.

9. **What are IngressControllers?**

   IngressControllers generally leverage underlying infra providers to expose a Service to the public by using a public internet-facing load balancer. It can also expose a Service via network ports on a node using HostPort. Layer 7 routing (HTTP/HTTPS) support provided by Ingress Controllers via paths enables a single ingress controller for multiple Services.

10. **How do Kubernetes handle persistent storage?**

    Kubernetes abstracts the underlying storage by using Physical Volumes. Persistent volume claims reserve storage within those physical volumes for pods.

CHAPTER 7
Kubernetes Deployment

Introduction

How can one perform zero-downtime upgrades? Although it may or may not be a popular interview question, these shows:

- Interest in getting your opinion from the interviewer.
- From the interviewer's perspective, you have cleared the first round and are ready for advanced questions.

The tricky part of this question is that the answer is no one; I mean, no one has come up with a mechanism that will work 100% of the time for non-trivial applications.

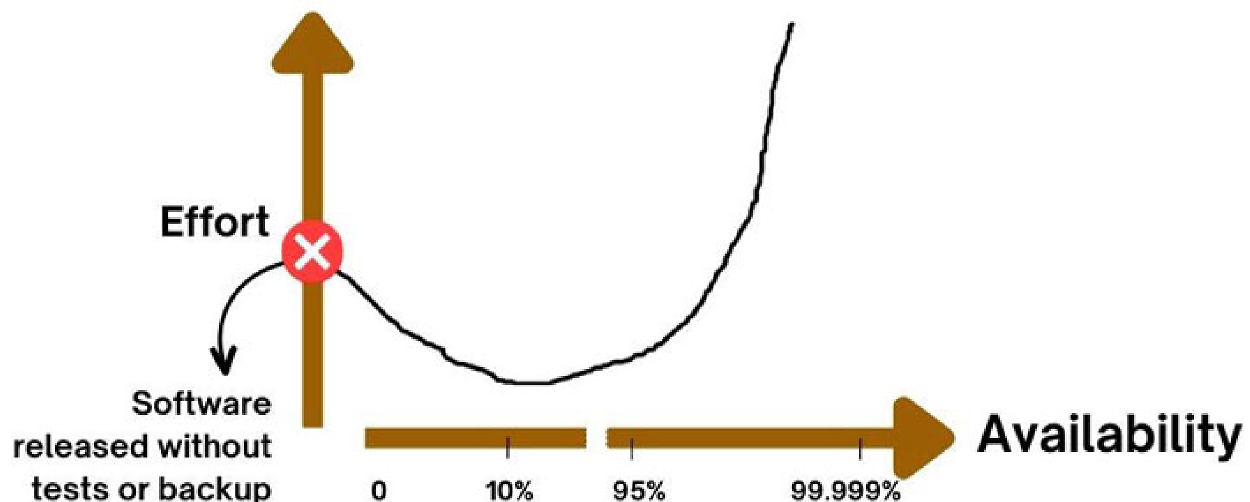


Figure 7.1: *Effort versus availability*

Five nines are the holy grail of software deployments. It means the application is available 99.999% of the time. This translates to the service being down for only around 5 minutes per year.

Compare that to *two nines*, with approximately four days/year of downtime. In today's world of microservices, a service can depend on 100s of services.

Even if each service can boast *five nines*, collectively, the service will have less than that as the probabilities are multiplied.

For just two systems with 99.99 availability, the theoretical maximum availability is as follows:

```
99.99 × 99.99  = 99.98%
```

With more dependent services, the probability drops down pretty quickly.

Going from N–9s to N+1–9s, the effort is exponential, not linear. This means that to make service with 99.9% (**1**–9) availability available for 99.99% (**2**–9s), the effort is not 20 times; it is a minimum of 100 times! To achieve the next level of 3–9s (99.999%), the effort is 1000, and yeah, it quickly becomes super expensive!

By the end of this chapter, you will know the different strategies for deploying applications and the questions to ask the interviewer before giving an answer on the strategy for safely upgrading applications.

Structure

In this chapter, we will discuss the following topics:

- Causes of deployment issues.
- Widely adopted strategies for safe deployments:
  - Blue-green deployment.
  - Canary deployment.
- How Kubernetes helps in implementing these strategies

Objectives

After reading this chapter, you will be able to list the contributing factors for deployment failures. You will be able to explain the different deployment strategies, including their pros and cons. Also, realize the advantages Kubernetes gives you in implementing the deployment strategy.

Consumer expectations

It used to be ok if your computer crashed anytime you pushed it hard, and you waited 10 minutes for it to boot back up. Sure, you cursed it for making

you lose hours of work because you did not save the file, but it was within your expectation.

With free and $1 mobile apps like Fortnite, such an experience is completely unacceptable now. If *free* google search is not working for you, Google's reputation and revenue loss are immense.

These expectations have translated into higher expectations for enterprise software also. While we as consumers are ok with *please hold while I pull your record up*, the enterprise users no longer tolerate this nonsense. They are voting with their pocketbooks and going for cloud-based services with high expectations of speed and reliability.

Cost of doing business

So even though achieving high nines is expensive, current software developers cannot afford to make it an afterthought.

Calculating the availability, you need

Given the high cost, the enterprise must set a goal that is as realistic as possible. Having unrealistic goals lowers employee morale because the team never achieves the specified availability number, and it also makes it very expensive for the consumer.

What are the contributing factors to the lack of availability? These are the main ones:

- Poor coding practices, which include the following:
    - Code assumes a *happy path* only.
    - No retry mechanism is built-in.
    - No back-off mechanism in case of failures (resulting in the dreaded cascading failures)
    - No tests.
    - No health checks.
    - No monitoring support—insufficient or garbage logging.
- Poor configuration management
    - Configuration is manual.

    - Configuration is complex, not documented, and is known to only a select few.
    - Configuration can be changed without any access control.
    - Configuration changes have no review process or guardrails.
- Lack of CI/CD
    - Software is built manually on someone’s desktop.
    - Pipelines exist, but they are mostly *red*.
    - Pipelines are *green*, but they do not have any meaningful tests.
    - Pipelines have thousands of tests, and most of them are flaky and fail unpredictably.
- No backup, business continuance, or disaster recovery strategy
    - No backups exist.
    - Backups exist, but they are not done manually and infrequently.
    - Backups are done through automation, but there are no checks on whether they back up all the required information.
    - Backups are done with all the data, but no one knows how to restore the backup.
    - The documentation to restore exists, but it is out of date.
    - Documentation is up to date, but because no one tried restoring data, they found it incomplete because the person who has access to the DB decryption key is on vacation.
    - There is up-to-date documentation and process to deal with people issues, but the restoration is not tested regularly.
    - The restore test is manual and not automated.
- Security:
    - Systems and data are left open to the internet.
    - Hardcoded passwords
    - No guard rails for engineers to stop them from unintentional errors.
    - No protection against internal attacks.
    - Ransomware friendly.

- Not using TLS (HTTPS)
- No Access control.
- No DDOS avoidance strategy

It should be clear that most deployment issues are due to organizational culture.

It is a very dangerous sign that a company punishes the person committing a deployment error. The lack of investment by the company in the needed effort for high availability caused the error. Not the hapless person who types the wrong value as 3 a.m. while trying to get the system up and running.

When it is your turn to ask questions to the interviewer, ask them if they have a *Blameless Retrospective* culture. If the answer is what that or of some form is, *it is good in theory, but if we hold people accountable*, then be aware of what company you are getting into.

Kubernetes and deployments

How can you perform zero-downtime upgrades?

It should be clear that there is no short answer to this. To achieve this pinnacle of software delivery, the company must have the right culture, people, product, and customer expectations. The company having the right technology is the last and minor requirement.

I have seen highly available software running on bare metal, virtual machines, and the cloud. A well-written software adapts nicely and is written with these philosophies in mind:

- Do one thing and do it well.
- Be liberal in what you accept and conservative in what you omit (nice, being liberal in what you accept means that you expect malformed input and handle it accordingly).
- Make the easy thing to do the right thing to do.

So how does technology, specifically Kubernetes, help?

Kubernetes has supported well-known industry-accepted methodologies that help in getting close to achieving zero downtime.

The scenarios we will see in this, and upcoming sections are how to achieve those patterns using Kubernetes. The focus is on those scenarios, as most system downtime happens during upgrades.

Higher availability with stable software is possible, if not for the pesky developers who want to achieve value to the customers in bug fixes and new features ☺.

Blue-green deployment

For a detailed look into blue-green deployment, see https://codefresh.io/blue-green-deployments-kubernetes/. A typical blue-green deployment diagram:

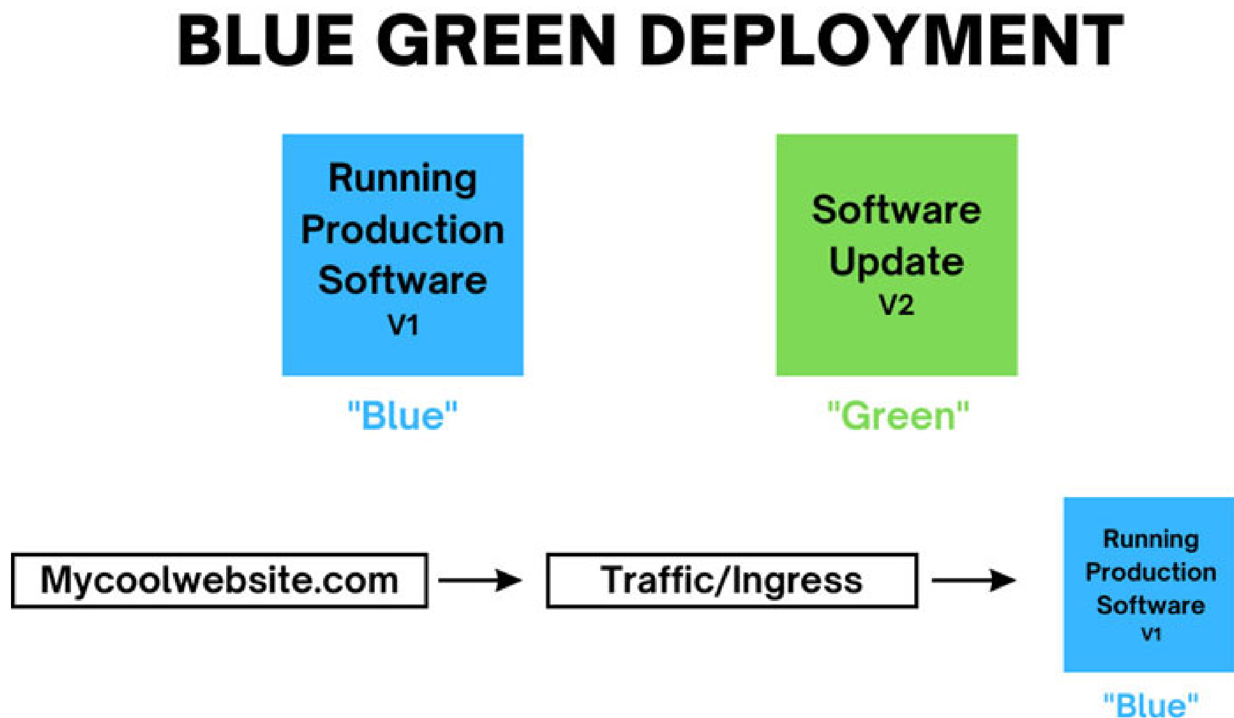


Figure 7.2: *Blue-green deployments*

Try it

Launch the **`blue`** version by running the following:

```
kubectl apply -f chap-7-blue-deployment-defn.yaml
```

The spec has a version label to denote whether it is blue or green:

```
Selector:
 matchLabels:
   app: frontend
   version: blue
```

Launch the service frontend with a selector to match the `version: blue`:

```
kubectl apply -f chap-7-blue-svc-defn.yaml
```

```
spec:
 selector:
     app: frontend
version: blue
```

Verify that the app works:

```
$ kubectl get svc
NAME     TYPE    CLUSTER-IP   EXTERNAL-IP  PORT(S)     AGE
frontend
NodePort  10.110.95.223  <none>    8080:32575/TCP  34h
$ curl localhost:32575
<!DOCTYPE html>
…
```

Let us intentionally launch a `green` version that we know will fail:

```
Kubectl apply -f chap-7-fail-green-deployment.yaml
```

The spec has the version label set to `green`:

```
Selector:
matchLabels:
 app: frontend
 version: green
```

As expected, the pods are not marked as `READY`:

```
Kubectl get pods
NAME                         READY  STATUS  RESTARTS  AGE
nginx-blue-deployment-5778cb9488-4ssqj    1/1   Running  0
30s
nginx-blue-deployment-5778cb9488-68zwm    1/1   Running  0
30s
nginx-blue-deployment-5778cb9488-pf5tj    1/1   Running  0
30s
```

```
nginx-fail-green-deployment-db478459d-jfbqs  0/1   Running  0
6s
```

You can verify that the app still works by running **curl localhost:<your service NodePort>**. To ensure that the deployment is bad, let us ask the service to direct the traffic to the bad Pods:

```
Kubectl apply -f chap-7-green-svc-defn.yaml
```

Verify that the app does not work by running **curl localhost:<your service NodePort>**:

```
curl localhost:32575
curl: (52) Empty reply from server
```

You can see the benefit of the blue-green deployment. Even though we switched to the green deployment by mistake, we can quickly change it back to blue since we have not deleted the blue deployment:

```
Kubectl apply -f chap-7-blue-svc-defn.yaml
```

90% of the time, it is the wrong configuration setting!

Verify that the app works:

```
Curl localhost:32575
<!DOCTYPE html>
…
```

Let us upgrade the blue deployment to green deployment:

```
Kubectl apply -f chap-7-green-deployment-defn.yaml
```

The image and version labels differ between blue and green deployments:

```
Labels:
  app: frontend
  version: green
spec:
containers:
- name: nginx
  image: nginx:1.20.2
```

Verify that the pods are **READY**:

```
Kubectl get pods
NAME                       READY   STATUS   RESTARTS   AGE
```

```
nginx-blue-deployment-5778cb9488-4ssqj  1/1   Running  0
2m17s
nginx-blue-deployment-5778cb9488-68zwm  1/1   Running  0
2m17s
nginx-blue-deployment-5778cb9488-pf5tj  1/1   Running  0
2m17s
nginx-green-deployment-58f4b66f7b-44996  1/1   Running  0
6s
nginx-green-deployment-58f4b66f7b-g5fj5  1/1   Running  0
6s
nginx-green-deployment-58f4b66f7b-pzxjl  1/1   Running  0
6s
```

Now that we have verified that they are **READY**, we can switch the traffic to the green deployment:

Figure 7.3: *Traffic switch to "green"*

```
kubectl apply -f chap-7-green-svc-defn.yaml
```

Verify that the service still works by running:

```
curl localhost:32575
```

It should return to the Nginx welcome page. Now, we can delete the blue deployment as we no longer need it:

```
kubectl delete -f chap-7-blue-deployment-defn.yaml
```

You can verify that the app still works.

For the next upgrade, we deploy the **`blue`** version, and if it succeeds, switch to it and delete the **`green`** version. The application lifecycle thus continues.

Canary deployment

Sometimes it is hard to create a good replica of a running deployment. The reasons can be that it is expensive, too many dependent services also need a replica, and so on.

This method is prevalent in internet-scale companies. Instead of a full-blown replica, a small set of instances are brought up, and a small portion of the traffic is diverted to them. If everything works fine, more instances are added, and the old versions are brought down gradually.

Try it

Let us launch the production deployment with three instances:

```
kubectl apply -f chap-7-prod-for-canary-test-deploy-defn.yaml
```

It is our regular deployment of Nginx with three instances. Launch our service to round-robin traffic among the instances:

```
kubectl apply -f chap-7-canary-test-service-defn.yaml
```

Verify that it works.

```
curl localhost:32575
<!DOCTYPE html>
…
```

Launch the canary that we will know will fail:

```
kubectl apply -f chap-7-fail-canary-deployment.yaml
```

Please wait for it to be **`READY`**:

```
kubectl get pods
NAME                                 READY   STATUS    RESTARTS   AGE
fail-canary-deployment-56f8858668-twnv4       1/1
Running   0      8s
nginx-prod-for-canary-test-deploy-656c99555-f8d9d   1/1
Running   0      12m
```

We know that it will fail to return the welcome page because of these lines. These lines remove the **index.html** file from the Nginx **html** folder. Without the file to serve, Nginx will throw an error:

```
initContainers:
  - name: fail-canary-deployment-init
  image: busybox
  command:
    - sh
    - -c
    - touch /usr/share/nginx/html/index.html && rm
    /usr/share/nginx/html/index.html
```

Let us verify that we do get an error from the canary instance. Please substitute **<32575>** with the port where your application is listening:

```
for VAR in {1..12}; do curl localhost:32575 2>&1; done | grep
Forbidden
<head><title>403 Forbidden</title></head>
…
```

To get an error, we have to make sure that we hit the service multiple times so that some requests go to the *bad* deployment.

We will delete the canary deployment now that we know it is a bad deployment:

```
kubectl delete -f chap-7-fail-canary-deployment.yaml
```

Verify that things are back to normal:

```
for VAR in {1..12}; do curl localhost:32575 2>&1; done | grep
Forbidden
$
```

The curl in the loop should return empty. Now, you can launch the upgrade that we know will succeed:

```
kubectl apply -f chap-7-canary-single-instance-defn.yaml
```

Verify that the app still works using curl. Now that the app works, you can scale it up using our previous chapter’s upgrade spec:

```
kubectl apply -f chap-6-deployment-update-defn.yaml
```

Verify that the app works using curl. You can now delete the previous deployment:

```
kubectl delete -f chap-7-fail-canary-deployment.yaml
```

Verify that the app still works using curl. You have successfully tried out canary-style deployment.

With proper health checks, the Kubernetes rolling updates strategy does the same as we did manually. It replaces one instance, sees if it works, and then replaces other instances. See https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment.

Comparison of canary and blue-green deployment

The following are a few salient differences between canary and blue-green deployments:

Canary deployment	Blue-green deployment
Less expensive as only a small number of instances are introduced with duplication of infrastructure required.	The production environment must be duplicated into the *green* deployment requiring twice the resources temporarily.
Architecturally more expensive as the application must be designed so that two versions can run simultaneously.	Since the resources are duplicated, the *green* version will not be exposed to production traffic until it is ready. Syncing the changes between the two environments still requires work.
Sticky sessions are required to ensure that only new sessions are sent to the canary server. If they are not, customers who added items to the cart using the prod servers and now click the buy button could be sent to the canary server, which might fail, resulting in a bad customer experience.	Production traffic does not hit the *green* environment; hence, the applications can be configured with whatever the company chooses.
The canary server might work fine, but the new version might fail when multiple instances are introduced.	All the instances would be deployed before the switch, so any issues with multiple instances would be discovered earlier.

Table 7.1: *Differences between canary and blue-green deployments*

Conclusion

Upgrades are hard and the source of many outages. Kubernetes allows multiple strategies to be implemented to detect errors early and even when

errors occur to limit the blast radius. It also provides mechanisms to roll back to the working version safely.

Contributing factors to loss of availability due to cultural issues were identified, along with process issues.

The difficulty in upgrading stateful applications where data is backed by databases was made clear. We used hands-on experiments to perform the most common upgrade strategies, namely, blue-green and canary deployments, using Kubernetes.

In the upcoming chapter, we will learn about Networking in Kubernetes. The topics to be covered are as follows:

- Default networking setup in Kubernetes.
- Network plugins and why we need them.
- Network policy with hands-on exercises.

Points to remember

- Zero downtime upgrades for non-trivial applications are hard.
- Picking the right amount of availability goals for service is critical for productivity.
- To achieve the availability goals, the enterprise culture must prioritize making resilience a priority.
- Kubernetes helps achieve the enterprise's availability goals by providing support for different deployment strategies, such as blue/green deployment, rolling upgrades, and canary deployment. Built-in support for health checks, pod migration, and network routing ensure that the application teams focus on application-level support for resiliency and observability.

Interview questions and answers

1. **What are the contributing factors for application downtime?**

   A culture that rewards feature delivery over code that reduces rework, poor coding practices, poor configuration management, lack of CI/CD, Backup and DR not tested regularly, and Security as an afterthought.

2. **What different types of deployments do you know for safe upgrades?**

   Blue/green deployment, where upgrades are done using a copy of the production environment. If everything works well, production traffic is shifted to the upgraded version. Production traffic is shifted to the previously working version if serious errors occur.

   Canary deployment is where a small portion of the traffic is directed toward the upgraded instances, and if there are no errors, old instances are gradually replaced with new ones.

CHAPTER 8
Kubernetes Services

Introduction

We will dig deeper into the services that Kubernetes provides in networking, storage, application/node management, and GitOps. No application is helpful if all it can do is talk to itself. To be beneficial at the minimum, an application takes some input and produces an output. An application becomes much more useful if it can make the same service available over a network.

On the people side, you might have heard the term networking effect. Research says that the impact of a person/software/service is proportional to the square of the consumers/nodes in their network. WhatsApp and Twitter's value is the square of the number of people in their network. This high impact is called the networking effect.

When we looked at Kubernetes services in the Kubernetes basics chapter, we saw that it kept track of Pod IPs and ports. You might also remember that it is a hard thing to do when you did it using the *hard way*. Without Kubernetes providing this abstraction, running pods over multiple servers would be tough.

Networking in Kubernetes requires a whole book on its own. We will be focusing on the questions that would be asked in an interview. With that in mind, you will learn about networking plugins in Kubernetes and network policy with a hands-on exercise. A deeper look into storage, application/node management, and the hottest topic in DevOps—GitOps.

Structure

In this chapter, we will discuss the following topics:

- Networking plugins
- Network policy
- Storage types
- Application management
- Node management

- GitOps

Objectives

After reading this chapter, you will be able to answer questions on why networking plugins are required and how to apply a network policy to provide access control to applications. You will have answers to questions about the different types of storage and which storage is best suited for what applications. What is GitOps and how GitOps uses application/node management features of Kubernetes for deployments using code will also be clear.

Network plugins

Although the default networking abstractions are good enough, Kubernetes architecture is built to replace the default networking implementation with other vendors providing advanced functionality.

Networking plugins are Calico, Flannel, Open V Switch (OVS), Weave, AWS VPC CNI, Azure CNI, etc.

Why should one require to use these plugins when the default works?

- Used by all cloud providers (Azure, AWS, and Google) to better integrate with their native networking solutions.
- Better network management.
- Better integration with underlying network hardware than the default network.
- Support for advanced network policy.
- Plugins like Cilium can provide advanced network security and monitoring.

Because all cloud solutions provide preconfigured network plugins, we need to be aware that these plugins exist and know how to implement network policy on top of them.

Container network interface (CNI)

CNI is implemented by the default network that uses iptables and NAT services the host OS provides. CNI contains the specification and libraries for writing plugins to configure network interfaces. As the default implementation is a software implementation, it has excellent flexibility and compatibility but consumes resources. There is little isolation between Pod-to-Pod traffic, control plane traffic, and host traffic.

Plugins offer seamless integration with underlying network hardware and thus can provide better security and management. For example, Azure CNI would create Azure native VNETs (Virtual NETwork) for Pod-to-Pod traffic and therefore get as good performance as an Azure VM.

Networking policy

In Kubernetes, there is a single network for all pods. So, a Pod can talk to any other pod in the same cluster.

Namespaces isolate a set of services, and RBAC rules can be applied to those who can access the namespace. However, since the IP address is shared, any Pod can talk to another pod in a different namespace if the Pod IP address is known.

Kubernetes introduced a Network Policy object to tighter control which entities a Pod can talk to. The network policy is specified in the following code:

```
# you need a CNI plugin for your cluster installed to make this
work.
# All major public cloud providers have the CNI plugin configured by
default.
# the following should work there.
# kubectl apply -f chap-8-simple-network-policy.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-http-https-from-prod-frontends-policy
spec:
  ingress:
- from:
  - podSelector:
    matchLabels:
      app: frontend
      env: production
  ports:
  - port: 443
  - port: 80
```

You specify ingress (incoming network traffic), egress (outgoing network traffic), and rules using Network Policy. In the previous sample, we specify that incoming

traffic to Ports **`443`** and **`80`** from pods with labels **`app=frontend`** in namespaces with labels **`env=prod`** will be allowed.

Try it

We will try it using an excellent visual editor provided by Cilium https://editor.cilium.io/. See also excellent tutorial materials at https://github.com/networkpolicy/ to learn more.

On your browser, type https://editor.cilium.io/. You will be presented with a welcome page:

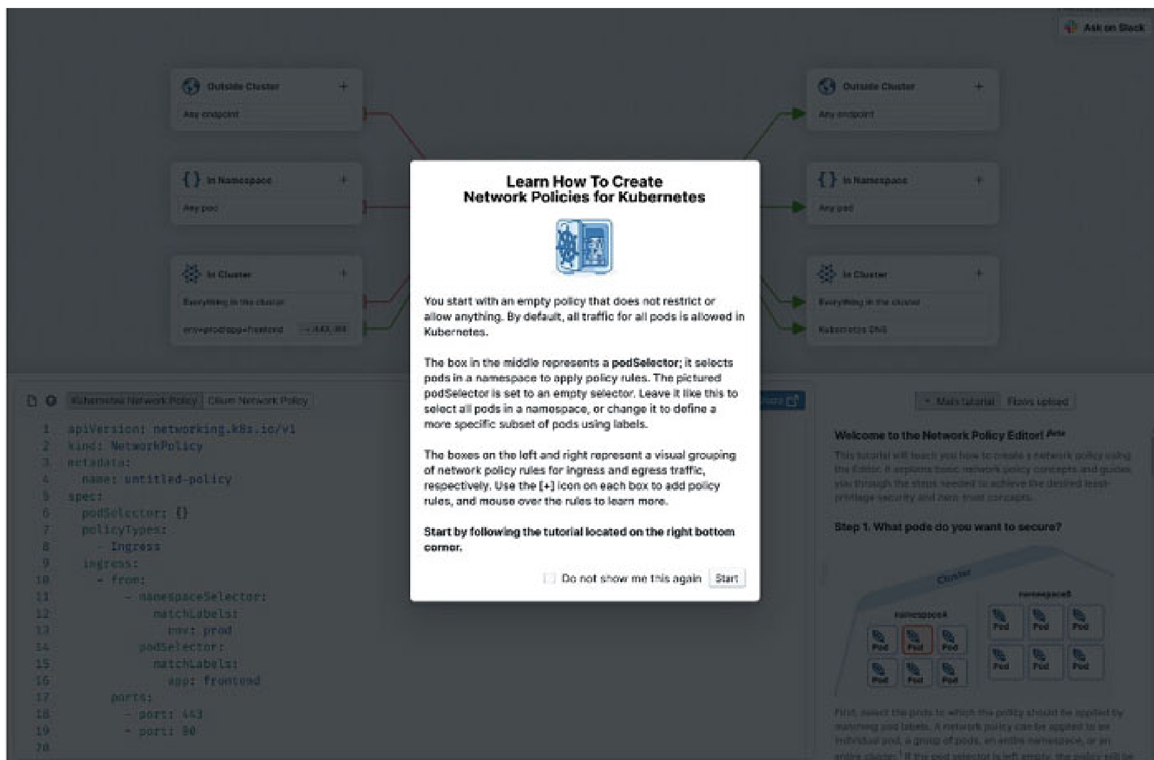


Figure 8.1: *Cilium network policy editor*

Click on **`Start`**:

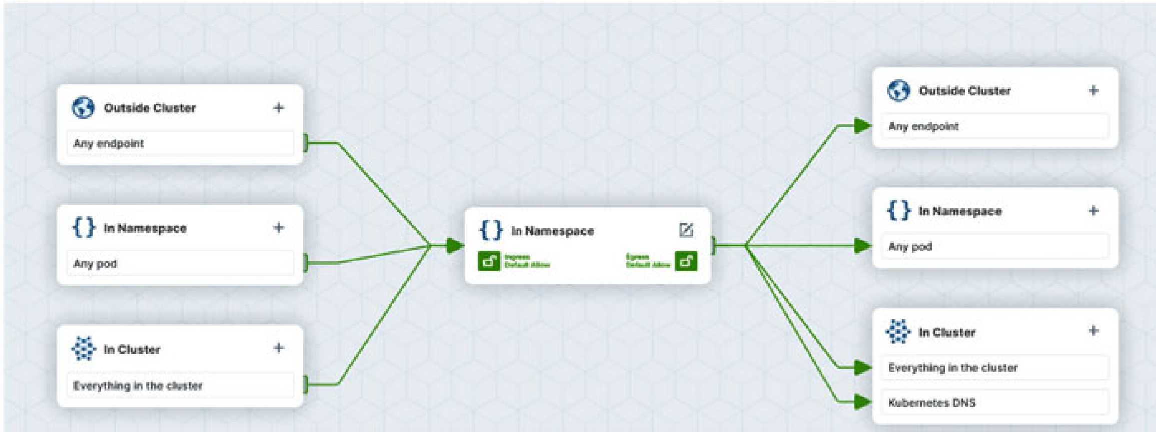
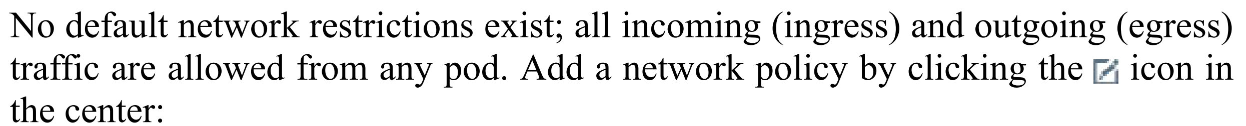


Figure 8.2: *Default all allowed network policy*

No default network restrictions exist; all incoming (ingress) and outgoing (egress) traffic are allowed from any pod. Add a network policy by clicking the  icon in the center:

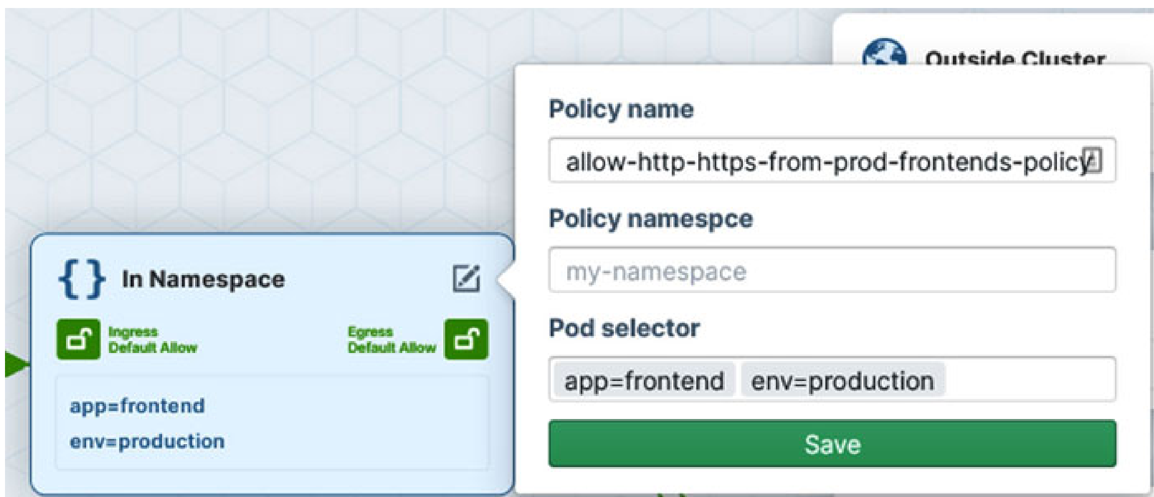


Figure 8.3: *Add network policy*

Let us restrict incoming traffic to coming from Pods with labels, **`app=frontend`** and **`env=production`**:

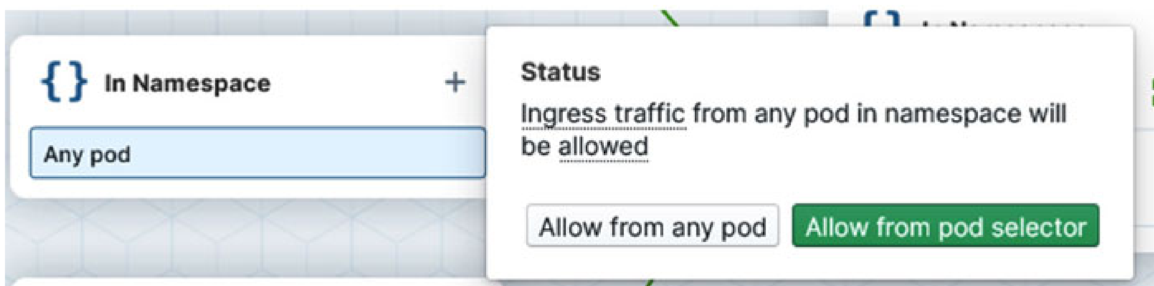


Figure 8.4: *Select Allow from the Pod selector*

Shift to the box on the left and click on **`Any pod`**. Then, click the Allow from pod selector on the **`Ingress`** section of the page.

Figure 8.5: *Add network policy rule for HTTP/HTTPS traffic*

Click on Add rule after typing **`app=frontend`** and **`env=production`** in the Pod Selector:

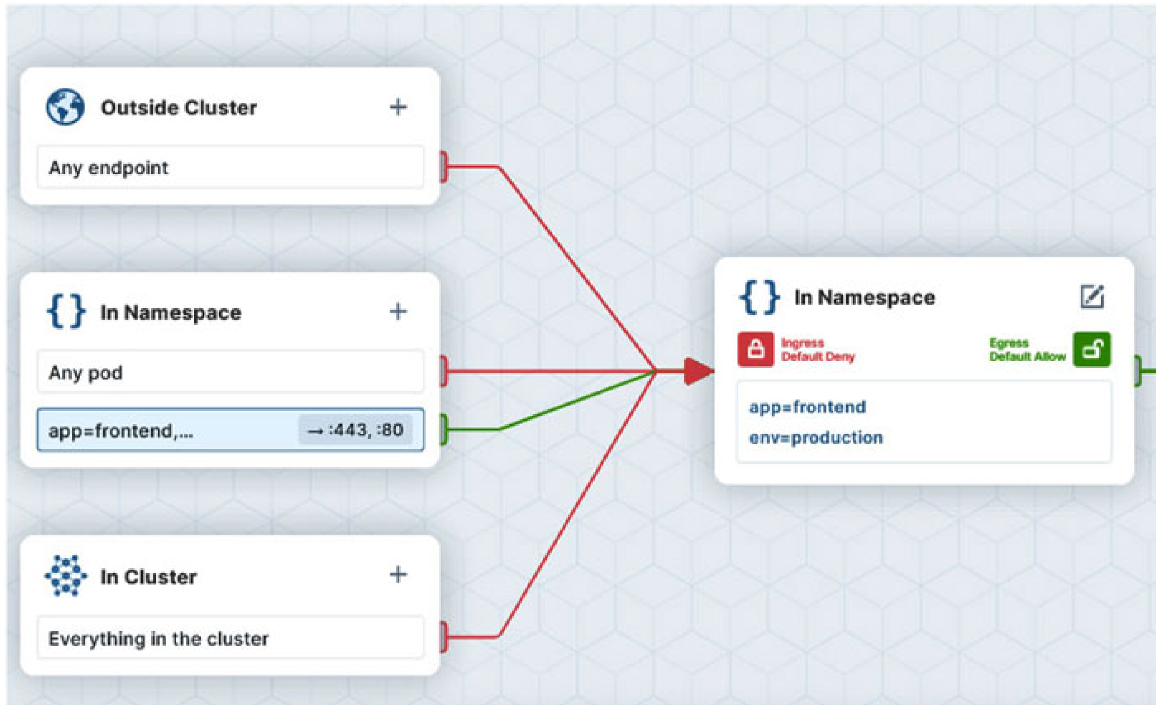


Figure 8.6: Network policy visualized

Shows the effect of your policy in a visual format. The resulting **`NetworkPolicy`** is shown as follows:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-http-https-from-prod-frontends-policy
spec:
  podSelector:
matchLabels:
  app: frontend
  env: production
  ingress:
- from:
  - podSelector:
    matchLabels:
      app: frontend
      env: production
  ports:
  - port: 443
  - port: 80
```

The preceding specification enforces the network policy of allowing only the front-end production pods to accept HTTP and HTTPS traffic. The enforcement uses the pod labels and expects front-end production to be labeled with the `app: frontend` and `env: production`. On every other Pod, no ingress traffic will be allowed.

Why the network policy that we implemented is important? Policies like these can provide critical protection for clusters used in the multi-tenant solution.

Storage

Block storage is the mother of all storage. You might know it as the `C:` drive. All bare metal hard drives are block storage devices. Complex storage systems with **Redundant Array of Independent Disks** (**RAID**) capabilities are still block storage devices. SCSI drives, SATA/SSD drives, and SD cards are all block storage devices.

Both old and new file systems (FAT, NTFS, ext4, and ZFS) assume that the underlying devices are block devices. The protocol has been around for a long time and is stable. The devices are mostly highly reliable, and redundancy/failure handling is done through abstractions such as RAID.

Hard drives are divided into enterprise-class and consumer-class devices. Consumer-class devices are way cheaper than enterprise ones. You might think consumer drives will fail faster than enterprise drives. The opposite is true. Enterprise-class drives die more quickly as the hard drives assume they are part of a RAID system. Failing spectacularly rather than underperforming is more critical. The data delivery continues without interruption or slowness because of the RAID functionality. (Look up a company called BackBlaze if you are really into hard drives).

Funner Link: Traditional Hard drives with spinning disks are sensitive to vibrations. See this video (**https://youtube/tDacjrSCeq4**) on why they do not like being yelled at! (Understanding Brendan Gregg's posts [**https://www.brendangregg.com/**] on-site reliability alone should get you a job in this area).

Main limitations of block devices

Block devices can be attached to only one server at a time. Imagine a physical drive with a cable attached to a computer. Even though they might be accessing

storage through a network (like ISCSI), a block device is tethered to the computer conceptually. This device cannot be directly shared with any other computer.

Conceptually, you must remove the cable and attach it to another server. The other server must then scan for block devices and *import* them before using them.

Advantages of attachment

When something is dedicated to you, there is less contention for the resource and less chance of corruption of files. Physically attached storage gives the best performance possible.

Network-attached storage

The advancements in networking and the need for storage to be shared across multiple servers created networked file systems such as **Network File System** (**NFS**) and **Common Internet File Systems** (**CIFS**/SAMBA). They can be attached to multiple servers using regular ethernet networking.

NFS is available in Linux/Unix environments. CIFS protocol drives all Windows networked storage and can be served from Linux servers using SAMBA software. Direct attached storage using block devices and NFS/CIFS storage shared among many servers are the two main types of storage enterprises use.

Networked storage adds an abstraction layer above the underlying block devices. Hence, resizing storage is super easy.

Main limitations of NAS

NFS has security concerns due to its use of insecure Remote Procedure Calls. CIFS ports could be left open to the world or access control misconfigured, allowing unfettered access to all the data. They can fail in unpredictable ways due to potentially multiple network hops needed to access the file system. Network bottlenecks can create a slow performance. Locks on files can occur and breaking them can be very hard. They don't perform that well with small files or huge number of nested folders.

Object storage

Object storage was available in general, but S3 made them super famous. S3 (Simple Storage Service = S'3') provided simple ways to **`PUT`** an object and **`GET`** an object or blob via HTTP protocols. They are well suited for storing large amounts of data that do not often change, like photos, videos, and so on.

All the tricks of HTTP load balancing can be applied to store objects reliably. Many systems have deduplication built-in (helpful in reducing storage needs in uncompressed backups).

Physical volume claims

Physical volume claims (**PVC**) let you specify the type of storage you want, abstracting the underlying provider to the pods which mount the storage.

They also handle attaching and destaching storage to the nodes. The type of storage you want to attach to you depends on the Pod's storage load characteristics.

You probably should choose a block storage backend if you need high-performance and random access (like a storage backend for the database). If the need is access to the same data that frequently changes, like configuration, static files, images, or videos, network-attached storage might be better.

Object storage would be a better option for storing blobs that you want redundantly stored or the automatic movement to cheaper storage based on age. Blob storage makes almost *infinite* storage available.

You might notice the hedging in the preceding recommendations with words such as *probably*, *might be*, and *would be*. Storage over the network is highly unpredictable, especially in a cloud environment. You must run it with an actual load to determine your best option.

In an interview, you can be confident that physically attached storage gives the best possible performance, especially for random access. Resizing after the fact is a pain, however.

Blob storage is more reliable than others (when backed by the cloud vendor).

NFS/network-attached block devices are in the middle regarding scalability (size-wise).

> **Some companies will launch multiple VMs for you in the cloud to find which ones have good storage performance and provide them for you to use.**

Node management

Worker nodes are where the pods are mostly run. It is tough to emulate them on your laptop (not impossible see

https://minikube.sigs.k8s.io/docs/tutorials/multi_node/). The primary Kubernetes server and the worker nodes are never the same in production.

In Azure, you pay only for the worker nodes. From a cost perspective, it is essential that you carefully choose the sizes of the VMs. Kubernetes provides the ability to label the nodes, and most cloud vendors allow you to mix and match different-sized VMs within the same cluster.

A suitable scaling algorithm can also dynamically scale the cluster based on load. For example, you can mark specific nodes as *high-memory and high-CPU*. By adding node selectors in your deployment definition, you can make your *high-memory* apps run only on *high–memory* labeled nodes.

Try it

We will use **https://www.katacoda.com/courses/kubernetes** and follow the `Launch a multi-node cluster using Kubeadm` scenario.

Figure 8.7: *Katacoda Start Kubernetes Cluster Scenario*

Follow the instructions to launch a multi-node cluster:

1. Initialize the primary node.
2. Deploy **Container Networking Interface (CNI)**—Weave in this case.
3. Wait till the pods are up and running:

```
controlplane $ kubectl get pod -n kube-system
NAME                                  READY   STATUS    RESTARTS   AGE
```

```
coredns-fb8b8dccf-9dm87          1/1   Running  0     92s
coredns-fb8b8dccf-jkqd7          1/1   Running  0     92s
etcd-controlplane                1/1   Running  0     44s
kube-apiserver-controlplane      1/1   Running  0     28s
kube-controller-manager-controlplane   1/1   Running  0     34s
kube-proxy-9vb9s                 1/1   Running  0     92s
kube-scheduler-controlplane      1/1   Running  1     26s
weave-net-qnszr                  2/2   Running  0     18s
```

4. Join the cluster.
5. View nodes:

```
$ kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
controlplane  Ready    master   4m33s   v1.14.0
node01        Ready    <none>   23s     v1.14.0
```

6. Deploy the test pod.
7. Launch the dashboard.
8. You should see a nice dashboard like the following:

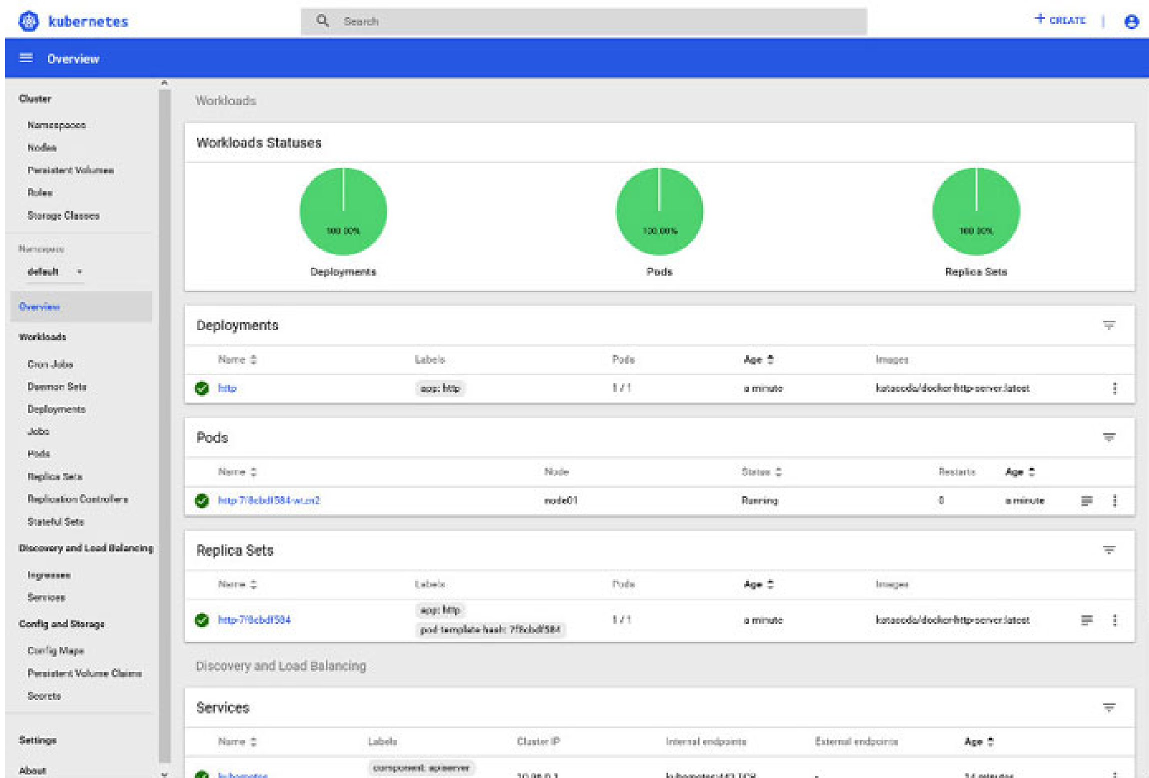


Figure 8.8: *Katacode Kubernetes cluster dashboard*

9. Increase the number of replicas to 3 by running the following command:

   ```
   kubectl scale --replicas=3 deployment http
   ```

10. Run the following command to see that all the pods are running on **node01**:

    ```
    kubectl get pods -o wide
    ```

    You should get an output similar to this:

    ```
    NAME             READY   STATUS   RESTARTS   AGE    IP       NODE
    NOMINATED NODE  READINESS GATES
    http-7f8cbdf584-b5jz4  1/1   Running  0
    5m25s  10.32.0.196  node01  <none>      <none>
    http-7f8cbdf584-pg66h  1/1   Running  0
    5m25s  10.32.0.195  node01  <none>      <none>
    http-7f8cbdf584-wtzn2  1/1   Running  0     12m
    10.32.0.193  node01  <none>      <none>
    ```

11. You will notice that no pods are scheduled on the **controlplane**. **kubeadm**, which sets up the cluster, marks the primary **controlplane** as non-schedulable. Let us change that and make it schedulable:

    ```
    kubectl taint node controlplane node-
    role.kubernetes.io/master:NoSchedule-
    ```

12. The exercise of scaling down the replicas and scaling them up again to see the pods scheduled in the **controlplane** node is left to the reader. You should see an output that is like the following.

    ```
    NAME             READY   STATUS   RESTARTS   AGE   IP        NODE
    http-7f8cbdf584-7lg5k  1/1   Running  0     16s   10.32.0.4
    controlplane
    http-7f8cbdf584-cgzv5  1/1   Running  0
    16s  10.32.0.195  node01
    http-7f8cbdf584-wtzn2  1/1   Running  0
    13m  10.32.0.193  node01
    ```

We have covered the basics of node management. With the taint node capability, you can cordon problematic nodes for debugging. In deployment specifications, you can specify **NodeAffinity** to ensure that the pods are placed the way you want. For example, you can ensure, that pods of the same application are not placed on the same node.

Configuration maps and SShh… secrets

A critical step in launching an application is its configuration. The 12-factor app (https://12factor.net/config) recommends a strict separation of config from code. We will see how we accomplish this in Kubernetes. Let us take an example of an application that needs to talk to a database.

At the minimum, you need configuration for the DB name, connection info, and password (which should not be stored in plain text). This application needs to talk to different databases depending on its environment. It should talk to the development database using the **`dev`** DB password for development environments. For test environments, it should talk to the test database using the **`test`** DB password (which is different from the **`dev`** DB password). Kubernetes gives two primitives for handling these two use cases.

> **(Note that you can still use `ENV` vars for configuration but using `ConfigMap` is better.)**

`ConfigMap` is used for configuration settings such as DB connection string. Kubernetes Secret is used for storing the DB password.

To specify the DB connections, the string creates a **`ConfigMap`** as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
  labels:
    app: db
    env: dev
data:
  db_connection_string: "postgresql+psycopg2://app_user:
  {password}@dbhost/mydb"
  db_user: "app_user"
  db_host: "dbhost"
  DB_NAME: "mydb"
```

The values under data are key: value pairs stored in Kubernetes.

Note that we have a placeholder for the DB password as **`ConfigMaps`** are not encrypted, and we do not store any secrets here. Apply the config by running the following command:

```
kubectl apply -f chap-8-db-user-pod.yaml
```

Let us mount it on a pod and see how it can be used in a pod. You can load the secrets as follows:

- ENV vars
- Files

The following spec loads it as both:

```
#kubectl apply -f chap-8-db-user-pod.yaml
apiVersion: v1
kind: Pod
metadata:
 name: configmap-demo-pod
spec:
 containers:
- name: demo
 image: alpine
 command: ["sleep", "3600"]
 env:
 # Define the environment variable
 - name: DB_USER # Notice that the case is different here
       # from the key name in the ConfigMap.
  valueFrom:
  configMapKeyRef:
   name: db-config # The ConfigMap this value comes from.
   key: db_user # The key to fetch.
 - name: DB_HOST
  valueFrom:
  configMapKeyRef:
   name: db-config
   key: db_host
 volumeMounts:
 - name: config
 mountPath: "/config"
 readOnly: true
 volumes:
# You set volumes at the Pod level, then mount them into containers
inside that Pod
- name: config
 configMap:
 # Provide the name of the ConfigMap you want to mount.
 name: db-config
 # An array of keys from the ConfigMap to create as files
 items:
```

```
  - key: "db_connection_string"
    path: "db_connection_string"
  - key: "DB_NAME"
    path: "db_name"
```

Get the Pod running:

```
kubectl apply -f chap-8-db-user-pod.yaml
```

We can verify it by running the following command:

```
kubectl exec -it configmap-demo-pod -- sh
```

Here is my session for checking whether it is loaded as expected. Please note that the commands are executed in the shell opened by the previous command:

```
# env
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOSTNAME=configmap-demo-pod
SHLVL=1
HOME=/root
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
DB_HOST=dbhost
DB_USER=app_user
# cat /config/db_connection_string
postgresql+psycopg2://app_user:{password}@dbhost/mydb/ #
# cd /config
# ls
db_connection_string db_name
# cat db_name
mydb
# exit
```

Now to the more complex problem of secrets. Before loading your secret, you need to convert it into ASCII by encoding it using the **base64** command:

```
echo -n 'My$u&4du5er$ecr%8' | base64
```

```
TXkkdSY0ZHU1ZXIkZWNyJTg=
```

We can use that to define our secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-password
data:
  password: TXkkdSY0ZHU1ZXIkZWNyJTg=
```

Load the secret by running the following command:

```
kubectl apply -f chap-8-db-password-pod.yaml
```

The following is the new pod spec that will load `Secret` and the `ConfigMap`:

```
#kubectl apply -f chap-8-db-password-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-secret-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # Define the environment variable
        - name: DB_USER # Notice that the case is different here
          # from the key name in the ConfigMap.
          valueFrom:
            configMapKeyRef:
              name: db-config # The ConfigMap this value comes from.
              key: db_user # The key to fetch.
        - name: DB_HOST
          valueFrom:
            configMapKeyRef:
              name: db-config
              key: db_host
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-password
```

```
          key: password
      volumeMounts:
      - name: config
        mountPath: "/config"
        readOnly: true
      - name: secrets
        mountPath: "/etc/secrets"
        readOnly: true
  volumes:
   # You set volumes at the Pod level, then mount them into
   containers inside that Pod
   - name: config
     configMap:
       # Provide the name of the ConfigMap you want to mount.
       name: db-config
       # An array of keys from the ConfigMap to create as files
       items:
       - key: "db_connection_string"
         path: "db_connection_string"
       - key: "DB_NAME"
         path: "db_name"
   - name: secrets
     secret:
       secretName: db-password
```

Run it using the following command:

```
kubectl apply -f chap-8-db-user-password-pod.yaml
```

The pod loads the secrets in two ways. One as a file, using under volumes:

```
- name: secrets
  secret:
  secretName: db-password
```

As an environmental variable, using:

```
- name: DB_PASSWORD
   valueFrom:
   secretKeyRef:
     name: db-password
     key: password
```

The following is my session inside the Pod to verify that it got loaded:

```
kubectl exec -it configmap-secret-demo-pod -- sh
```

```
/ # ls
bin    config dev    etc    home   lib    media  mnt    opt
proc   root   run    sbin   srv    sys    tmp    usr    var
/ # ls /etc/secrets
password
/ # cat /etc/secrets/password
My$u&4du5er$ecr%8/ #
/ # env
…
DB_PASSWORD=My$u&4du5er$ecr%8
…
/ # exit
```

Securing access to secrets within the Pod is critical when productionalizing deployments.

It is an open secret that Kubernetes secrets are not secrets at all. Anyone who has access to the cluster and the correct permissions can read them by using the following command:

```
kubectl get secret db-password -o yaml
```

One of Azure’s Key Vault, AWS Secrets Manager, or Hashicorp Vault should be used for true secret management.

Gitops

We revisit the question, *How do you perform zero-downtime upgrades?* With storage/data being actively used by an application, it becomes extremely difficult. To start with, applications should be able to run the older version and the new version simultaneously.

Or an approach like Salesforce needs to be used where they copy the production DB to a newer version, have traffic go to the old version in read-only mode for some time, and make the fresh DB catch up with the changes and switch to the new DB.

Even with such support, things can go wrong that require downtime.

Whether you have a zero-downtime upgrade or not, automating the upgrade process is the first step, and Kubernetes greatly helps with the primitives we have seen. We, for learning purposes, did our upgrades manually by running the

upgrade commands. It is a recipe for disaster in production and could also be a savior in some cases where automation goes wild.

Automation makes bad things faster and on a larger scale! That is not an excuse for not investing in automation, but a note that it is important to be aware of it and have safeguards that will blow a *fuse* when bad things happen.

The way to automate upgrades is to use a concept called Gitops. No person can make changes or run commands in a production environment unless it is an emergency break-the-glass situation. All changes are initiated via git commits. A CI/CD pipeline is triggered when there is a commit, and a script converts the change into desired configuration changes and deploys them.

Kubernetes automatically detects the differences and applies the changes. The pipeline might have additional imperative commands (ideally, those also should be declarative). Examples include taking a backup of the DB, sending notifications, and so on:

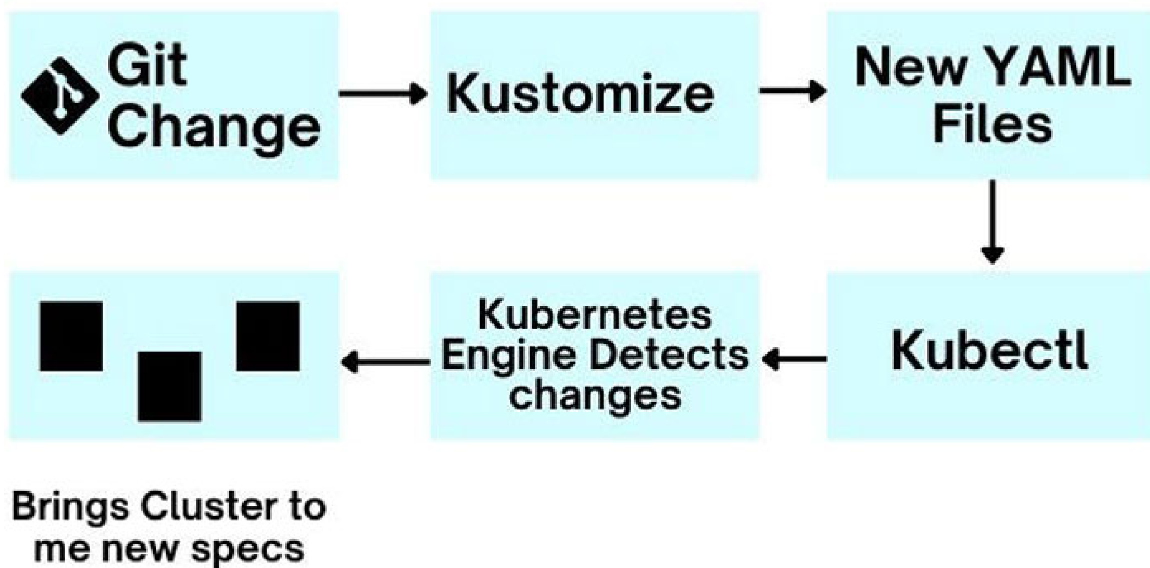


Figure 8.9: *GitOps propagating a change in git repository*

Try it

We will use kustomize (https://kustomize.io/) as the tool for driving our GitOps example. As of `kubectl` version 1.14, it is included part of `kubectl` distribution,

so no separate installation of kustomize is required. We will implement the simplest of GitOps operations to increase the number of replicas in a deployment.

kustomize uses a file named **kustomization.yaml** by default. We populate that file with the configurations and secrets we want the Pod to use. Once we have the base deployment, kustomize can customize it further by having overlays for different environments. We will define the **ConfigMap** and **Secret** and the deployment definition used by kustomize:

```
#!/bin/bash
# Create a kustomization.yaml file
# This file is used to define the resources to be customized
# we will use the pod definition we used to demonstrate ConfigMap
and Secret
# as the pod that must be created.
cat <<EOF >./kustomization.yaml
resources:
- chap-8-gitops-deploy.yaml
configMapGenerator:
- name: db-config
  literals:
  - db_connection_string="postgresql+psycopg2://app_user:
  {password}@dbhost/mydb"
  - db_user="app_user"
  - db_host="dbhost"
  - DB_NAME="mydb"
secretGenerator:
- name: db-password
  literals:
  - username=admin
  - password="My$u&4du5er$ecr%8"
EOF
```

Run the preceding script as follows:

```
chmod u+x ./kustomization.sh && ./kustomization.sh
```

This will generate a file named **kustomization.yaml**. The file will generate the same configuration settings and secrets as **chap-8-db-config-map.yaml**, **chap-8-db-password-secret.yaml**:

```
configMapGenerator:
- name: db-config
  literals:
```

```
  - db_connection_string="postgresql+psycopg2://app_user:
  {password}@dbhost/mydb"
…
secretGenerator:
- name: db-password
  literals:
  - password="My$u&4du5er$ecr%8"
```

It references **`chap-8-gitops-deploy.yaml`** as its base deployment definition:

```
resources:
- chap-8-gitops-deploy.yaml
```

Let's deploy it.

```
kubectl kustomize ./ | kubectl apply -f -
```

kustomize expands **`kustomization.yaml`** and creates Kubernetes manifests. You can save them to a file to inspect them. I chose to apply them directly. **`kubectl`** also offers a nice way to check the status of all the resources in the **`kustomization.yaml`** file:

```
kubectl get -k ./
NAME                       DATA   AGE
configmap/db-config-692g4bbb5f   4    74m
NAME                      TYPE    DATA   AGE
secret/db-password-5cc2btdmmd   Opaque   2    68m
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/gitops-deployment   1/1    1       1       46m
```

You can also use `kubectl describe -k ./` to get more details on all the resources.

Let us increase the number of replicas the GitOps way. The GitOps way is to make a change in a file and commit it. The change in the repo is detected, and a command is run. We create a patch specification called **`set_replicas.yaml`**. In that, we will set the number of replicas to 3:

```
#!/bin/bash
# This will create a kustomization.yaml file
# The increase in replicas will be applied to the deployment as a
patch.
# Create a patch increase_replicas.yaml
cat <<EOF > set_replicas.yaml
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: gitops-deployment
spec:
replicas: 3
EOF
cat <<EOF >./kustomization.yaml
resources:
- chap-8-gitops-deploy.yaml
patchesStrategicMerge:
- set_replicas.yaml
EOF
```

The session in which I change it back to 1 by changing the number of replicas in **set_replicas.yaml** is shown as follows:

```
gshiva@k8s-for-job-seekers$ ./kustomization-set-replicas.sh
gshiva@k8s-for-job-seekers$ kubectl kustomize ./ | kubectl apply -f
-
deployment.apps/gitops-deployment configured
gshiva@k8s-for-job-seekers$ kubectl get -k ./
NAME         READY  UP-TO-DATE  AVAILABLE  AGE
gitops-deployment  3/3   3       3       62m
gshiva@k8s-for-job-seekers$ vi set_replicas.yaml
gshiva@k8s-for-job-seekers$ kubectl kustomize ./ | kubectl apply -f
-
deployment.apps/gitops-deployment configured
gshiva@k8s-for-job-seekers$ kubectl get -k ./
NAME         READY  UP-TO-DATE  AVAILABLE  AGE
gitops-deployment  1/1   1       1       64m
```

In CI/CD engines like Jenkins/ArgoCD, which watch for changes in a **repo**, the setup would be something as follows:

```
watch for changes in the kustomization repo.
if set_replicas.yaml is changed:
  trigger: cd <set_replicas.yaml folder> && ./kustomization-set-
  replicas.sh && kubectl kustomize ./ | kubectl apply -f -
```

We have tried out how changes to Kubernetes deployments are done using the GitOps way.

Conclusion

The networking setup between applications is complex and setting them up securely is even more challenging. Kubernetes, with its pluggable architecture for networks, allows network plugins to be installed. The network plugins can directly take advantage of the network hardware, making it more performant than the default plugin. Network policy support from the network plugins makes securing network traffic use the same declarative approach independent of the underlying network provider.

Matching storage performance to the application needs enables a good customer experience while optimizing costs. Block devices such as hard drives, SSD, and SD cards are Directly Attached Storage. They give the best performance for random access because they are dedicated to the server and are directly attached. **Network Attached Storage** (**NAS**) uses protocols like CIFS/Samba (Windows) and NFS (Linux/Unix) to attach storage to servers using Ethernet. NAS allows storage to be attached to multiple servers simultaneously, making sharing files easy. They are also suitable for seamless storage expansion and large files and require sequential access. Millions of small files nested under the deep hierarchy of directories are their kryptonite. Object storage like S3 provides more cloud-native storage using HTTP/HTTPS protocol. They offer almost infinite scaling capacity and easier business continuity and disaster recovery benefits. They are very well suited for files that do not change much, like photos and videos. Persistent Volume Claims offer a way to abstract the underlying storage provider to the pods that mount it.

Node management features of Kubernetes allow more control over pod placement. It is helpful to specify what types of nodes a particular deployment must use (for example, GPU workloads must be run on nodes with GPU installed).

ConfigMaps and Secrets separate configuration from application code, allowing Pods to use them as file mounts and ENV variables. While Kubernetes Secrets are better than storing them as plain text, they could be more secure. Dedicated secret management solutions such as Azure KeyVault, AWS Secrets Manager, or Hashicorp Vault must be used in production.

GitOps allows the management of Kubernetes deployment as code. They follow the same workflow as a developer: change a file, commit, and CI/CD engine generates the artifact to be deployed. Kustomize is a tool that makes it easy to integrate Git with Kubernetes.

Points to remember

- Network support makes an application have a higher impact.

- Network plugins allow tighter integration with the network hardware in the data center.
- **Container Network Interface** (**CNI**) is an interface specification that the network plugins must adhere to provide network configuration capabilities.
- Namespaces are used to group applications in Kubernetes.
- Namespace-specific RBAC rules can be applied to control access to the namespace.
- Network Policy can be used to control network traffic between pods.
- Storage devices can be classified as Block, Network Attached Storage (NAS), and Object Storage.
- Block, NAS, and Object Storage have advantages and disadvantages that must be considered to match service needs for optimal performance and cost.
- ConfigMaps and Secrets are used to configure applications during deployment.
- GitOps uses code to manage applications and their deployments.

Interview questions and answers

1. **By default, can a Pod in one namespace talk to another Pod in a different namespace?**

   Yes. Kubernetes does not enforce any restriction within its virtual networking layer.

2. **How can you control network access to a Pod?**

   You can control it by using Kubernetes Network Policy.

3. **What issues can one run into when using persistent storage in Kubernetes?**

   Introducing state into any application causes issues. But then, all useful applications store data. Block storage can be attached to only one node at a time; if the node fails, it might be difficult to detach and attach it to another node cleanly. If the storage is on Kubernetes nodes themselves, what happens when that node fails must be considered. Even Network Attached Storage (NAS), which can be attached to multiple nodes at once, can have issues with locking, creating many files under a directory.

4. **How can you avoid persistent storage issues in Kubernetes?**

It can be avoided by storing data in managed storage services provided by a cloud vendor. Databases and file services could manage it. Pushing storage to cloud-native storage like S3 should also be considered.

5. **What are the different ways that ConfigMaps and Secrets be applied in a Pod?**

   ConfigMaps/Secrets can be expanded on a Pod in the following two ways:

   - ENV variables
   - Mounted as files

6. **How secure are Kubernetes secrets?**

   Not that much. Anyone who has access to the namespace can decrypt the secret. Use of *real* secrets management solutions such as HashiCorp Vault, Azure KeyVault, AWS KMS, or GCP KMS is recommended for production use.

7. **What is GitOps? What are its benefits?**

   GitOps is a methodology that uses a code repository (for example, Git) for driving application and infrastructure deployments. Since all operations are done through code, operations get the same benefits as following **Software Development Life Cycle** (**SDLC**), namely, version control, code reviews, approvals, and no manual changes.

CHAPTER 9

Section Summary and Interview Questions and Answers

Introduction

DevOps/**SRE (Site Reliability Engineering**) is one of the hottest areas in the **IT (Information Technology)** job market. With an average salary of 38% higher than the System Administrator's salary, it is a very lucrative field to be in. By joining the inclusive community of Kubernetes, you can have a purposeful job experience along with career growth opportunities.

In this chapter, we will summarize all the chapters we have seen so far and have all the interview questions and answers consolidated for easy reference.

Structure

In this chapter, we will review the following topics:

- Kubernetes/DevOps career map.
- Kubernetes adoption.
- DevOps/SRE culture.
- OS Fundamentals.
- Kubernetes basics and deployments.
- Kubernetes features in networking/storage/application/node management.
- GitOps.
- Interview questions and answers

Objectives

You will be ready for the interview by brushing up on this chapter. The answers to all the interview questions we have seen so far will be at the top of your mind.

Section summary

DevOps/SRE jobs are in high demand in the industry. With an average starting salary of $100K–$181K, which is 38% higher than the systems administrator salary, it is a very lucrative field to be in. By joining the inclusive community of Kubernetes, you can have a purposeful job experience along with career growth opportunities.

Namanh Kapur has an excellent video on *How I Would Learn to Code (If I Could Start Over)*–see https://youtu.be/k9WqpQp8VSU. He has articulated very well the importance of mindset and practicing using real environments for success in this area.

When you are starting, you are applying for a Kubernetes administrator or an operations engineer position. It is good to think about where you want to be in five years and know that a career in Kubernetes has a path to achieve that position.

You can choose between **the Individual Contributor** (**IC**) path or the management path.

Independent of your choice, be prepared for rough spots throughout your career, and having a growth mindset is essential for achieving your goal. See the following figure:

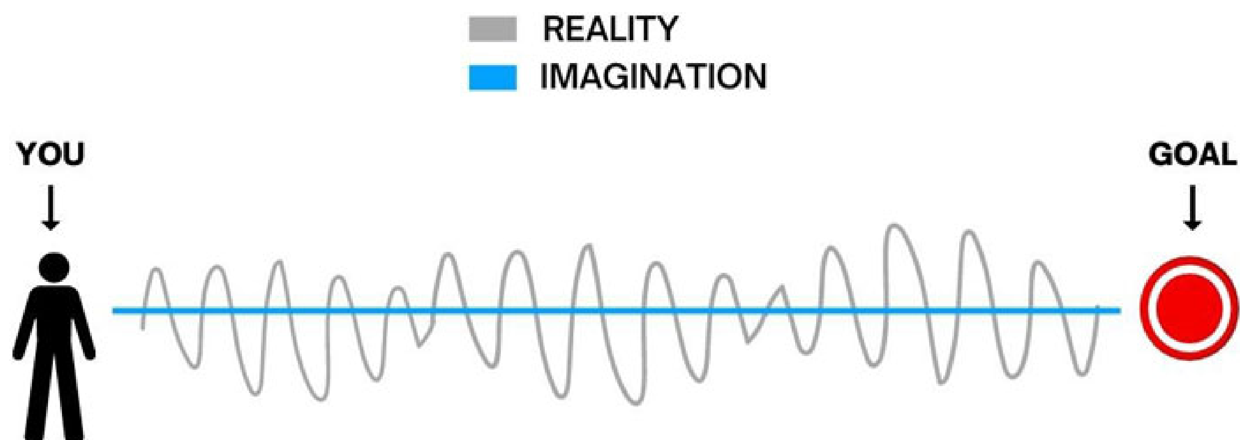
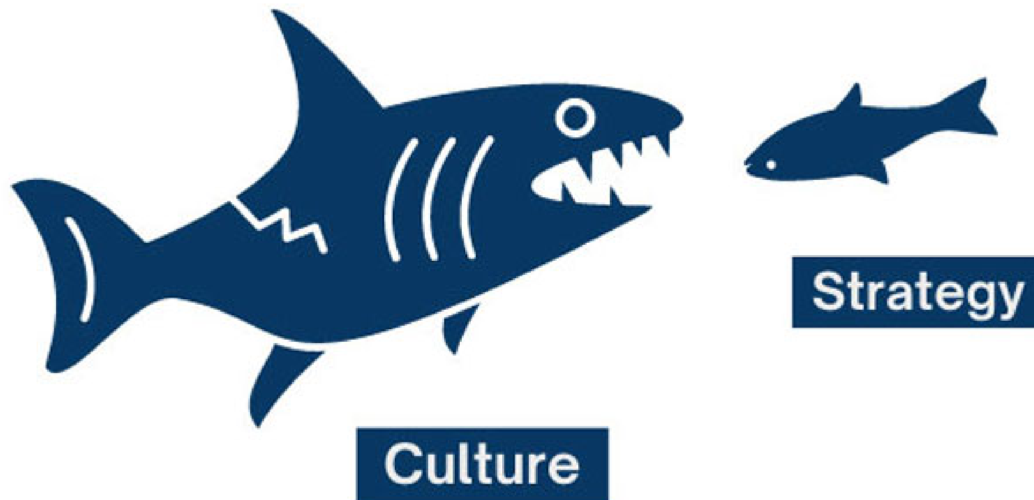


Figure 9.1: *Imagination versus reality*

Lack of inclusivity has been recognized as an industry-wide problem, especially in DevOps/SRE. Kubernetes community and leaders are taking active steps to make it inclusive. Anyone of any gender, sexual orientation, religion, caste, age, national origin, or color is welcome in this community.

You will find people who look like you in the inclusive Kubernetes community.

Kubernetes adoption is held back only by the lack of personnel who can make sense of the complex Kubernetes ecosystem and how to integrate their existing investments with Kubernetes. See the following figure:



Culture Eats Strategy

Figure 9.2: *"Culture eats strategy for breakfast"—Peter Drucker.*

For an employee, the preceding quote can be paraphrased as *Mindset eats skills and plans for breakfast*. Refer to the following figure:

Figure 9.3: *Mindset>Skills>Plans*

The importance of mindset has been mentioned again and again, as reinforced in *Namanh Kapur's* excellent video on *How I Would Learn to Code (If I Could Start Over)*—see **https://youtu.be/k9WqpQp8VSU**.

The key elements of DevOps/SRE culture are learning, continuous improvement, collaboration, intelligent risks, and relentless focus on customers (including internal customers) have been shown in the following figure:

Figure 9.4: *DevOps culture values*

Learning to enjoy the journey is more important than being focused on the destination. Although you will learn the skills in the upcoming chapters,

practicing learning with curiosity on how Kubernetes works, continuously improving Kubernetes skills, and teaching others through open-source tutorials, LinkedIn posts, etc. will provide ample demonstratable evidence for you. Enjoying the time when you overcome the struggle of learning a new concept is what will keep you going and getting the job.

The preceding reason is why we spent three chapters on culture and mindset. Without alignment on the culture and mindset, it is very difficult to sustain in this field without getting burnt out.

Operating system fundamentals

Having a good grasp of Operating System(OS) concepts will make working with Kubernetes easier. Look at the following figure given for reference:

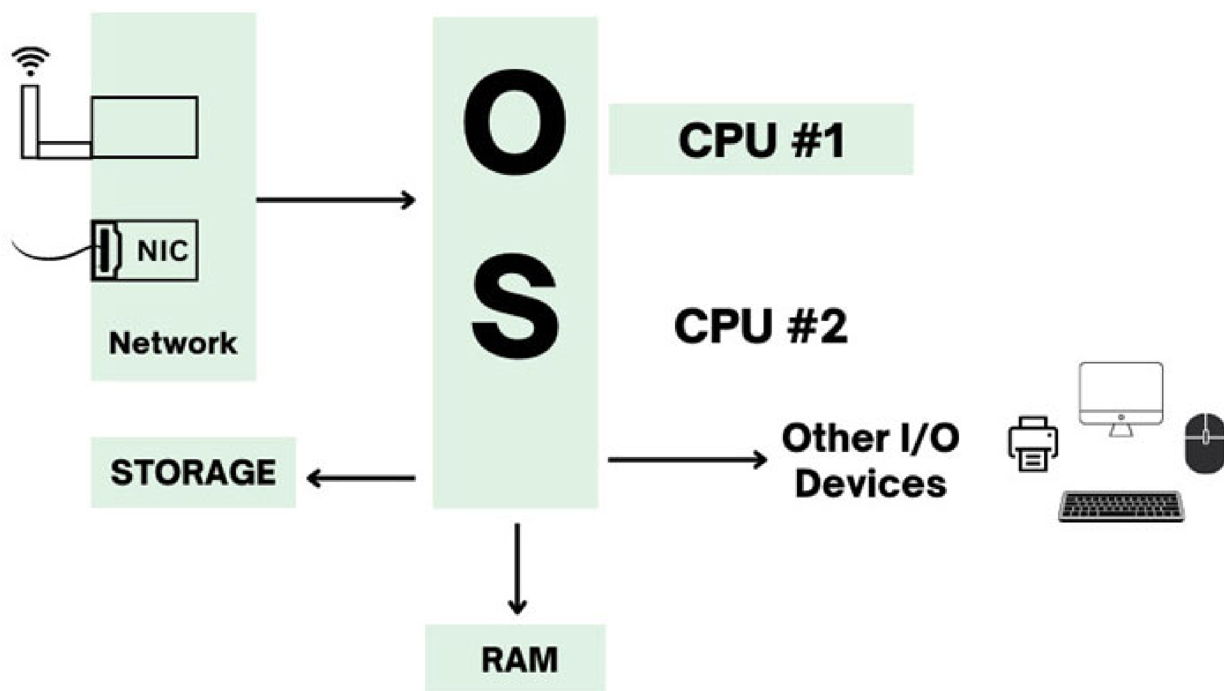


Figure 9.5: *OS: complete picture*

The computer is made of the following four key components:

1. Processor (CPU), which does the actual computation.
2. Memory (RAM) is used to store temporary data.

3. Storage (HDD/SDD) is used for storing data permanently.
4. Network (NIC), which is used for communicating with other computers.

The operating system helps coordinate the four key components so that computations are done as efficiently as possible. It queues the work for the processor, fetches data from RAM and HDD/SDD, and processes the network packets, among other things.

Understanding how the network works is key to troubleshooting Web applications.

Containers/docker

Containers are processes that have been isolated by the OS through the use of Cgroups and namespaces (see the following figure):

Figure 9.6: *Running containers == running processes*

The following lists the salient points of containers:

- Running containers is just a process.
- Processes require server resources such as CPU/RAM/storage and network.
- Containers require the same resources as processes such as CPU/RAM/storage and network.
- Containers start with the same speed as a process, with very little overhead.
- Containers are the latest iteration of the virtualization of servers.
- Containers can be seen as a lightweight virtualization mechanism.
- Containers share the server's entire resources as opposed to the dedicated RAM/CPU/storage of a traditional VM.
- Container registries like Docker Hub make sharing images super easy.
- Dockerfile enables the rebuilding of container images anywhere.
- Container networking enables multiple containers to listen on a standard port and takes care of routing traffic to the container by exposing host network ports.
- Docker image has a root file system mounted at runtime to the container. This enables the Centos container to run on an Ubuntu host.
- Build a container using instructions specified in Dockerfile.
- Using multi-stage builds, you can keep the container image as small as possible.
- You can run, stop, and delete containers.
- You can push multiple versions of your container to DockerHub/Registry for sharing.

Kubernetes basics and primitives

Kubernetes was born out of the need for orchestrating container deployments. The following list gives you a quick overview of Kubernetes and its primitives:

- A solid understanding of the fundamentals of processes and their resource usage (CPU/RAM/storage/network) makes working with the complex ecosystem of Kubernetes a breeze.

- Kubernetes emerged as the winner of the container orchestration battle between Apache Mesos and Docker Swarm.
- Kubernetes does container orchestration across multiple hosts in an automated fashion of what you could do manually.
- Free labs such as **https://labs.play-with-k8s.com/** make Kubernetes accessible to anyone with a browser.
- Kubernetes uses declarative specifications to launch applications and services.
- Declarative code specifies the end state and leaves the implementation to the underlying provider. Imperative code directs the provider on what should be done and in what order.
- The primitives of Kubernetes are Pods, ReplicaSets, Deployments, DaemonSets, Services, and IngressControllers.
- For persistent storage, Kubernetes uses Physical Volumes (PV) and persistent volume claims (PVC).

Deployments using Kubernetes

Managing the deployment of applications while ensuring availability is a key functionality of Kubernetes. The following figure shows blue-green deployment:

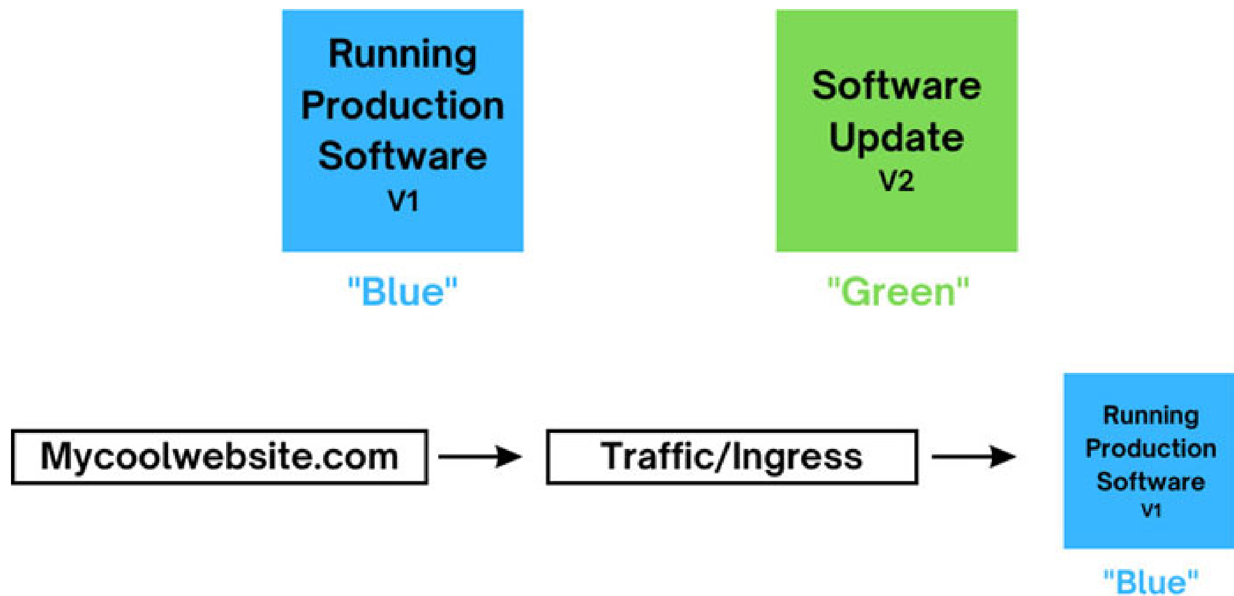


Figure 9.7: *Blue-green deployments*

The points to remember about Kubernetes deployments are as follows:

- Zero downtime upgrades for non-trivial applications are hard.
- Picking the right amount of availability goals for service is critical for productivity.
- To achieve the availability goals, the enterprise culture must be oriented toward making resilience a priority.
- Kubernetes helps achieve the enterprise's availability goals by providing support for different deployment strategies, such as blue/green deployment, rolling upgrades, and canary deployment. Built-in support for health checks, Pod migration, and network routing ensure that the application teams focus on application-level support for resiliency and observability.

Kubernetes services/features

Kubernetes provides many extensible features and concepts to perform container orchestration at scale. The following lists its support for advanced networking, isolation, configuration, and storage:

- Network support makes an application have a higher impact.
- Network plugins allow tighter integration with the network hardware in the data center.
- CNI (Container Networking Interface) is an interface specification that the network plugins must adhere to provide network configuration capabilities.
- Namespaces are used to group applications in Kubernetes.
- Namespace-specific RBAC rules can be applied to control access to the namespace.
- Network policy can be used to control network traffic between pods.
- Storage devices can be classified as Block, NAS (Network Attached Storage), and Object Storage.
- Block, NAS, and Object Storage have advantages and disadvantages that must be considered to match service needs for optimal performance and cost.
- ConfigMaps and Secrets are used to configure applications during deployment.

GitOps

GitOps uses code to manage applications and their deployments. Refer to the following figure:

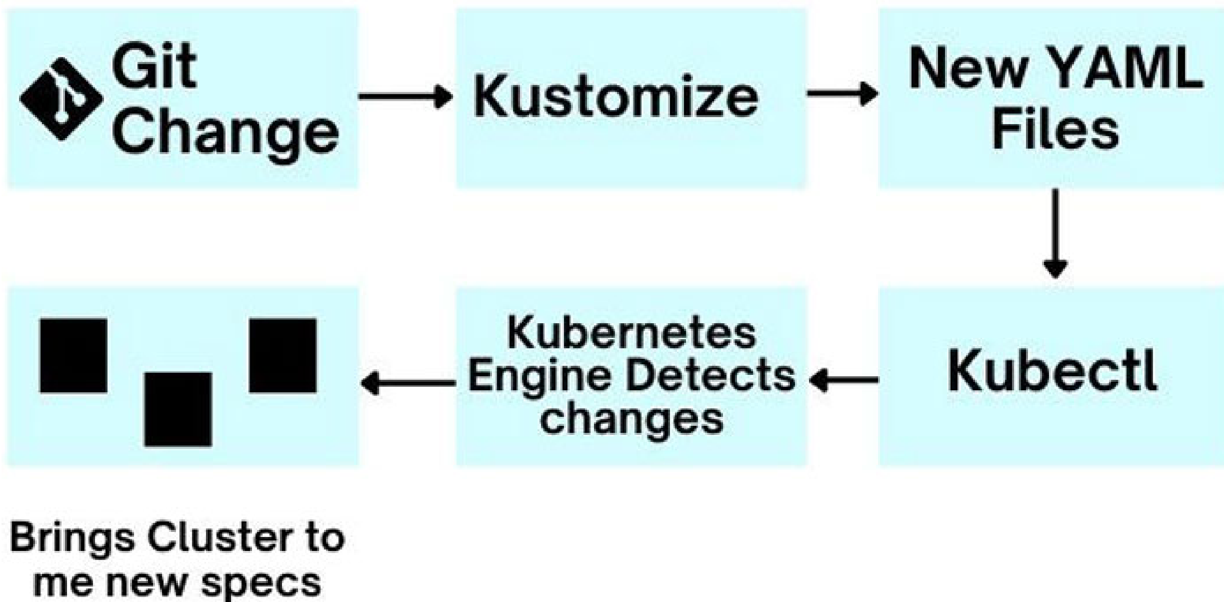


Figure 9.8: *GitOps propagating a change in the Git repository*

Interview questions and answers

The following is a consolidated list of all the questions and answers in the previous chapters.

Chapter 1: Kubernetes/DevOps career map

1. **Why do you want to become a Kubernetes engineer?**

   I like fixing issues and enjoy being part of the team that helps developers deliver features to our customers faster, safer, and better. Kubernetes enables that for developers by taking care of security, operability, and observability.

2. **Can you give me examples of where you fixed issues that made the situation better?**

   If you have experience in the IT field:

   Give an example where you helped find an issue with a software release or production issue. Interviewers also like to hear where you had a non-technical solution to a problem. One example of mine is where to quickly solve customer issues; I suggest having a dedicated chat room where assigned developers (we called them Superheroes) would be watching, and anyone in the field can ask them questions.

   If you do not have experience in the IT field:

   Interviewers are looking for people with the right attitude. Have an example ready where it was a tough issue, preferably close to a technical area but not necessary. An example could be fixing the radio or a clock at home. Or helping a family member solve a tough issue, like helping your mother talk to a person far away via video chat when your mother was very worried about meeting that person and could not meet (very relatable during the COVID lockdown). More authentic is better.

3. **Where do you see yourself in five years?**

   Depending on the career path (manager or IC role), imagine yourself coming to the office five years from now and describe the tasks that you would be doing ideally. For an Architect, it would be making proposals for architectural changes that would build on the success you were instrumental in achieving and incorporate the latest advancements in the field. You would also be presenting this proposal to other teams to get their buy-in.

Chapter 2: Kubernetes adoption in the industry

1. **What is the de facto standard for container orchestration? Why?**

   Kubernetes is the de facto standard for container orchestration. While good solutions for container orchestration like Nomad and Mesos exist, the market has overwhelmingly voted with its feet and dollars toward Kubernetes.

2. **Is it possible to have 10 years of hands-on experience in Kubernetes?**

   Kubernetes was released to the public in 2014 and has gained stability and adoption only in the past five to six years. So, 10 years of hands-on experience in Kubernetes is not possible.

3. **What are the primary challenges that enterprises face when implementing Kubernetes?**

   The main challenges that enterprises face are as follows:

   - How to implement a hybrid cloud? Since it is not a good idea to migrate completely on-premises applications to the public cloud.
   - How to abstract Kubernetes complexity from developers by leveraging the multiple tools/methodologies available from the Kubernetes ecosystem? (Aka *Day 2 problems*).

4. **They say that Kubernetes is the answer to all my deployment issues. What is your opinion on that?**

   While Kubernetes is a critical tool in a company's digital transformation, it is not a panacea. Even with hosted Kubernetes solutions, it requires dedicated teams for day-to-day operations and customizing tools for improved developer experience and site reliability.

Chapter 3: Introduction to DevOps/SRE culture

1. **Why is the culture of a company important?**

   A company's culture is an excellent predictor of the people it attracts and how excited you will be to show up on Monday morning. Having culture fit determines how engaged the employees would be and, in turn, the success of the company.

2. **What is DevOps?**

   DevOps is a movement that brings developers and operations closer together to deliver value to the customer.

3. **I am hiring an engineer for my DevOps team. Is having a DevOps team sufficient to implement DevOps?**

   DevOps team focusing on deployment and site reliability issues to develop expertise and customize solutions to the enterprise/developer needs is essential but not sufficient. Support for the shared responsibility and development of resilience both in personnel and software, is required from the top and the bottom.

4. **What are the key elements of DevOps/SRE culture?**

   The key elements are focus on learning, continuous improvement, collaboration, taking intelligent risks, and, most importantly, relentless focus on customers.

5. **A good interviewer would give more importance to your mindset or skills.**

   While demonstrated skillset would trump other factors in general, given that two candidates have a similar skillset, the interviewer would give more importance to the interviewee's mindset. The person with the growth mindset would be chosen.

6. **Do you need to be in a company to practice DevOps principles?**

   No. DevOps principles of having curiosity, continuously learning, and making improvements can be done when doing daily chores. You can also practice honoring commitments and solving problems without blaming interactions with your family.

7. **Can you give examples of when you practiced DevOps principles?**

Be prepared with examples where you demonstrated curiosity and improvements to reducing toil. While professional examples are ideal, you can use personal experiences also and, in fact, might be remembered more than professional ones.

Chapter 4: Operating system fundamentals

1. **My phone has only 16 GB. What must I be complaining about?**

   The complaint is about limited storage on the phone. The phone runs out of storage very quickly when taking photos or videos. This amount has nothing to do with the phone's RAM. RAM is also measured in GB. This can be confusing. An engineer should have a very clear understanding of the components of a computer.

2. **What are the main components of a computer?**

   The main components are the CPU, RAM, Network, and Storage. GPUs are becoming an important component in servers due to the high demand for video processing and machine learning.

3. **Why do we need an operating system?**

   An operating system is needed to provide an abstraction layer to the CPU, RAM, Storage, Network, and other peripherals. A multi-tasking operating system like Linux/Windows ensures optimal resource utilization so that a single application waiting for input does not block other applications from running. Advanced features of an operating system provide better security and isolation between applications such as cgroups and namespaces, which are used for containerization.

4. **Describe what happens when I type http://www.google.com on my browser.**

   Ask the interviewer what depth they want the answer on. Start with the basics:

   The browser checks the type of URL (HTTPS versus HTTP versus file, and so on) and launches a request accordingly. Here is the request for the HTTP protocol. The browser asks the operating system to find the IP address for the domain name **www.google.com**.

   www.google.com is resolved into an IP address using DNS servers. (DNS prefers UDP but will use TCP if required.) DNS uses the well-known Port `53`. Your computer establishes a TCP/IP connection between your computer and the IP address returned from the previous step.

Using the TCP/IP connection, it sends an HTTP GET request to the IP address of **www.google.com**. The server listening on Port `80` returns an HTML page with a search box. The browser renders the HTML page obtained in the previous step.

The details required by the interviewer can be arbitrary and most likely involve the interviewer's area of expertise. Sure, unlikely, but possible that a hardware engineer would like to know whether you know about key bounce handling that needs to occur to make sure when you press *w* on your keyboard, only one *w* and not *wwwww* shows up on the browser. A network engineer might be interested to know about the NAT protocol being used between your router and Google's servers. A backend server person might be interested to know about how geo-caching. The *simple* task of viewing the Google landing page results from years of hardware, software, and design improvements, and one answer will not satisfy all interviewers.

Chapter 5: Containers/docker

1. **What are the differences between containers and virtual machines?**

   The underlying technology between virtual machines and containers is completely different from each other. Virtual machines provide complete isolation of resources such as CPU, RAM, and storage. Due to such isolation, it is possible to run Windows, Solaris, and Linux VMs. As they are images of an entire server, they are large, making it very difficult to share them with others. Also, it is impossible to share resources across VMs running on the same server due to isolation.

   Containers, on the other hand, are isolated processes launched by the operating system. They share the kernel, CPU, and RAM. While different distributions of Linux OS can be launched on the same server, it is impossible to launch a Windows container on a Linux Host due to the sharing of the kernel. Containers can be shared easily due to their relatively small size. The code to build a container is even smaller and thus can be shared using code repositories such as GitHub.

2. **How come containers are so fast?**

   Containers are regular processes in an operating system and thus have very little overhead to launch them.

3. **What can slow down the launching of containers?**

   The launching of containers can be slow due to startup code that needs to run within the container and if it requires information from external systems to start.

4. **How do you share a container image?**

   Container images are shared by pushing them to a registry.

5. **What makes the building of containers reproducible?**

   Container images are built using Dockerfile specifications. This makes them reproducible.

6. **How do you run a container and expose a port?**

   You use `docker run -d -p 8080:80/tcp nginx`. This runs a Nginx container that listens on Port `80` and is exposed via host network interfaces at Port `8080`.

7. **Can you launch multiple containers listening to port 80 on the same host? How and when does it work?**

   Yes. Docker virtual network layer allocates unique IP to each container. From a container perspective, it is a complete server that does not share any resource with any other container. Multiple containers can listen to the same internal port without any conflicts. They cannot be exposed to the same host port.

8. **How are labels used in docker images? How are they different from tags?**

   Labels are key-value pairs used to attach metadata to container images. These are very useful for CI/CD, where decisions can be made using the data in the labels. Since labels can be specified during build time, they cannot be changed after the image is built. This makes it more reliable than tags which can be changed at any time.

Chapter 6: Kubernetes—basics

1. **Is Kubernetes simple?**

   No. Kubernetes is an extremely complex ecosystem with tools built on top of tools to handle the hard problem of deploying applications at scale.

1. **If Kubernetes is so complex, how does one go about learning it?**

   A solid grasp of Kubernetes, the fundamentals of processes, storage, and networking makes understanding Kubernetes a matter of learning the syntax. Learning the syntax and practicing with Kubernetes is how one goes about mastering Kubernetes.

2. **What are the primary primitives of Kubernetes?**

   The primitives of Kubernetes are Pods, ReplicaSets, Deployments, DaemonSets, Services, and IngressControllers.

3. **What are Pods?**

   Pods are the smallest unit of deployment in Kubernetes.

4. **What are ReplicaSets?**

   Replica Sets are a collection of Pods where the number of pod replicas is specified.

5. **What are Deployments?**

   Deployments are used for deploying applications and upgrading them. The upgrade strategy can be specified when replacing an existing deployment with a newer application version.

6. **What are DaemonSets?**

   DaemonSets are used when you want only one instance of an application running on every Kubernetes node.

7. **What are Services?**

   Service acts as proxy routing network traffic between healthy pods of a deployed application.

8. **What are IngressControllers?**

   IngressControllers generally leverage underlying infra providers to expose a Service to the public by using a public internet-facing load

balancer. It can also expose a Service via network ports on a node using HostPort. Layer 7 routing (HTTP/HTTPS) support provided by Ingress Controllers via paths enables a single ingress controller for multiple Services.

9. **How does Kubernetes handle persistent storage?**

   Kubernetes abstracts the underlying storage by using physical volumes. Persistent volume claims reserve storage within those physical volumes for Pods.

Chapter 7: Kubernetes deployments

1. **What are the contributing factors for application downtime?**

   A culture that rewards feature delivery over code reduces rework, poor coding practices, poor configuration management, lack of CI/CD, Backup and DR not tested regularly, and security as an afterthought.

2. **What different types of deployments do you know for safe upgrades?**

   Blue/Green deployment, where upgrades are done using a copy of the production environment. If everything works well, production traffic is shifted to the upgraded version. Production traffic is shifted to the previously working version if serious errors occur.

   Canary deployment is where a small portion of the traffic is directed toward the upgraded instances, and if there are no errors, old instances are gradually replaced with new ones.

Chapter 8: Kubernetes Services (Networking—Web Services, Storage—Persistent Volumes, Application Management, Node Management, and GitOps)

1. By default, can a Pod in one namespace talk to another Pod in a different namespace?

   Yes. Kubernetes does not enforce any restriction within its virtual networking layer.

2. **How can you control network access to a Pod?**

   You can control it by using Kubernetes' Network Policy.

3. **What issues can one run into when using persistent storage in Kubernetes?**

   Introducing state into any application causes issues. But then, very few applications that are useful do not store data. Block storage can be attached to only one node at a time; if the node fails, it might be difficult to detach and attach it to another node cleanly. If the storage is on Kubernetes nodes themselves, what happens when that node fails must be considered. Even NAS, which can be attached to multiple nodes at once, can have issues with locking, creating many files under a directory.

4. **How can you avoid persistent storage issues in Kubernetes?**

   It can be avoided by storing data in managed storage services provided by a cloud vendor. It could be managed by databases and file services. Pushing storage to cloud-native storage like S3 should also be considered.

5. **What are the different ways that ConfigMaps and Secrets be applied in a Pod?**

   ConfigMaps/Secrets can be expanded on a Pod in the following two ways:

   1. ENV variables
   2. Mounted as files.

6. **How secure are Kubernetes secrets?**

Not that much. Anyone who has access to the namespace can decrypt the secret. The use of *real* secrets management solutions such as HashiCorp Vault, Azure KeyVault, AWS KMS, or GCP KMS is recommended for production use.

7. **What is GitOps? What are its benefits?**

   GitOps is a methodology that uses a code repository (for example, Git) for driving application and infrastructure deployments. Since all operations are done through code, operations get the same benefits as following SDLC (Software Development Life Cycle), namely, version control, code reviews, approvals, and no manual changes.

CHAPTER 10

Kubernetes on Various Platforms

Introduction

Kubernetes is commonly seen using some distribution of Linux OS and running on x64 (amd64, Intel) processors. Kubernetes can also run-on ARM processors (Raspberry PI) and Windows OS (with certain limitations).

It is strongly recommended to use Kubernetes run and maintained by cloud vendors for enterprise production use. Kubernetes is highly complex software that needs to be updated regularly (after thoroughly testing stable versions at scale). One can run into issues requiring deep knowledge of Kubernetes and OS kernel. It is challenging for an enterprise whose focus is not Kubernetes to invest resources in this unless they have regulatory concerns or a dedicated team. We will look into the top three public cloud providers and highlight the differences between their implementations of Kubernetes. You will learn about deploying Kubernetes on Azure, AWS, and GCP and how to monitor them.

Structure

In this chapter, we will discuss the following topics. Kubernetes Support for Windows Deployment and monitoring of Kubernetes using:

- AKS on Azure
- EKS on AWS
- GKE on GCP
- Raspberry PI (Free Bonus)

Objectives

After reading this chapter, you will be able to answer questions on the advantages and limitations of Windows support in Kubernetes. Questions on differences between Azure, AWS, and GCP Kubernetes implementations are also covered. You will have hands-on experience deploying and performing essential monitoring on AKS, EKS, and GKE.

Windows support

It is not that well known that Kubernetes has windows support. Why would you need Windows support when it runs so well as Linux?

Many enterprise applications are written for the Windows server OS, and it is not worth migrating them to Linux.

Microsoft❤ Linux: Things predicted to happen only when *pigs fly* is happening for Microsoft products. Of all the products, one would think MS SQL server would be the hardest to port to Linux. It is available with almost all features on a Linux-based docker image!

A windows worker node can join an existing Kubernetes cluster, which can be managed just like any other Kubernetes node. Windows servers cannot, however, act as the primary server. The primary server node has to run on Linux. A flannel networking plugin is required for establishing connectivity.

Limitations (highlights as of Apr 2022)

Please see **https://kubernetes.io/docs/setup/production-environment/windows/intro-windows-in-kubernetes/** for up-to-date and complete information.

- Windows Server 2019 is the only OS supported for the nodes.
- Privileged containers are not allowed.
- The windows containers should also be using Windows Server 2019 as the base OS (not running alpine on ubuntu like fun).
- CPU/memory limits must be monitored carefully, and enough reserve should be provided for both. Windows does not have an **out-of-memory** (**OOM**) process killer like Linux. The end effect is that over-provisioning memory can result in slowness due to paging.

- Storage—only the entire volume can be mounted on a windows container.
- Networking—the host networking node is not available. A single service can only support up to 64 backend pods.
- Outbound ping does not work. Use curl instead.
- The pod cannot have a mixture of Windows and Linux containers.

Azure—Azure Kubernetes Service (AKS)

Azure provides hosted Kubernetes as **Azure Kubernetes Service** (**AKS**). The service highlights are listed in this paragraph. Unlike the other cloud providers in this section, you get charged only for the worker nodes. You have no access to the primary control node. Azure performs all the upgrades required on the master and worker nodes. AKS with uptime SLA (Service Level Agreement) is available for purchase if you want a financially backed SLA. AKS should be deployed in multiple availability zones to handle region failures in Azure. Azure guarantees that the upgrades will be performed within the 24-hour gap between regions.

Azure Pod identity is a cool/concept where you can connect Azure Active Directory to a Pod. This way, the pod can access the resources allowed for that role. Super helpful when you only want pods to access resources that belong to the same environment. That way, a pod in a developer environment cannot access the production database even if it was configured with the proper credentials.

Setting up Kubernetes in HA mode is a challenge. All the public cloud providers give you that option. In production, that feature alone is worth considering cloud-hosted Kubernetes.

Try it

Let us try AKS (**https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-deploy-cluster?tabs=azure-cli**):

1. Launch AKS cluster.
2. Run a sample application.
3. Monitor it.

4. Kill a Node.
5. Clean up the cluster.

AWS—Elastic Kubernetes Service (EKS)

AWS **Elastic Kubernetes Service** (**EKS**) is Amazon AWS's version of managed Kubernetes. EKS's control plane also runs in HA mode across multiple availability zones to ensure that an outage in one data center does not bring down your entire cluster.

EKS's control plane consists of at least 2 API server instances + 3 etcd instances running in 3 availability zones. The scaling is done based on demand. To reiterate the non-trivial nature of the effort required to set HA correctly, note that this setup needs extensive testing before it is deployed. All scenarios, from a single API server to the entire data center going down, need to be tested for every Kubernetes upgrade. Continuous monitoring and ensuring that the failover does occur as expected also need continuous testing.

With the AWS Pod authenticator, it is possible to hook AWS **Identity and Access Management** (**IAM**) to the pods so they can connect to only the AWS resources they are authorized for. For example, if read-only access to the AWS DB instance were provided to a principal in AWS IAM, the write access by a Pod assigned to the principal would be denied access.

One difference between AKS and EKS, is that EKS service costs 10¢/hour per cluster, independent of whether your cluster is running any workloads or not.

Try it

Let us try EKS (**https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html**):

1. Launch EKS cluster.
2. Run a sample application.
3. Monitor it.
4. Kill a Node.

5. Clean up the cluster.

GCP—Google Kubernetes Engine (GKE)

It is no surprise that Google offers one of the best (if not the best) implementations of Kubernetes. Besides being the mother of Kubernetes, many of the critical Kubernetes developers are employees of Google. Kelsey Hightower (**https://en.wikipedia.org/wiki/Kelsey_Hightower**) is one of their most well-known developer advocates. His contribution to evangelizing Kubernetes has made Kubernetes what it is today.

One critical difference between GKE and EKS/AKS is speed. While in EKS/AKS, you could spend 10–20 minutes for the cluster to be fully functional; GKE is up and running in minutes. GKE brings advanced and latest features of Kubernetes earlier than other vendors. Like HA, seamless integration and testing of the advanced features is very hard. A good example is Istio integration. Istio (**https://istio.io/**) is the leading service mesh implementation. GKE has built-in support for Istio.

Autopilot mode is another unique feature of GKE. In this mode, you also leave the management of the worker nodes to GKE. OS upgrades and security patches are all managed for you. From a compliance point of view, it is a huge benefit.

Cloud-native security is another feature available only in GKE. Using GKE sandbox, pods/namespaces can be further isolated from other Pods running on the same cluster. This ability reduces the number of clusters that need to be deployed, especially for multi-tenanted applications. Sandbox feature provides good enough isolation while reducing the management costs of operating Kubernetes clusters.

Try it

Let us try GKE (**https://cloud.google.com/kubernetes-engine/docs/deploy-app-cluster**):

1. Launch GKE cluster.
2. Run a sample application.
3. Monitor it.
4. Kill a Node.

5. Clean up the cluster.

Raspberry PI cluster—for fun and profit

In an interview, my answer on whether an enterprise should run its own cluster or go for managed clusters is that they should go for the managed service.

Personal learning is a different story. Operating a cluster on your own with real servers gives an understanding of Kubernetes that you can never get from using a managed service. Although it is true that most of the scenarios can be tried on your cluster on the cloud (which I strongly recommend that you do), the learnings are at a different level when you pull out a network cable or unplug an external drive from your personal cluster.

Raspberry PI

Raspberry PI is a $35 computer that brought computing to millions of children (and adults) at an affordable cost. Docker/containers, after a long time of not supporting ARM architecture, started officially supporting ARM-based deployment and container images. These revolutions combined enabled robust, scalable cluster computing on ARM devices.

Balena (**https://www.balena.io/**) is an excellent example of managing fleets of cheap IoT devices at scale. It is free for up to 10 units. By switching the SD cards that boot up your Raspberry PI, you can cost-effectively try the Docker-based Balena and Kubernetes solutions.

Running Kubernetes on Raspberry PI

To get started, you can run both the primary and worker node on a single Raspberry PI (RPi). You can get the $35 version of RPi4 with 2 GB RAM, but the $55 version with 4 GB RAM is recommended. Running a 2 GB version will quickly show how swapping affects performance, especially on an SD card.

The easiest way to install Kubernetes is to use k3sup (pronounced ketchup). k3sup (**https://github.com/alexellis/k3sup**) is so good that it can set up a Kubernetes cluster across an RPi and x86 machines. If you have Linux running on your laptop, you can add that as a worker node! Install using a single command:

```
k3sup install --local
```

Try it

Let us try the Raspberry PI cluster (https://blog.alexellis.io/raspberry-pi-homelab-with-k3sup/):

1. Launch Raspberry PI cluster.
2. Run a sample application.
3. Monitor it.
4. Kill a Node.
5. Clean up the cluster.

Conclusion

Kubernetes can be run on many platforms, from tiny IoT devices like Raspberry PI to Windows to multi-core GPU monsters on the cloud. Though limited, Windows support provides a single pane of glass management for Windows applications. Managed Kubernetes solutions offered by cloud providers (Azure, AWS, and GCP) let enterprises focus on their core competency while allowing the hard work of managing Kubernetes to be done by the Kubernetes experts. All cloud providers give HA capability while offering each of their unique selling points. Using hands-on experiments, we launched our cluster on all the clouds, monitored them, and cleaned them up.

Running Kubernetes clusters on Raspberry PI (RPi) is highly recommended for your learning. We repeated the same experiment of launching a cluster, monitoring it, and cleaning it up on RPi. The investigation allows touching all the components that make up a cluster physically.

In the upcoming chapter, we will look at questions you might be asked about optimizing Kubernetes performance with hands-on exercises.

Points to remember

- Kubernetes is available not only in Linux but also on Windows with limitations.

- Managed Kubernetes is available on all cloud providers, including Azure (AKS), AWS (EKS), and GCP (GKE).
- Using the free trial offerings, you can have hands-on experience in launching and managing Kubernetes on AKS, EKS, and GKE.
- For a deeper hands-on experience, you can try a true multi-cluster using Raspberry PIs.

Interview questions and answers

1. **Why do we need Managed Kubernetes when we can launch our own easily?**

   Running Kubernetes in production is hard. Setting up and testing on a continuous basis HA requires a dedicated Kubernetes team. Upgrading and applying security patches requires expertise. Support for Kubernetes and kernel issues is extremely difficult to find. Unless it is a very large tech enterprise, it takes the focus of the teams away from delivering their unique value to customers.

2. **Does Kubernetes run on Windows?**

   Yes, but with many limitations. The key ones are that the windows server cannot be the primary server, you cannot run Linux containers on the Windows server, and privileged containers are not allowed.

3. **What are the differences between AKS, EKS, and GKE?**

   For most practical purposes, they are the same except for costing and support for advanced features. AKS does not charge for the primary servers, whereas EKS and GCP charge for them. All of them offer the ability to hook access control to their Identity Access Management (IAM) system. GKE offers built-in support for Istio service mesh and an autopilot mode where all the updates for the worker nodes are done by GKE.

CHAPTER 11

Kubernetes Performance Optimizations

Introduction

Kubernetes has an overhead for performing orchestration across nodes. Optimized distributions for the Kubernetes platform can significantly reduce critical performance bottlenecks/overhead.

How do you address performance issues in Kubernetes?

If you are asked this question in an interview, you are playing the end game, and it is important to finish strong. Even if you are not interviewing, the process listed in this chapter will make you a hero when faced with performance issues in your enterprise's cluster.

This chapter will teach you how to quickly identify performance bottlenecks and the top 10 fixes you can use for Kubernetes performance issues.

Structure

We will cover the following:

- Utilization Saturation and Errors (USE) methodology
- Top 10 Kubernetes performance optimizations

Objectives

After mastering this chapter, you will have a framework to analyze performance issues methodically. The methodology works for any computer performance diagnostics, not just Kubernetes. You will also know how to fix them using the top 10 performance optimizations for Kubernetes.

Computer performance

Computer performance can mean different things to different people depending on what their chief concerns are for them. The most crucial concern for a laptop or a mobile phone is how good the battery performance is.

For large data centers, the power consumption of the servers would be the chief concern. For games, it is GPU performance. It is essential to define what would be the performance concerns for you as a future Kubernetes engineer/administrator/developer.

The chief concern in the case of Kubernetes would be the primary computer resources, as illustrated in the following figure:

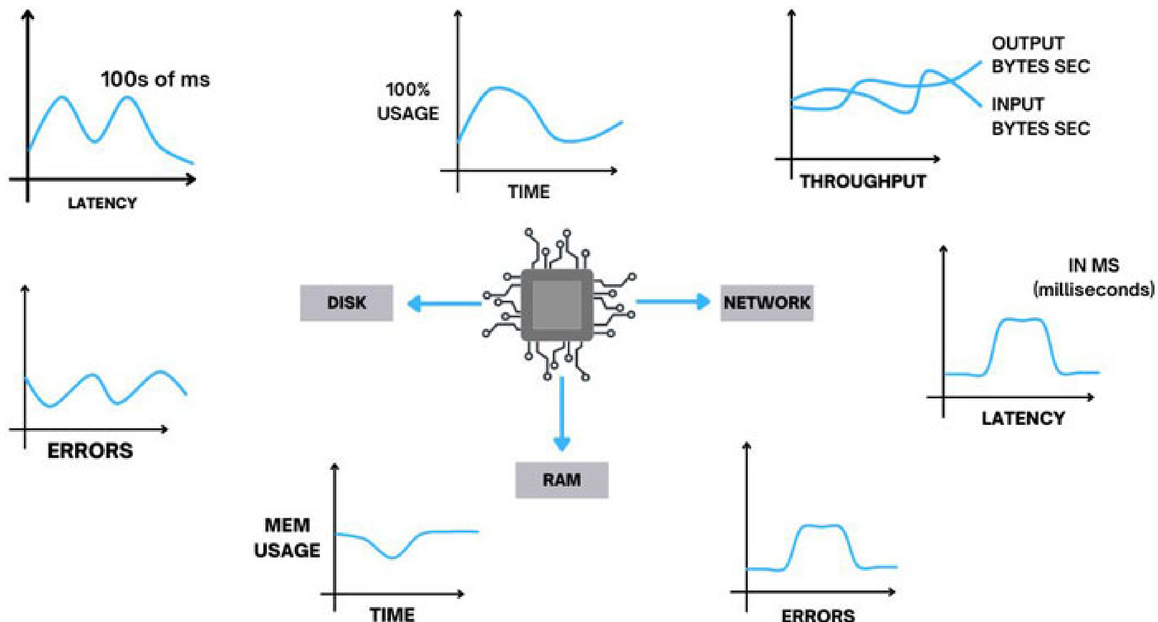


Figure 11.1: *Performance monitoring of CPU/network latency/memory usage, and so on.*

These are the following computer resources of interest:

- **CPU/GPU**: Processing load. (GPU in the case of machine learning workloads, which is becoming more and more common.)
- **Memory**: Consumption. Paging/Page swapping.
- **Storage**: Throughput, Latency, and Errors
- **Network**: Throughput, Latency, and Errors.

Utilization Saturation and Errors (USE) methodology by Brendan Gregg

Brendan Gregg is a Senior Performance Engineer at Netflix and promotes the USE method on his website (**https://www.brendangregg.com/usemethod.html**).

Utilization Saturation and Errors (**USE**) methodology works as follows:

- Determine the utilization of the resources*. If they are maxed out, investigate further what is loading up the resources.
    - *Resources are CPU, memory, network, and storage.
- Determine any saturation of resources. Saturation is not as straightforward as utilization, and it takes practice to know how to measure saturation. Saturation measures the degree to which the resource has work queued up that it cannot service. The queue is large for a resource; investigate further why the queue is large. The saturation problem is illustrated using a bank teller analogy below:

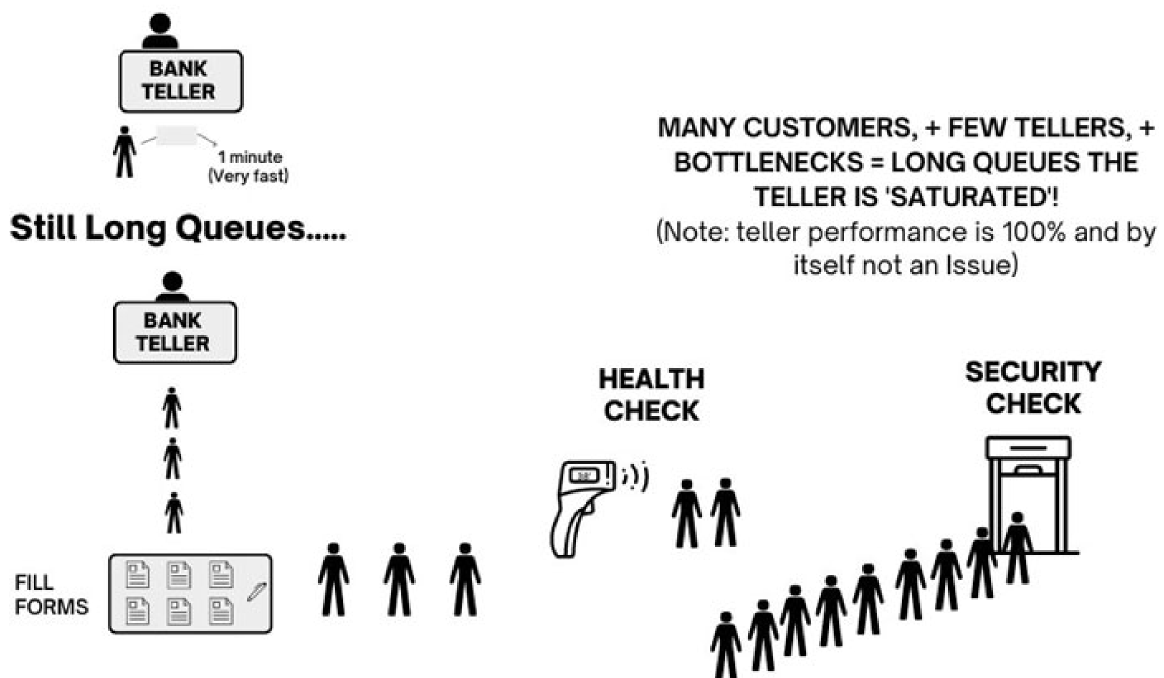


Figure 11.2: *Computer saturation problem*

- Determine errors in any of the resources. CPU and memory errors are rare (but one of the hardest to diagnose if they exist). More likely are network and storage errors. Investigate further the contributing factors for the errors.

Using the USE method; you can get to the contributing factors of a performance issue 80% of the time (source **https://www.brendangregg.com/usemethod.html**)

Kubernetes-specific performance tools

As more and more Kubernetes usage moves to the public cloud, you might not even have access to the underlying host. The USE method principles still apply and guide you as you use the Kubernetes-specific tools.

kubectl top command

It is similar to the top tool in Linux and lists all the pods by resource usage. To run, use the following:

```
kubectl top pod
```

To view the pods that use the most resources. For nodes' resource consumption, use the following:

```
kubectl top node
```

Sample output for both is shown as follows:

```
kc top pod -A
NAMESPACE     NAME                              CPU(cores)   MEMORY(bytes)
ingress-nginx  ingress-nginx-controller-8cf5559f8-
x7q6w  2m      137Mi
kube-system   coredns-78fcd69978-8nvpn           3m       24Mi
kube-system   coredns-78fcd69978-v78f9           3m       22Mi
kube-system   etcd-docker-desktop              26m      93Mi
kube-system   kube-apiserver-docker-desktop       57m      330Mi
kube-system   kube-controller-manager-docker-desktop   18m      61Mi
kube-system   kube-proxy-m8wpk                 1m       27Mi
kube-system   kube-scheduler-docker-desktop       4m       31Mi
kube-system   metrics-server-8bb87844c-gdcqj       4m       15Mi
kube-system   storage-provisioner              2m       14Mi
kube-system   vpnkit-controller                1m       8Mi
kc top node
NAME        CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
docker-desktop   208m      5%    2881Mi      22%
```

I had to follow the instructions found in https://hashnode.sujaypillai.dev/enable-kubernetes-metrics-server-on-docker-desktop to get the preceding working. The only difference is that we used the following command: `kc edit deploy/metrics-server`

To add the `--kubelet-insecure-tls` **parameter as shown below:**

```
133          - --secure-port=4443
```

```
134              - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
135              - --kubelet-use-node-status-port
136              - --kubelet-insecure-tls # Add this parameter which is not present in original download file
137              image: k8s.gcr.io/metrics-server/metrics-server:v0.4.1
```

Figure 11.3: Changes made for getting kubectl top command working

To make the change to the arguments.

Kubernetes dashboard

You can use the Kubernetes dashboard to view your resource utilization quickly. Please note that this has been the entry vector for many Kubernetes hacks, so be very careful about allowing access to the dashboard. It used to be deployed as a default in many Kubernetes distributions, but no longer due to the security vulnerability.

I used the instructions from https://willschenk.com/articles/2021/k8_dashboard_on_docker_desktop/ to get it running on my local docker desktop Kubernetes installation. The dashboard looks as shown in the following figure:

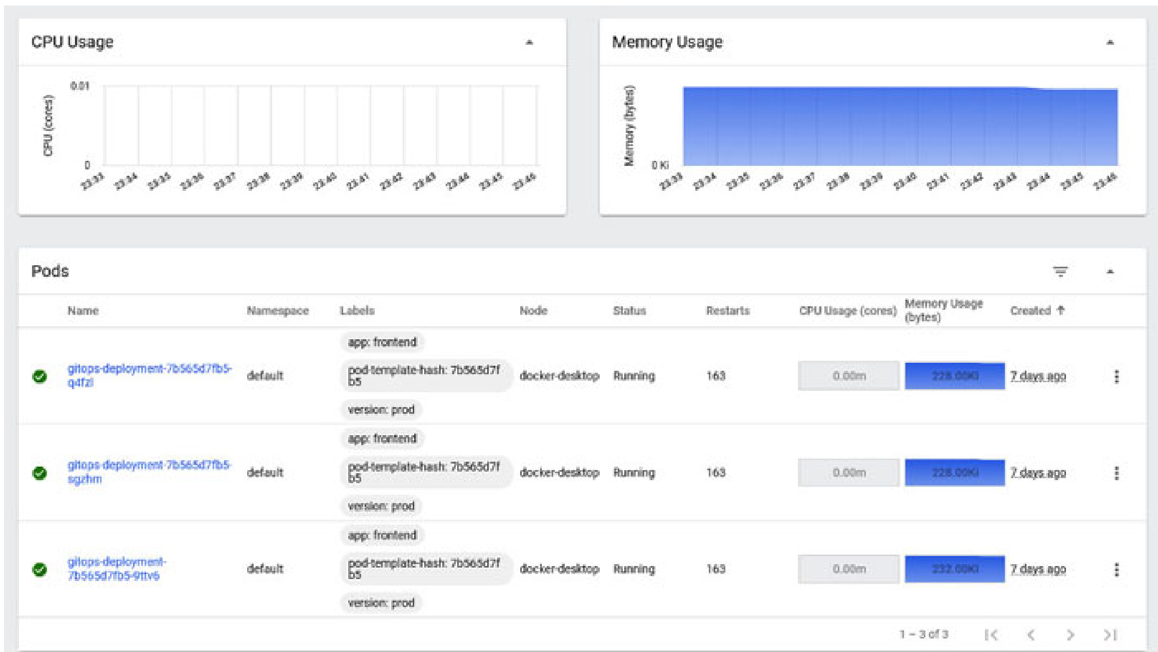


Figure 11.4: Kubernetes dashboard

Top 10 performance optimizations

(Sources: **https://platform9.com/blog/10-kubernetes-performance-tips/** and **https://opsani.com/blog/10-kubernetes-performance-tuning-tips-best-practices/**)

We will use a symptom-based approach to determine what optimization would work best for a given set of symptoms. As in real life, symptoms are just that, and multiple contributing factors must be considered before determining the course of action.

In case of an interview, when asked *How will you fix Kubernetes issues?* Or *What optimization would you do?* You should issue a caveat like "it is very context specific" and then ask. *What are the symptoms?* The goal is to demonstrate that you are methodical in your approach to solving problems.

Try to match any of the following symptoms, or if they do not, ask if they are seeing anything like the ones listed as follows.

Symptom #1

The Pod takes too long to come up or does not start reliably.

Optimization #1

Container image size

Depending on the image pull policy set in the deployment. The images must first be downloaded onto the worker node before the Pod can start. Large images mean that the time it would take to download would be too long. This would be a problem with rapid releases and deployment on hundreds of nodes.

The following are the steps to reduce the image size:

1. Use Docker multi-stage builds—which puts the bare minimum of data required to run the application.
2. Have only one application per Docker image; if required, combine the containers that need to run together in a Pod.
3. Use container optimized based images such as alpine. Alpine base images start under 5MB!
4. Remove any unneeded packages from the image.

Symptom #2

Certain pods/namespaces/workloads use up more than their fair share of the resources.

Optimization #2:

Pod resource limits

Review the CPU/memory limits specified in the pods that consume too many resources. In most cases, there might be no limits set, so they need to be added to only use their fair share.

Optimization #3:

Namespace-based resource limits

Like noisy neighbors need a good fence, namespaces that can take up too many resources due to rogue Pods can be restricted by specifying namespace level CPU/memory limits.

You can create a **`LimitRange`** that sets the limits for CPU/memory for all Pods in the namespace using a definition like the following:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range-example
spec:
  limits:
  - default:
  cpu: 800m
defaultRequest:
  cpu: 800m
max:
  cpu: 800m
min:
  cpu: 200m
  type: Container
```

Optimization #4:

Node tainting/node affinity

Certain workloads genuinely need high resources or specific hardware.

By specifying node affinity in the workload definition, you can restrict those workloads to only running on those specific worker nodes. Use the **`nodeSelector`** in your workload definition like the following:

```
…
spec:
  containers:
  - name: demo
  …
  nodeSelector:
  GPUType: HIGHPERFGPU
```

Node tainting works the other way. You can keep workloads from being scheduled on those highly used nodes by tainting a node.

```
kubectl taint nodes <nodename> needgpu=true:NoSchedule
```

Only workloads with the toleration **`needgpu=true`** would be allowed to run on **`<nodename>`**.

Optimization #5:

Set pod priorities

You can set certain Pods to be launched first or later by using **`PriorityClass`**. This will ensure that the critical Pods are started first. Add the following to the workload/Pod definition:

```
priorityClassName: high-priority
```

Symptom #3

Kubernetes itself is running slow across all nodes.

Optimization #6:

Increase resources for worker nodes

This would be another advantage for public clouds. You could dynamically increase the CPU/memory allocated for the workers' nodes on the fly. This optimization is done when the nodes are starved of resources, and all other optimizations will not work.

Optimization #7:

Configure Kubernetes features

Kubernetes has extra features that are configurable. Changing them could help the performance. The complete list is present at **https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/**. Some interesting ones are as follows:

- CPU manager (**https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/**) allows you finer control over the placement of sensitive workloads to the CPUs that they must be scheduled on.
- Pod overhead (**https://kubernetes.io/docs/concepts/scheduling-eviction/pod-overhead/**) lets you specify an overhead on CPU/memory limits that can be referenced in Pod definitions.

Optimization #8:

Switch to minimal host

Well-tuned operating systems specifically tuned for Kubernetes, such as Talos (**https://www.talos.dev/**). They do not have any extra services (like support for printing) running and thus consume minimum overhead. This option should be considered only when you have an experienced team that can handle the worst-case scenario of the underlying OS provider going away or the company behind it stops supporting it.

Symptom #4

Resource utilization is optimal. Still customers report application is slow.

Optimization #9:

Move the cluster closer to the customer

Network latency could be an issue for your customers. Deploying the applications to a data center close to your customer could help.

Symptom #5

Reading/writing data is slow, specifically for database-dependent services.

Optimization #10:

Configure storage

Depending on storage access patterns, adjust the storage class that a Pod uses. In the public cloud, the options would be to use SSDs for storage or switch to natively supported DB like Cosmos DB.

Conclusion

Kubernetes deployments, once under load, will run into performance issues. Recalling that Kubernetes launches regular processes, we got a brief introduction to USE methodology that can diagnose 80% of any computer performance issue, including those encountered in Kubernetes. Using Kubernetes-specific performance tools such as the command line `kubectl top` command and the Kubernetes dashboard gets us closer to solving Kubernetes performance issues. Symptom-based methodology and the top 10 performance fixes for Kubernetes give us enough information to tackle interview questions about Kubernetes performance optimizations. The symptoms and the optimizations are as follows:

- **Symptom #1:** The Pod takes too long to come up or does not start reliably.
    - **Optimization #1:** Container image size
- **Symptom #2:** Certain pods/namespaces/workloads use up more than their fair share of the resources
    - **Optimization #2:** Pod resource limits
    - **Optimization #3:** Namespace-based resource limits
    - **Optimization #5:** Set pod priorities
- **Symptom #3:** Kubernetes itself is running slow across all nodes.
    - **Optimization #6:** Increase resources for worker nodes
    - **Optimization #7.** Configure Kubernetes features
    - **Optimization #8:** Switch to minimal host
- **Symptom #4** Resource utilization is optimal. Still, customers report applications are slow.
    - **Optimization #9:** Move the cluster closer to the customer

- **Symptom #5** Reading/writing data is slow, specifically for database-dependent services.
    - **Optimization #10:** Configure storage

In the upcoming chapter on Kubernetes troubleshooting tips. We will be using a methodical approach to solving Kubernetes problems. You would be able to project confidence even during hands-on sessions in which you have to diagnose a Kubernetes problem.

Points to remember

- Solving performance issues in Kubernetes leverages your understanding of networking, processes, CPU, and RAM utilization.
- Utilization, Saturation, and Errors (USE) methodology solves 80% of production issues.
- Use the scientific method based on experience, start with a hypothesis, and validate it using the data.
- Based on symptoms, you can perform a quick analysis and try out simple steps to solve performance issues.

Interview questions and answers

1. **How do you approach performance issues in Kubernetes?**

   I approach them as solving any process performance issues, as running containers are processes. Kubernetes API server, etcd servers are also processes. Based on the information available and the symptoms, I create a hypothesis and check the resource usage using the utilization, saturation, and errors methodology. If my hypothesis checks out, I apply the fixes for the short term to mitigate the issue. I participate in retro meetings to figure out how the issue can be prevented from happening again.

2. **What will you do if certain Pods/Namespaces/Workloads use up more than their fair share of the resources?**

   I will review the CPU/memory limits placed on pods/namespaces/workloads and see if fixing them helps. If certain workloads truly require more resources, I will move them to nodes with larger resources using node tainting/node affinity. In rare cases, I will review if setting pod priorities via `PriorityClass` would help.

CHAPTER 12
Kubernetes Troubleshooting Tips

Introduction

During the interview, you might be presented with a problem or describe a scenario and asked to troubleshoot. We will go through all the steps that you need to ask the interviewer to identify the issue.

In this chapter, you will learn the following:

- Steps you should take to determine why services are not available.
- Flow chart that you can conceptually memorize to answer the questions about Kubernetes troubleshooting.

Structure

In this chapter, we will discuss the following topics:

- Mental model to have while troubleshooting Kubernetes deployments.
- A step-by-step methodology to address Kubernetes problems.

Objectives

You should be able to address any Kubernetes issue methodically and, thus, demonstrate your mastery of Kubernetes principles to the interviewer. After understanding and practicing this chapter, you can answer questions on Kubernetes troubleshooting questions confidently.

Kubernetes troubleshooting mental model

For Kubernetes troubleshooting, you should start bottom first, starting with the following list in order:

- Pods
- Service

- Ingress
- DNS

As shown in the following figure:

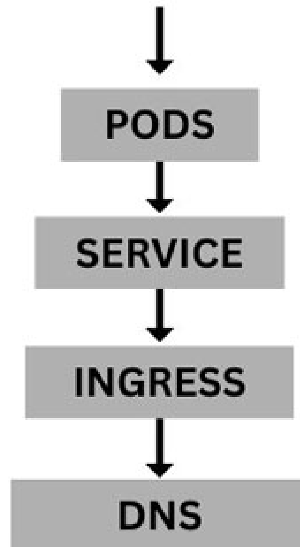


Figure 12.1: *Kubernetes troubleshooting model*

In the next section, we will go through how to troubleshoot each error condition at each level.

https://learnk8s.io/troubleshooting-deployments heavily inspired the following solutions. In your job, the flow chart, the commands, the different error messages, their meaning, and how to handle them are well illustrated in the troubleshooting guide.

Our focus in this chapter is to get you to answer the interview questions. Practice is the only way to achieve mastery in troubleshooting.

YAML problems

You can have issues with applying Kubernetes manifest YAML file itself:

```
kubectl apply -f manifest.yaml
error: error validating “"manifest.yaml”:": error validating
data: apiVersion not set; if you choose to ignore these errors,
turn validation off with --validate=false
```

Using the Visual Studio Code Kubernetes extension (**https://marketplace.visualstudio.com/items?itemName=ms-kubernetes-tools.vscode-kubernetes-tools**), you can detect these errors by opening the file. The red squiggly line shown as follows shows something wrong with the file. Clicking on the yellow light bulb gives you more information on the error (refer to the following figure).

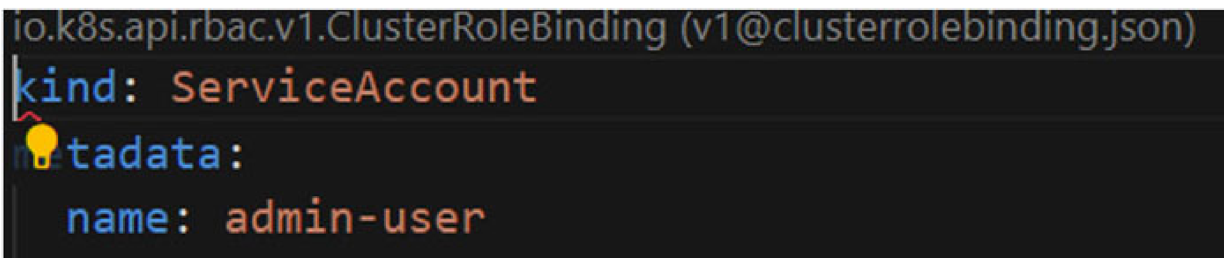


Figure 12.2: *Visual Studio hint*

You can also use one of the many online YAML validation tools and docs to fix the YAML/Kubernetes syntax issues.

Troubleshooting pods

Once you get the Pod running reliably, most other problems can be solved. If the Pod is not up, efforts in other areas will not fix the issue. The Pods' performance issues can be addressed using the **`kubectl top`** command and other measures like reducing image size.

Get pod information

The first step is to get information about the Pod itself. You can get that by running:

```
kubectl get pods --namespace=frontend
```

The output should be something like the following:

```
Good
NAME                                        READY
STATUS     RESTARTS    AGE
nginx-blue-deployment-5778cb9488-4ssqj   1/1      Running
0          2m17s
nginx-blue-deployment-5778cb9488-68zwm 1/1      Running
0           2m17s
```

```
nginx-blue-deployment-5778cb9488-pf5tj 1/1     Running
0          2m17s
Bad
NAME                                  READY STATUS
RESTARTS         AGE
nginx-deployment-54b9b5f666-tf2ft 0/1   ImagePullBackOff
0          87s
```

The two most common ones are as follows:

1. **ImagePullError**
2. **ImagePullBackOff**

It means that Kubernetes is having trouble downloading the image to the worker node.

Get Pod definition

Get the Pod definition using the following command. It should give you more information on the error:

```
kubectl get -o yaml pods/nginx-deployment-54b9b5f666-tf2ft
```

For example, the following output shows that Kubernetes cannot find the image specified:

```
started: false
state:
  waiting:
   message: Back-off pulling image "alpine:4.15.3"
    reason: ImagePullBackOff
```

In the preceding, check for the image field and the path listed. Try the following troubleshooting steps:

- Try downloading it locally using docker pull.
- If possible, try downloading it on the worker node.
- Check the version and the image path.
- Check the registry credentials if you are retrieving them from a private repository. You can use the following command to check for docker credentials:

  ```
  kubectl get secret regcred --output=yaml
  ```

Once you fix the path and the credentials for the Pod, the Pod should be present in the worker node, and Kubernetes should be able to attempt to start it.

Double-check by using events

Using the following command, you can ensure that the Pod is present on the worker node:

```
kubectl describe pods/ gitops-deployment-85b4f9f49d-5mj5j
```

Make sure the following line is present in the output:

```
Normal   Pulled     50s   kubelet            Successfully pulled
image "alpine:3.15.2" in 1.1143289s
```

You can also use the following command:

```
kubectl get events --all-namespaces
```

To get the big picture of what is happening in your cluster. You would see the same event listed in the output. You can narrow down the scope by specifying the namespace:

```
kubectl get events --namespace=frontend
```

Logs, logs, logs

Figure 12.3: *Logs*

The Pod might be restarted, or the application itself may not come up correctly configured. You can view the Pod logs by running the following:

```
kubectl logs pods/ gitops-deployment-85b4f9f49d-5mj5j --
namespace=frontend
```

Good logging is important. To be Kubernetes friendly, all logs should go to `stderr` or `stdout`. Sending the output to files in the container makes it hard to debug, as you need access to the container to view the logs.

You can also `tail` logs by running

```
kubectl logs -f pods/ gitops-deployment-85b4f9f49d-5mj5j --
namespace=frontend
```

It will show the app's logs as they are updated in the container.

Pod configuration

If the Pod is not starting because it cannot mount the configs, ensure the configurations are correct by running the following command.

```
kubectl get configmaps --namespace=frontend
```

Make sure that the namespace is correct. You might have the Secrets, and ConfigMaps defined in a different namespace (most likely the default namespace). Secrets and ConfigMaps are namespace specific and not cluster-wide.

Also, make sure that the **secrets** are correct.

```
kubectl get secrets --namespace=frontend
```

Storage configuration

If the storage mounts are not accessible to the Pod, then it will not be able to start. Check that the storage is available by ensuring that volume mounts are correct:

```
kubectl get pvc --namespace=frontend
kubectl get pv --namespace=frontend
```

Verify using exec into the Pod

Run the **kubectl exec** command into a running Pod and verify that the application is available at the container Port and the service.

An example session is shown as follows:

```
kubectl run --image=alpine:3.5 -it alpine-shell -- /bin/sh
/ # apk update
/ # apk add curl
/ # echo get the POD IP by running the following command
outside this shell:
```

```
/ # PODIP=$(kubectl get -o jsonpath='{.items[0].status.podIP}'
pods)
/ # curl http://PODIP
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Flow chart summary

Pods troubleshooting flowchart has been given in the following figure:

PODS TROUBLESHOOTING MODEL
FLOW CHART

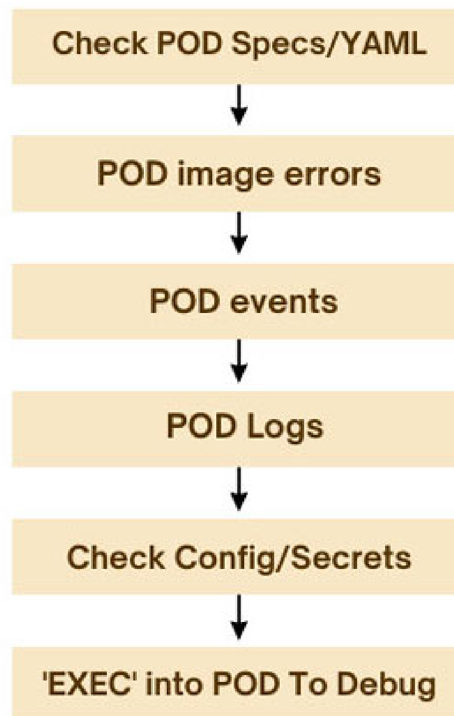


Figure 12.4: *YAML syntax/validation->Pod image present->Pod events->Logs->ConfigMaps->Secrets->Storage Mounts-> exec into Pod*

Pod troubleshooting conclusion

The Pod should be running now once you have followed and verified all the steps in the Pod troubleshooting flowchart.

Troubleshooting services

Once the Pods have been verified as running successfully, the next stop is service. Service directs the traffic to the underlying Pods independent of which worker node they are running.

Get service information

You can obtain all the service names by running the following:

```
kubectl get svc --namespace=frontend
```

You will get an output listing the service names in that namespace:

```
NAME          TYPE        CLUSTER-IP     EXTERNAL-IP
PORT(S)         AGE
frontend      NodePort    10.99.30.154
<none>         8080:31035/TCP   3s
kubernetes    ClusterIP
10.96.0.1       <none>        443/TCP         21d
```

To get the services running in all namespaces, use **`--all-namespaces`** instead of **`--namespace=`**….

Get the service definition

```
kubectl get -o yaml svc/frontend
```

You will get an output like this:

```
apiVersion: v1
kind: Service
metadata:
annotations:
  …
name: frontend
namespace: frontend
…
spec:
clusterIP: 10.99.30.154
```

```
clusterIPs:
- 10.99.30.154
…
ports:
- nodePort: 31035
  port: 8080
  protocol: TCP
  targetPort: 80
selector:
  app: frontend
sessionAffinity: None
type: NodePort
status:
loadBalancer:
  ingress:
  - hostname: localhost
```

To test whether the service is working, run the following command:

```
kubectl get svc/frontend --namespace=frontend
```

Examine the output:

```
NAME       TYPE       CLUSTER-IP     EXTERNAL-IP
PORT(S)          AGE
frontend   NodePort   10.99.30.154
<none>        8080:31035/TCP   9m17s
```

The **`TYPE`** can be of **`NodePort`** or **`LoadBalancer`**. The cloud provider provides the **`LoadBalancer`** port and IP. The principles are the same; we will focus on the **`NodePort`** scenario.

To check if the service works for a service exposed via **`NodePort`**, run the following command:

```
curl http://<Worker Node IP>:<NodePort shown in the output
(within 30000-32767)>
```

For example:

```
curl -o - -I http://localhost:31035
HTTP/1.1 200 OK
```

If you get something like the following, or if it just hangs, it is troubleshooting time.

```
curl -o - -I http://localhost:31440
curl: (52) Empty reply from server
```

Make sure the labels match

The service definition labels should match those present in the workload definition:

```
selector:
  app: frontend
  version: prod
```

The preceding selector will fail if the workload definition has only the following:

```
selector:
  matchLabels:
    app: frontend
```

Or if the version is different, as shown as follows:

```
selector:
  matchLabels:
    app: frontend
    version: green
```

Make sure that the container port and the target port match

The workload container port number is defined as follows:

```
ports:
- containerPort: 80
```

Should match the target port specified in the service definition:

```
ports:
- port: 8080
  targetPort: 80
```

Flow chart

The flowchart depicting service troubleshooting is given in the following figure:

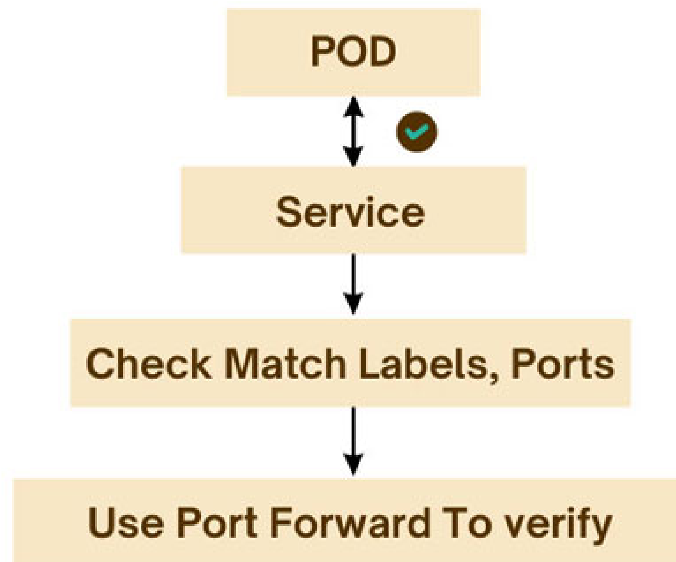


Figure 12.5: *Service troubleshooting flowchart—Pod-> Service Proxy -> Service NodePort*

Troubleshooting Ingress

Ingress provides an IP and Port to service for access from outside the cluster. Follow the following instructions to troubleshoot issues with Ingress.

Get Ingress information

```
kubectl get ingress --namespace=frontend
NAME                CLASS    HOSTS
ADDRESS            PORTS    AGE
nginx-ingress       nginx    nginx.localdev.me    192.168.65.4
80       22d
```

Get Ingress definition

```
kubectl get -o yaml ingress/nginx-ingress --namespace=frontend
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
```

```
…
  labels:
    name: nginx-ingress
  name: nginx-ingress
  namespace: frontend
spec:
  ingressClassName: nginx
  rules:
  - host: nginx.localdev.me
    http:
      paths:
      - backend:
          service:
            name: frontend
            port:
              number: 8080
        path: /
        pathType: Prefix
status:
  loadBalancer:
    ingress:
    - ip: 192.168.65.4
```

The previous command shows that the backend service is named frontend and listening on Port 8080. From the previous section, we know that the service is working. Confirm that the service name and port are correct.

To check whether the Ingress is working, run the following:

```
curl http://<host in the ingress definition>/
```

In this case, it is:

```
curl -sL -w "%{http_code}" -I "nginx.localdev.me" -o /dev/null
```

If you get a 200 as the output, then we are good. If not, then it is a fun time ☺.

Confirm that the endpoints are working

Make sure that the backend service is working:

```
curl -sL -w "%{http_code}" -I "<kubernetes host>:<service node
port>" -o /dev/null
```

Get the Ingress Pod

Get the Ingress Pod by running the following command:

```
kc get all -n ingress-nginx
NAME                                                   READY
STATUS      RESTARTS       AGE
pod/ingress-nginx-admission-create--1-6bkx9    0/1
Completed   0              14d
pod/ingress-nginx-admission-patch--1-xwjhm     0/1
Completed   1              14d
pod/ingress-nginx-controller-cb87575f5-snk4m   1/1
Running     1 (6d6h ago)   14d
```

Get the Ingress logs

Get the logs by running the following command using the Pod name from the previous step:

```
kc logs pod/ingress-nginx-controller-cb87575f5-snk4m -n
ingress-nginx
---------------------------------------------------------------
--------
NGINX Ingress controller
  Release:       v1.1.2
  Build:         bab0fbab0c1a7c3641bd379f27857113d574d904
  Repository:    https://github.com/kubernetes/ingress-nginx
  nginx version: nginx/1.19.9
---------------------------------------------------------------
--------
W0424 22:36:07.312148       6 client_config.go:615] Neither --
kubeconfig nor --master was specified.  Using the
inClusterConfig. This might not work.
I0424 22:36:07.341369       6 main.go:223] "Creating API
client" host="https://10.96.0.1:443"
W0424 22:36:29.902367       6 main.go:264] Initial connection
to the Kubernetes API server was retried 1 times.
```

```
I0424 22:36:29.902449       6 main.go:267] "Running in
Kubernetes cluster" major="1" minor="22" git="v1.22.5"
state="clean" commit="5c99e2ac2ff9a3c549d9ca665e7bc05a3e18f07e"
platform="linux/amd64"
```

Check for any certificate errors. Once you fix the certificate path and the port issues, the service should be available.

Troubleshooting flow chart

Let us now have a look at the Ingress troubleshooting flowchart in the following figure:

INGRESS TROUBLESHOOTING FLOWCHART

Figure 12.6: *Ingress troubleshooting flowchart—get Ingress info, curl <ingress host>, check Ingress Pod, check certificate errors, verify via curl*

DNS issues

This troubleshooting step depends upon how the cluster IP is associated with the DNS record. Use the dig tool to troubleshoot the issue.

Worst case scenario

Sometimes all the settings will be right, and still, the application may not behave correctly. This is a strict no-no in production; sometimes, desperate situations require desperate measures.

You can put SSH into the Pod itself by running the following command:

```
kubectl exec -it -n ingress-nginx pod/ingress-nginx-controller-
cb87575f5-snk4m -- /bin/sh
/etc/nginx $ ls -lth
total 116K
-rw-r--r--    1 www-data www-data   21.4K Apr 24 22:36
nginx.conf
-rw-r--r--    1 www-data www-data       2 Apr 24 22:36
opentracing.json
…
/etc/nginx $ ps aux
PID   USER     TIME  COMMAND
     1 www-data  0:00 /usr/bin/dumb-init -- /nginx-ingress-
     controller --publish-service=ingress-nginx/ingress-nginx-
     co
     6 www-data  8:47 /nginx-ingress-controller --publish-
     service=ingress-nginx/ingress-nginx-controller --election-
     id
    27 www-data  0:00 nginx: master process
    /usr/local/nginx/sbin/nginx -c /etc/nginx/nginx.conf
    33 www-data  0:43 nginx: worker process
    34 www-data  0:43 nginx: worker process
    35 www-data  0:46 nginx: worker process
    36 www-data  0:53 nginx: worker process
    37 www-data  0:02 nginx: cache manager process
   175 www-data  0:00 /bin/sh
   183 www-data  0:00 ps aux
/etc/nginx $
```

It will give you a shell from which you can work on. (An excellent secure setup will/should not let you do this in production.) Once inside the container, you can run any command allowed for the user.

Sometimes the user has limited rights. In that case, you can launch an image with an entry point override and user override to see what is going on locally with:

```
docker run --rm -it -u root:root --entrypoint sh  nginx
```

Flow chart

The whole chapter is illustrated in the following figure:

KUBERNETES TROUBLESHOOTING FLOWCHART

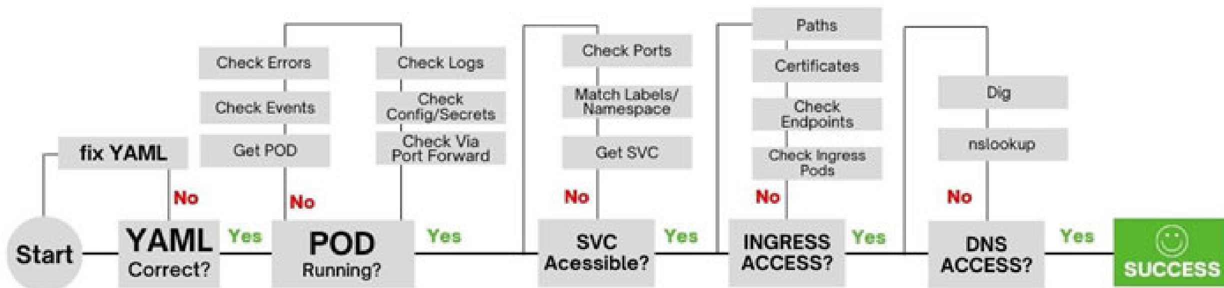


Figure 12.7: *Kubernetes troubleshooting flowchart*

Conclusion

We presented a mental model on how to handle Kubernetes issues. A structured approach to debugging problems instead of relying on instinct separates professionals from amateurs. The code-red situation can get quite hot, and the people who approach it calmly and methodically are prized in any organization.

The methodology will come naturally to you with practice, and it can be practiced at home using Raspberry Pi or through software simulation. You will answer the interview questions on troubleshooting with confidence, as the interviewer will notice your methodological approach even if you do not figure out the contributing factors in time.

We have also practiced hands-on exercises on handling most of the Kubernetes issues using a systematic approach.

In the upcoming chapter, we will take a quick tour of Kubernetes tools and extensions. We will cover commonly used tools such as Helm, Kustomize, and kubectl.

Points to remember

- Having a structured approach for troubleshooting helps in drama-less diagnosis and fixing of issues.

- Most of the Kubernetes issues you will encounter will be with Pod not coming up and service not being accessible.
- Starting with the bottom of the heap ensures that you have covered your bases.
- The order of diagnosis is Pods->Services->Ingress->DNS.
- For Pods, review YAML, Pod Status, Pod Logs, ConfigMaps->Secrets->Storage Mounts.
- For Services, ensure network traffic is flowing from Pod-> Service Proxy -> Service NodePort.
- For Ingress troubleshooting, Get Ingress Info, `curl <ingress host>`, check Ingress Pod, and check certificate errors.
- For DNS issues, use the dig tool and DNS configuration.

Interview questions and answers

1. **How do you troubleshoot Kubernetes issues?**

   I have a structured approach to troubleshooting issues. Even though it might seem that it is a waste of time, this ensures that we have covered the bases and not gone after red herrings. The structured approach is to start from the bottom of the heap, which is Pods, and all the way up to DNS.

2. **I have an issue with a Service; how will you troubleshoot it?**

   After making sure that the Pods are ok, I will check the Service configuration to make sure that the namespace and matching labels are correct. I will make sure that the IP/port of the service matches the expected IP/port. Using curl or Netcat, check that the service responds on the IP/port. I will start a diagnosis pod in the same namespace to ensure that I can reach the underlying Pod IPs/Ports plus that the Service IP/port is reachable.

CHAPTER 13
Kubernetes Tools and Extensions

Introduction

Kubernetes provides us with great primitives that enable us to orchestrate container deployment. Kubernetes can be viewed as an excellent 200-piece lego set that can build anything you want. With such flexibility and complexity, we require tools and extensions that provide higher-level abstractions to manage Kubernetes at scale.

The good news is that Kubernetes was built with extensibility in mind. Many tools by companies (big and small) and the open-source community are available, most of which are free. We will go through a selected few tools and extensions so that you can answer interview questions such as:

How will you manage the deployment of apps across multiple environments?

What are the tools available to manage apps at scale?

In this chapter, you will learn about the most common/popular tools used to manage apps at scale in Kubernetes.

Structure

In this chapter, we will discuss the following topics:

- Popular tools available for Kubernetes
- Hands-on exercises for Helm and Kustomize
- Links to other Kubernetes tools

Objectives

Cover the basics, plus hands-on experience on the popular tools for app management or communities so that you can confidently answer questions in this area. After reading this chapter, you will have hands-on experience with

app management tools such as Helm and Kustomize and awareness of similar tools such as Helmsman and Helmfile.

Helm

Helm is the *de facto* package manager for Kubernetes. Like a good package manager (such as rpm, apt, or chocolatey), it provides the following:

- Dependency management
- Configuration based on templates
- Install/upgrade/uninstall functionality

Need for Helm

One of the primary cases for using Helm is the ability to deploy open-source applications the right way on Kubernetes. You could build your own Kubernetes manifest files for WordPress and use the Docker images directly.

Then, you would have to maintain them and keep them up to date with the latest features available on the application. Because it is a custom code, anybody who must sustain/understand it will have a tough time. Getting an application to run in production is not a trivial task, as it must scale, be secure, and be easy to observe.

Helm provides a standard packaging mechanism through which application developers can package their applications so that the users can install them by just typing the following command:

```
helm install my-release <application name>
```

Helm repo

There are approximately 9K applications (which are still growing) that are available at **https://artifacthub.io/**, as shown below:

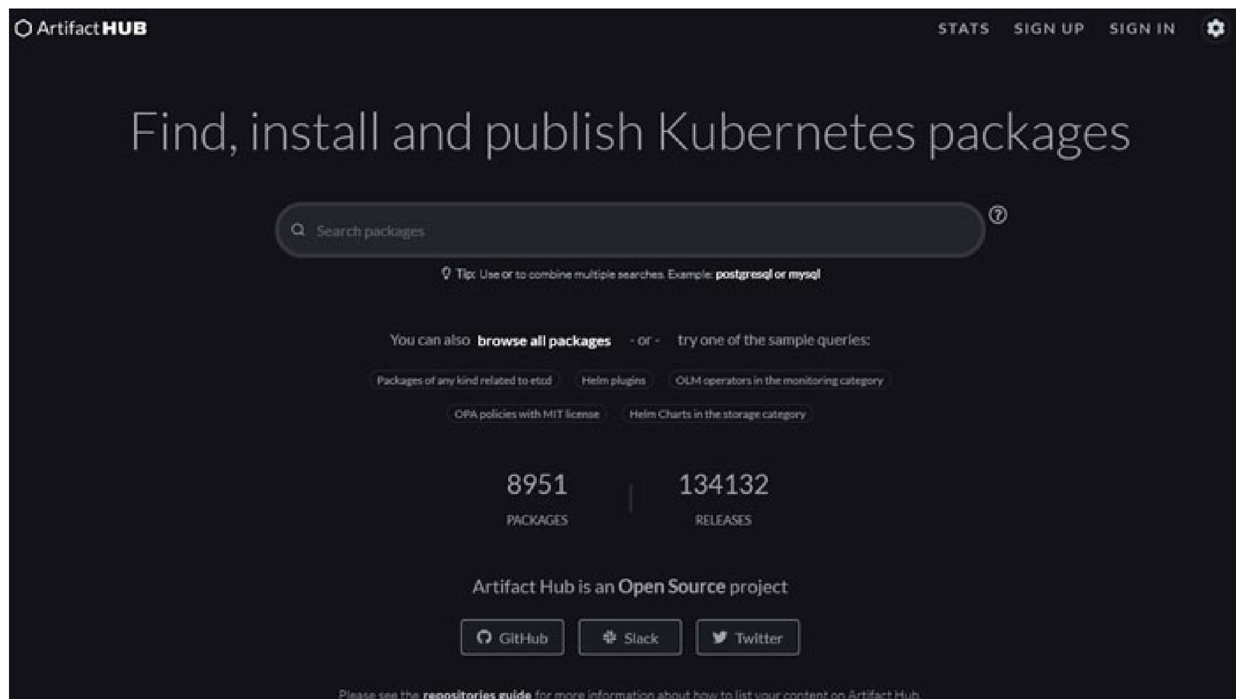


Figure 13.1: *ArtifactHub.io registry for Helm charts*

> **You can host your own Helm repository if required for security concerns. See https://chartmuseum.com/docs/#using-with-local-filesystem-storage.**

Sample deployment: Install WordPress

Install WordPress using the following commands:

```
helm install my-release bitnami/wordpress
```

Check for the status using the following command:

```
helm status my-release
```

It will let you know whether the service is ready (it takes 5–10 minutes):

```
Watch the status with the following:
```

Once the service is up and running, check on the URL to log in by following the instructions provided by **helm status <release-name>**:

```
export SERVICE_IP=$(kubectl get svc --namespace default my-release-wordpress --include "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
 echo "WordPress URL: http://$SERVICE_IP/"
```

```
echo "WordPress Admin URL: http://$SERVICE_IP/admin"
```

In this case, it was localhost. Here is the pudding:

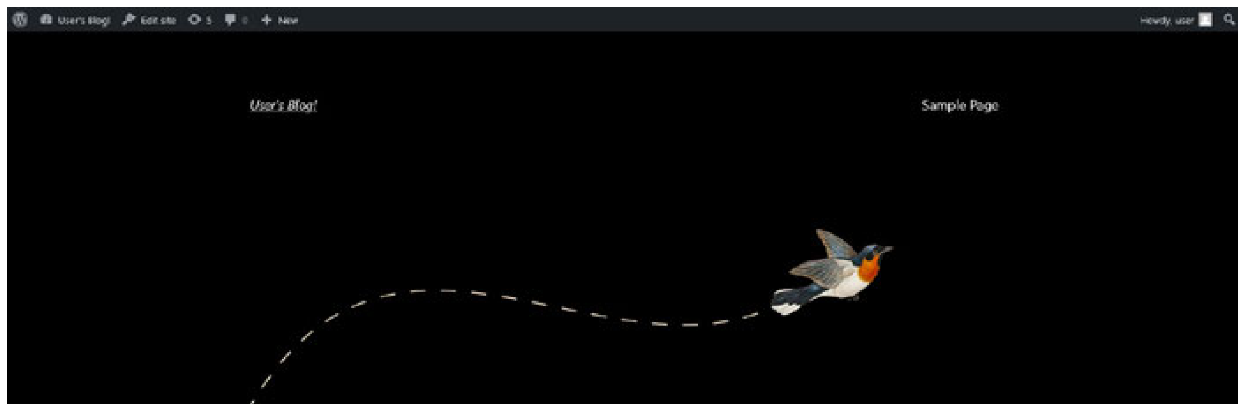


Figure 13.2: *Localhost*

And the admin page:

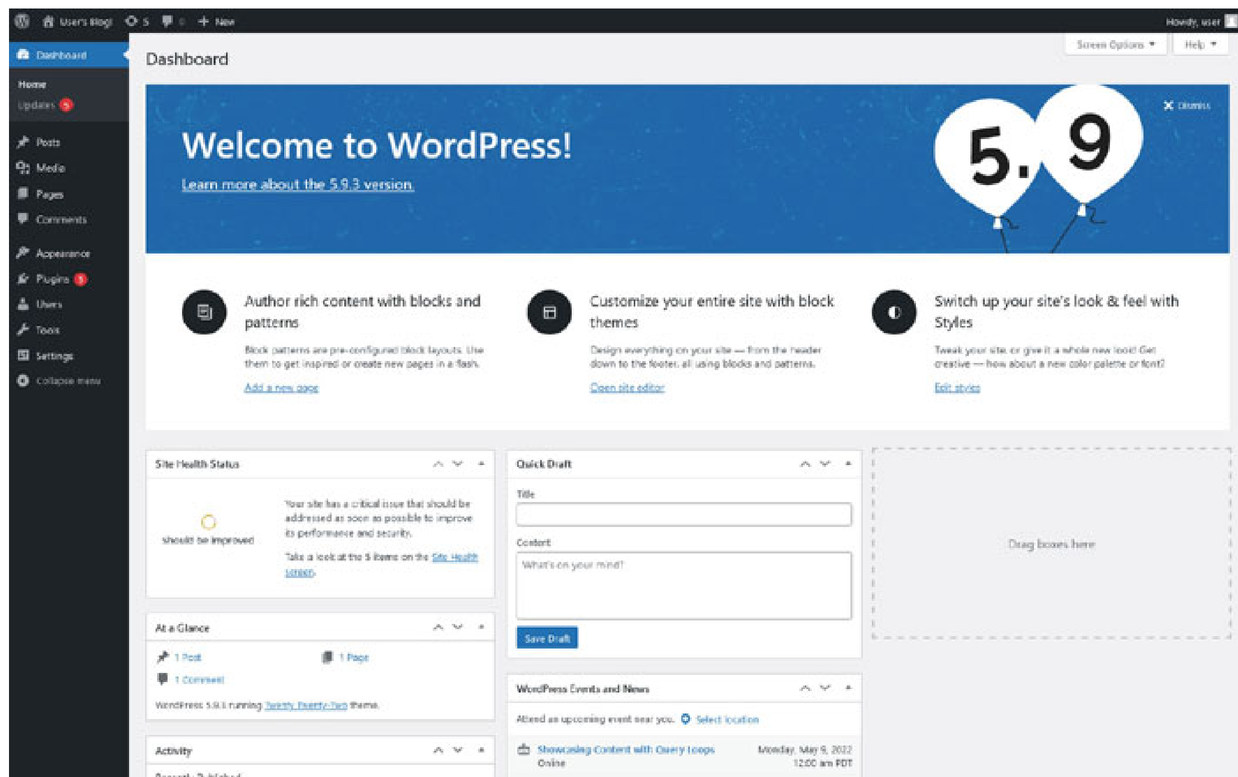


Figure 13.3: *Admin page*

Kustomize

Kustomize takes a different approach. Philosophically Kustomize developers believe that Helm adds unnecessary complexity and is difficult to debug.

With Kustomize, you primarily have Kubernetes YAML files rendered with values required into a folder. Using GitOPS or kubectl, installs/upgrades can be done using the plain Kubernetes files.

Any issues can be easily debugged as there are no templated values.

We will do a sample Kustomize deployment demonstrating its GitOps friendliness. We are going to use the code from https://github.com/gshiva/kustomize-examples. To get started, run the following command:

```
git clone https://github.com/gshiva/kustomize-examples.git
```

Run the following command:

```
cd kustomize-examples/frontend-app
kubectl apply -k overlays/dev/
```

As you can see from the previous command, kubectl already has built-in integration with Kustomize, so no separate installation is required.

By changing the files under dev, you can override settings in the base definitions. Combining this with Git changes and a simple watcher which applies these patches to the Kubernetes cluster, GitOps is achieved.

Helm or Kustomize?

In an interview, you might be asked what you would choose—Helm or Kustomize.

The following table highlights the structure and code differences between Helm and Kustomize:

Helm	Kustomize

Code structure:
my-helm-chart
`├── Chart.yaml`
`├── charts`
`├── templates`
`│   ├── NOTES.txt`
`│   ├── _helpers.tpl`
`│   ├── deployment.yaml`
`│   ├── ingress.yaml`
`│   ├── service.yaml`
`│   └── tests`
`│       └── test-connection.yaml`
`└── values.yaml`
	Code structure:
	`├── front-end-app`
	`│   ├── deployment.yaml`
	`│   ├── hpa.yaml`
	`│   ├── kustomization.yaml`
	`│   └── service.yaml`
	`└── overlays`
	`    ├── dev`
	`    │   ├── hpa.yaml`
	`    │   └── kustomization.yaml`
	`    ├── production`
	`    │   ├── hpa.yaml`
	`    │   ├── kustomization.yaml`
	`    │   ├── rollout-replica.yaml`
	`    │   └── service-loadbalancer.yaml`
	`    └── staging`
	`        ├── hpa.yaml`
	`        ├── kustomization.yaml`
	`        └── service-nodeport.yaml`

Mostly working on determining the right settings for `values.yaml`.
`replicaCount: 1`
`restartPolicy: Never`
`# Evaluated by the post-install hook`
`sleepyTime: "10"`
`index: >-`
`  <h1>Hello</h1>`
`  <p>This is a test</p>`
`image:`
`  repository: nginx`
	Mostly working on `kustomization.yaml` and the associated template files.
	`apiVersion: kustomize.config.k8s.io/v1beta1`
	`kind: Kustomization`
	`resources:`
	`  - service.yaml`
	`  - deployment.yaml`
	`  - hpa.yaml`

<table>
<tr><td>tag: 1.11.0
pullPolicy: IfNotPresent</td><td></td></tr>
<tr><td>A sampling of the Helm template code:
. . .
spec:
replicas: {{ .Values.replicaCount }}
template:
metadata:
{{- if .Values.podAnnotations }}
annotations:
{{ toYaml .Values.podAnnotations | indent 8 }}
{{- end }}
. . .</td><td>A sampling of Kustomization code which is plain Kubernetes manifest code. The override of values happens using overlay files.
apiVersion: apps/v1
kind: Deployment
metadata:
name: frontend-deployment
spec:
selector:
matchLabels:
app: frontend-deployment
template:
metadata:
labels:
app: frontend-deployment
spec:
containers:
- name: app
image: nginx:latest
ports:
- name: http
containerPort: 8080
protocol: TCP</td></tr>
</table>

Table 13.1: *File structure and code samples from Helm and Kustomize*

Interviewers do not think negatively (the good ones, at least) about people with strong opinions. Ideal candidates have strong opinions that are loosely held.

You should lead with your opinion if you have enough experience and have a strong view of templating versus a non-template-based approach. If you are new, you should know the strengths and weaknesses of Helm and Kustomize.

Helm	Kustomize
Pro: Strong Community	Passionate community
Pro: Thousands of apps	Can leverage any source of manifests, including Helm
Pro: Beginner-friendly in the beginning	Simple to debug and run
Con: Difficult to debug	Difficult to get started due to the lack of a package manager and repository

Table 13.2: Pros and cons of Kustomize

Like good open-source projects, it need not be Helm versus Kustomize. One prevalent pattern is to use widely available Helm charts to generate the YAML files, which are then customized using Kustomize.

Like the question of what programming language is best suited for the company, this question is answered by seeing the following:

- Existing code in the company
- Familiarity with Helm or Kustomize by the employees
- Frankly, the whim of the CTO or equivalent

The suggestion for the answer to Helm versus Kustomize is as follows:

- State the pros and cons of both.
- Demonstrate that you are willing to work on both.
- State your preference along with the why.
- Your view is that the proper framework depends on the developers, testers, and their familiarity.

Tools for tools

Helm itself is hard to manage for multiple environments. Many tools are available to help developers deploy to Kubernetes without requiring them to have a deep understanding of Kubernetes.

We will do a quick intro to these tools in the following sections. The objective is to demonstrate awareness of these tools to the interviewer.

Helmsman

Helmsman (https://github.com/Praqma/Helmsman) helps you manage Helm. Helmsman means the person who helms the ship. Helmsman works by using a simple **Tom's Obvious, Minimal Language** (**TOML**) to describe your deployments declaratively. You can run it using the helmsman binary or using the docker image.

Sample invocation:

```
helmsman -f example.toml
```

And the example.toml would have your apps defined like the following from (**https://github.com/Praqma/helmsman/blob/master/examples/example.toml**):

```
[namespaces]
[namespaces.production]
 protected = true
  [[namespaces.production.limits]]
   type = "Container"
   [namespaces.production.limits.default]
    cpu = "300m"
    memory = "200Mi"
[namespaces.staging]
 protected = false
  [namespaces.staging.labels]
   env = "staging"
  [namespaces.staging.quotas]
    "limits.cpu" = "10"
    "limits.memory" = "30Gi"
    pods = "25"
# define any private/public helm charts repos you would like to
get charts from
# syntax: repo_name = "repo_url"
# only private repos hosted in s3 buckets are now supported
[helmRepos]
argo = "https://argoproj.github.io/argo-helm"
jfrog = "https://charts.jfrog.io"
# myS3repo = "s3://my-S3-private-repo/charts"
# myGCSrepo = "gs://my-GCS-private-repo/charts"
# custom = "https://$user:$pass@mycustomrepo.org"
# define the desired state of your applications helm charts
# each contains the following:
[apps]
[apps.argo]
 namespace = "production" # maps to the namespace as defined in
 namespaces above
 enabled = true # change to false if you want to delete this
 app release [default = false]
```

```
chart = "argo/argo" # changing the chart name means delete and
recreate this release
version = "0.8.5" # chart version
### Optional values below
valuesFile = "" # leaving it empty uses the default chart
values
```

Helmfile

Helmfile's (https://github.com/helmfile/helmfile) goal is similar to Helmsman's goal of having a declarative spec for deploying helm charts. The invocation is as follows:

```
helmfile apply
```

Helmfile assumes that there is a file named **helmfile.yaml**. The contents of **helmfile.yaml** can be as simple as follows:

```
releases:
- name: prom-norbac-ubuntu
namespace: prometheus
chart: stable/prometheus
set:
- name: rbac.create
  value: false
```

The key benefit of Helmfile is its integration with the HashiCorp vault. Kubernetes has challenging secrets to set via git, as you should not put secrets in your repository. Links to the vault can be freely added, as no one can access the secrets without access to the vault.

K9s

K9s (https://k9scli.io/) is a cool terminal-based UI to interact with multiple Kubernetes clusters. As an admin of multiple clusters, it gives an easy interface to navigate and observe your clusters.

```
Context: docker-desktop                    <0> all       <a>      Attach    <l>
Cluster: docker-desktop                    <1> default   <ctrl-d> Delete    <p>
User:    docker-desktop                                  <d>      Describe  <shift-
K9s Rev: v0.25.18                                        <e>      Edit      <s>
K8s Rev: v1.22.5                                         <?>      Help      <f>
CPU:     n/a                                             <ctrl-k> Kill      <y>
MEM:     n/a
                                       Pods(all)[10]
 NAMESPACE↑   NAME                                  PF READY RESTARTS STATUS  IP            NODE            AGE
 default      frontend-deployment-6f7ddfcfdc-v8qdk  •  1/1          0 Running 10.1.0.186    docker-desktop  25h
 kube-system  coredns-78fcd69978-n2821              •  1/1          0 Running 10.1.0.179    docker-desktop  7d
 kube-system  coredns-78fcd69978-x8xq2              •  1/1          0 Running 10.1.0.180    docker-desktop  7d
 kube-system  etcd-docker-desktop                   •  1/1          0 Running 192.168.65.4  docker-desktop  7d
 kube-system  kube-apiserver-docker-desktop         •  1/1          0 Running 192.168.65.4  docker-desktop  7d
 kube-system  kube-controller-manager-docker-desktop •  1/1         0 Running 192.168.65.4  docker-desktop  7d
 kube-system  kube-proxy-kxrzx                      •  1/1          0 Running 192.168.65.4  docker-desktop  7d
 kube-system  kube-scheduler-docker-desktop         •  1/1          0 Running 192.168.65.4  docker-desktop  7d
 kube-system  storage-provisioner                   •  1/1          0 Running 10.1.0.181    docker-desktop  7d
 kube-system  vpnkit-controller                     •  1/1        849 Running 10.1.0.182    docker-desktop  7d

<pod>
```

Figure 13.4: *K9s CLI*

Awesome Kubernetes

The Kubernetes ecosystem is so vast that multiple curated *awesome* Kubernetes lists exist. For getting started, see **https://github.com/ramitsurana/awesome-kubernetes**. For tools, see **https://github.com/tomhuang12/awesome-k8s-resources**.

Conclusion

Kubernetes management is complex, especially managing application deployments across multiple environments. To manage at scale, you want the deployments to follow the GitOps model of all changes driven by code commits. You also wish the deployments to be described in a declarative fashion similar to Kubernetes manifests. Helm and Kustomize are two popular tools used for declaratively managing Kubernetes. The philosophical difference between Helm and Kustomize lies in whether templating should be used for the actual deployment or just plain Kubernetes manifests should be used. You can get the best of both worlds by using manifests generated by Helm as input to Kustomize. A brief introduction to tools such as Helmsman, Helmfile, and K9s, along with pointers to the awesome curated lists for Kubernetes, provide excellent starting points for your Kubernetes mastery journey.

The upcoming chapter will investigate advanced plugins to Kubernetes for authentication, storage, network, and security.

Points to remember

- The advantage and disadvantage of Kubernetes is its ecosystem of tools.
- Key tools for application deployment are Helm and Kustomize.
- There are tools for managing Helm, such as Helmsman and Helmfile.
- K9s is a cool terminal-based UI that makes managing multiple Kubernetes clusters easier.

Interview questions and answers

1. **What is Helm, and why do we need it?**

   As part of CNCF, Helm is the de-facto package manager for Kubernetes applications. To deploy a complex application, we need to deploy multiple underlying services. We might need to deploy database servers for easier testing and development. Having multiple individual manifests is possible. However, it is hard to deploy them as a single unit and sharing them is hard. A standard mechanism for passing configuration is also required. Helm provides all the preceding services, including a hub for discovery.

2. **What would you choose Helm or Kustomize?**

   I would avoid having to choose the two and would like to leverage open-source helm charts for standard applications and have the option to use Kustomize for in-house applications. Another approach is to store the output of Helm as input to Kustomize, which gives us the best of both worlds. I would go with the standard used by the enterprise and watch for places where Helm or Kustomize would be a better choice.

CHAPTER 14
Kubernetes Plugins

Introduction

The Kubernetes tools and extensions make the awesome Kubernetes functionality easy to use. In enterprise environments, running Kubernetes using the default settings is not sufficient. Kubernetes provides a way to replace key functionality such as:

- Authentication/Authorization
- Networking
- Storage
- Security
- Linking with existing enterprise systems using plugins.

For instance, by default, Kubernetes gives access to all the resources to anyone who has access to the cluster. In enterprises, you want access controlled by the enterprise's Active Directory or LDAP. It is just one example where plugins need to be used to use Kubernetes in an enterprise environment.

In this chapter, you will learn about the main categories of plugins in Kubernetes, popular plugins in those categories, and when you would use them.

Structure

In this chapter, we will discuss the following topics:

- Kubernetes plugins
- Service mesh
- Network plugins
- Storage plugins

- Security plugins
- Authorization plugins
- Custom resources/operators
- Managing VMs using Kubernetes

Objectives

When an interviewer asks you about service mesh, or how you would connect distributed storage or plugins, or the extensibility of Kubernetes, you should be able to answer this question clearly with examples and why you chose that plugin. If a plugin piques your interest, this chapter will give you hints on where to look further.

Non-goals

Each plugin deserves its own book, and unless you are being hired as a specialist in, say, Service Mesh, you are not expected to know the inner workings and settings of a plugin. This chapter does not provide details on how to get a plugin working or all the possible ways it can be configured/used.

After reading this chapter, you will be able to answer questions about the main categories of Kubernetes plugins, examples of them, and when/why you would use them. You will also gain exposure to a wide variety of plugins, and you will be able to make an intentional choice on which plugin you would like to become an expert in.

Plugins

All the functionality that an enterprise needs cannot be written by Kubernetes developers. Keeping that in mind, Kubernetes core developers have made a considerable effort to provide hooks at all Kubernetes key functionality so that they can be extended or, in some cases, replaced entirely.

The main categories are plugins are as follows:

- Service mesh
- Network plugins

- Storage plugins
- Security
- Custom resources/operators

Service mesh

Figure 14.1: *I lied*

Service mesh is implemented using network plugins, so it should be in the networking plugins section. Service meshes technically are not plugins, so I lied ☺.

Service meshes are essential in almost any enterprise that runs at scale. They provide a whole range of functionality and deserve to be a class on their own.

Why do you need a service mesh?

Kubernetes networking policy settings are limited in functionality. The network policy can, at best, say which resource can talk to which other resources (Pod A can talk to Pod B, and so on).

There is no way for:

- Enforcing all communications must be through SSL.
- Implementing a circuit breaker policy across the cluster.
- Providing secure connections between multiple clusters.
- Dynamically create and maintain policy-based access control.
- Traffic shaping (send 28% of traffic here and the rest over there).

Whenever one needs more than what the Kubernetes network policy provides, the Service Mesh option should be considered.

Istio

Istio (**https://istio.io/**), developed by Google and IBM in partnership with the Envoy team in Lyft, has emerged as the leader in service mesh solutions. Istio provides the following features (from: **https://istio.io**):

- Runnable on Kubernetes and bare metal.
- Standard, universal traffic management.
- Telemetry
- Security

Using Helm, you can install Istio by running the following:

```
# From: https://istio.io/latest/docs/setup/install/helm/
# Install the Helm client, version 3.6 or above.
# Configure the Helm repository:
helm repo add istio https://istio-
release.storage.googleapis.com/charts
helm repo update
# Installation steps
# Create a namespace istio-system for Istio components:
kubectl create namespace istio-system
# Install the Istio base chart, which contains cluster-wide
resources used by the Istio control plane:
#  When performing a revisioned installation, the base chart
requires the --defaultRevision value to
#  be set for resource validation to function. More information
on the --defaultRevision option can
#  be found in the Helm upgrade documentation.
```

```
helm install istio-base istio/base -n istio-system
# Install the Istio discovery chart which deploys the istiod
service:
helm install istiod istio/istiod -n istio-system --wait
```

The most compelling use case is using Istio to connect on-prem services to Kubernetes pods securely.

The following are the disadvantages of Istio:

- It is time-consuming to set up and implement.
- Istio is highly complex to configure and maintain.
- Since it is built out of multiple projects, the documentation might not be synchronized or up-to-date.
- Unless tan expert isavailable, it is very hard to customize to the enterprise's needs.

Honorable mentions

- **Open Service Mesh—https://openservicemesh.io/**—Open Service Mesh (OSM) is a lightweight and extensible cloud-native service mesh.
- **Linkerd**—**https://linkerd.io/2/reference/architecture/**—Second most popular service mesh. The Kubernetes-only implementation makes it simpler.
- **Consul Connect**—**https://www.consul.io/docs/internals/architecture.html**—Consul Connect is a Service Mesh solution from HashiCorp.

Network plugins

Kubernetes is compatible with **Container Network Interface** (**CNI**) specifications, and thus, has many network plugins. Network plugins provide advanced functionality such as:

- Connection to native network devices (for example, Cisco routers).
- Going beyond what iptables can provide.
- Cloud providers network appliance integration.

Cilium eBPF-based network plugin

Cilium (**https://cilium.io**) is a CNI-compatible network plugin that uses eBPF (**https://ebpf.io/what-is-ebpf/**) to provide high-performance and highly secure network connectivity. The features it provides beyond the default network plugin are as follows:

Cilium provides advanced features in the following areas:

- Networking
  - Service load balancing
  - Scalable Kubernetes CNI
  - Multi-cluster connectivity
- Observability
  - Identity-aware visibility
  - Advanced self-service observability
  - Network metrics + policy troubleshooting
- Security
  - Transparent encryption
  - Security forensic + audit
  - Advanced network policy

Example usage

Cilium can connect workloads across Kubernetes clusters using a single connectivity zone without requiring proxies. This enables simple, secure, and high-performance cross-cluster connectivity.

Storage plugins

Kubernetes provides **Physical Volume Claim** (**PVC**) to abstract the underlying storage. For them to work, storage plugins provide the implementation side of the interface.

Storage plugins provide connections to the following:

- Underlying cloud storage (AWS—EBS, Azure Storage—BLOB, AFS, and so on).
- Distributed storage such as Ceph (**https://docs.ceph.com/en/latest/#**) and Minio (**https://min.io/**).
- Storage hardware such as NetApp.

The following are the examples of storage plugins:

- Azure Disk
- Azure File
- AWS EBS
- GCE Persistent Disk
- GlusterFS
- Portworx
- Ceph

Ceph, which has its origin in RedHat, is a unified system that provides objects (like S3), blocks (like Hard disks), and file storage (like NFS). Ceph's block device can be used for Kubernetes PersistentVolumeClaims. The plugin makes it a very attractive option for an enterprise with a functioning large storage cluster with many nodes running Ceph. For Ceph-Kubernetes integration, see—**https://github.com/ceph/ceph-csi/**. For a tutorial on configuring Kubernetes to use Ceph, see—**https://docs.ceph.com/en/latest/rbd/rbd-kubernetes/**.

Security plugins

Authentication and authorization are two areas where enterprises have a wide variety of needs. Authentication Plugins allow enterprises to connect to the following:

- Active Directory
- LDAP
- Custom AuthN/AuthZ service

The authentication plugins are compiled into Kubernetes code and cannot be extended. The fully implemented Kubernetes built-in authentication plugins

are Static Token File, X.509 Client Certificate, and OpenID Connect token. If these work for you, then there is no need for a custom security plugin.

The framework authentication plugins allow you to inject authentication logic proxies that Kubernetes can use for authentication are Authenticating Proxy and Webhook Token. For a very nice introduction and example of LDAP authentication, see **https://itnext.io/implementing-ldap-authentication-for-kubernetes-732178ec2155**.

Authorization plugins

Whether an authenticated person is allowed access to resources is generally stored in a separate service in the enterprise. It also could be in the software itself. For example, salary information is available only to the employer and their management. You will not want anyone else that is not authorized to view your salary.

Kubernetes allows a pluggable interface to provide a custom authorization source.

Open Policy Agent (OPA)

Open Policy Agent (**OPA**) **https://www.openpolicyagent.org/** allows you to configure policy separately from code:

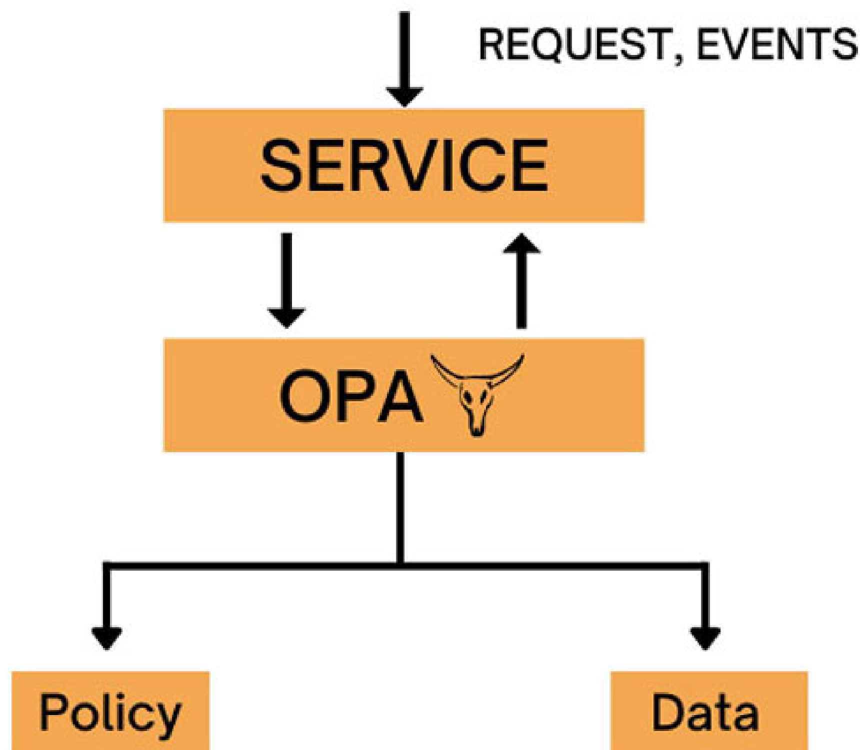


Figure 14.2: *OPA architecture*

OPA is a general policy engine that uses custom rego language. Rego language is very complex and has a very steep learning curve.

OPA is also very powerful and is widely adopted by companies as a policy engine for their entire infrastructure, not just Kubernetes. For Kubernetes-specific policy engine, see the Gatekeeper product described in the next section.

OPA has an extensive ecosystem that includes support beyond Kubernetes, such as Terraform, Spring Java apps, PHP, Linux SSH/sudo, and so on. For the full list, see **https://www.openpolicyagent.org/docs/latest/ecosystem/**. It also has an impressive list of adopters such as Atlassian, Intuit, and so on. See **https://github.com/open-policy-agent/opa/blob/main/ADOPTERS.md**.

The key benefit of OPA is keeping the policy separate from the application code. Its main use case is to provide centralized authorization for distributed applications, including microservices. You can try out their interactive

sample, which implements the following policy here—**https://www.openpolicyagent.org/docs/latest/#putting-it-together**.

The desired policy in English is as follows:

1. Servers reachable from the Internet must not expose the insecure "**`http`**" protocol.
2. Servers are not allowed to expose the "**`telnet`**" protocol.

The preceding policy is expressed in rego code:

```
package example
import future.keywords.every # "every" implies "in"
allow := true {                              # allow is true if…
count(violation) == 0                         # there are zero violations.
}
violation[server.id] {                          # a server is in the
violation set if…
some server in public_servers                    # it exists in the
'public_servers' set and…
"http" in server.protocols                      # it contains the insecure
"http" protocol.
}
violation[server.id] {                          # a server is in the
violation set if…
some server in input.servers                     # it exists in the
input.servers collection and…
"telnet" in server.protocols                     # it contains the
"telnet" protocol.
}
public_servers[server] {                         # a server exists in the
public_servers set if…
some server in input.servers                     # it exists in the
input.servers collection and…
some port in server.ports                       # it references a port in
the input.ports collection and…
some input_port in input.ports
port == input_port.id
some input_network in input.networks            # the port references
a network in the input.networks collection and…
```

```
input_port.network == input_network.id
input_network.public              # the network is public.
}
```

Gatekeeper

Gatekeeper uses OPA as its authorization engine, specifically customized for Kubernetes access control. It is used as Admission Controller and can be extended to other Kubernetes access control/policy implementations.

It also uses the custom rego language, which is not an easy language to learn.

Custom operators

Kubernetes operators are extensions that check the actual state of the resources in the cluster and the desired state. Kubernetes operators are an integral part of Kubernetes extensions that are used by Kubernetes itself to manage deployments, services, and so on.

Kubernetes allow custom extensions at the operator level also. This will enable one to have declarative code to perform custom operations. **https://operatorhub.io/** has a searchable list of operators.

Here are some examples of custom operators in CI/CD (**https://operatorhub.io/?category=Integration+%26+Delivery**):

- ArgoCD—**https://operatorhub.io/operator/argocd-operator**
- Flux—**https://operatorhub.io/operator/flux**
- GitLab—**https://operatorhub.io/operator/gitlab-operator-kubernetes**
- Jenkins—**https://operatorhub.io/operator/jenkins-operator**

VM management from Kubernetes using KubeVirt

Installing KubeVirt (**https://github.com/kubevirt/kubevirt**) VM management allows us to use Kubernetes to launch VMs like any other node in the cluster. This is useful for running jobs that require an entire VM.

AWS Fargate

AWS Fargate plugin allows us to use the Kubernetes interface to launch serverless computers for containers. This allows elastic scaling without requiring the use of dedicated worker nodes. For more details on ECS versus EKS versus Fargate see **https://sysdig.com/learn-cloud-native/kubernetes-security/aws-eks-with-and-without-fargate-understanding-the-differences/**.

Another interesting use case is launching highly secure Firecracker (**https://firecracker-microvm.github.io/#home**)VMs using the familiar Kubernetes interface. Although Fargate service is specific to AWS, you can try Firecracker VMs on your machine (Intel x86_64 processors only as of now) or in your on-prem data center. See **https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/getting-started.md**.

Conclusion

The default options provided by Kubernetes are insufficient for production use due to security and performance reasons. Kubernetes is built for extensibility, and multiple plugins and extensions are available to satisfy production needs. The plugins are mainly in the network and storage areas. Service Mesh and Policy Agents allow advanced features such as connection to external clusters, observability, authentication, and authorization. Custom operators and associated custom resources will enable the extension of Kubernetes in various areas, as listed at **https://operatorhub.io/**. VMs can also be controlled using the same declarative manifests using custom extensions.

This chapter briefly went through a very comprehensive set of plugins/extensions with links. The reader is encouraged to dig deep into the areas of their interest.

In the upcoming chapter, we will review typical interview questions across all chapters.

Points to remember

- Plugins (for the full list, see **https://landscape.cncf.io/**)allow Kubernetes to be extended for critical functionalities such as authentication, authorization, networking, storage, security, and custom integrations with enterprise systems.

- Service Mesh allows advanced policy enforcement for applications deployed in enterprise Kubernetes clusters. Istio is the most famous among many.
- Cilium and eBPF-based network plugins offer service mesh-like features without compromising on performance.
- Storage plugins abstract underlying storage such as Azure Disk, AWS EBS, and GCE Persistent Disk.
- OPA and Gatekeeper allow separation of authorization/authentication policy from application code.
- Custom operators are another mechanism to make Kubernetes easy to use for developers.
- VM management (instead of pods) is also possible with KubeVirt. You get the same Kubernetes interface for managing VMs.
- Multiple ways that Kubernetes can be extended allow enterprises to make DevX better.

Interview questions and answers

1. **Why do we need Plugins?**

   Plugins enable us to extend Kubernetes functionality and, in some cases, even replace certain parts like Network Management. Various plugins extend Kubernetes functionality for integration with enterprise and cloud provider systems.

2. **Why do we need a Service Mesh?**

   Whenever one needs more than what the Kubernetes network policy provides, the Service Mesh option should be considered. Examples would be: Enforcing all communications must be through SSL, implementing a circuit breaker policy across the cluster, providing secure connections between multiple clusters, dynamically creating and maintaining policy-based access control, and traffic shaping (send 28% of traffic here and the rest over there).

3. **What is OPA, and how is it different from GateKeeper?**

   OPA is a general policy engine that uses custom rego language.

Gatekeeper uses OPA as its authorization engine. It is specifically customized for Kubernetes access control. It is used as Admission Controller and can be extended to other Kubernetes access control/policy implementations.

4. **Can Kubernetes be used to manage VMs?**

   Yes, it can use plugins like KubeVirt.

CHAPTER 15
Kubernetes Questions

Introduction

In this chapter, we have all the Kubernetes interview questions and answers consolidated for easy reference.

Structure

In this chapter, we will review Kubernetes interview questions and answers.

Objectives

You will be ready for the Kubernetes-focused interview questions by brushing up on this chapter. The answers to the Kubernetes interview question we have seen so far will be at the top of your mind.

Interview questions and answers

The following is a consolidated list of all the questions and answers in the Kubernetes-focused chapters. The following figure shows the Kubernetes troubleshooting steps:

KUBERNETES TROUBLESHOOTING FLOWCHART

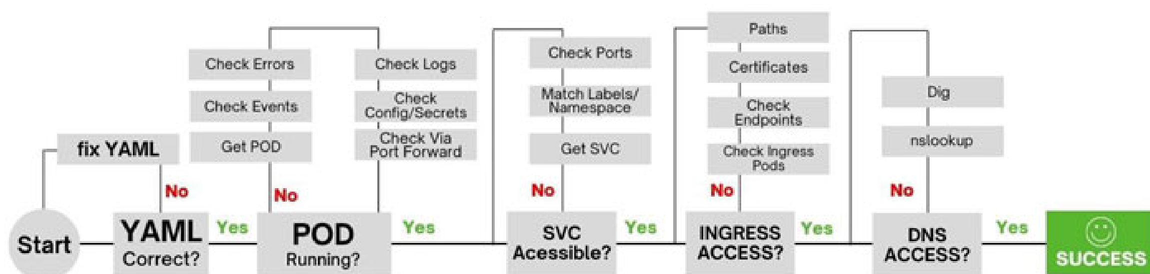


Figure 15.1: *Kubernetes troubleshooting flowchart*

%

Chapter 10: Kubernetes on Various Platforms (Windows, Azure, AWS, and Google Cloud)

1. **Why do we need Managed Kubernetes when we can launch our own easily?**

   Running Kubernetes in production is hard. Setting up and testing on a continuous basis, HA requires a dedicated Kubernetes team. Upgrading and applying security patches requires expertise. Support for Kubernetes and kernel issues is extremely difficult to find. Unless it is a very large tech enterprise, it takes the teams' focus away from delivering their unique value to customers.

2. **Does Kubernetes run on Windows?**

   Yes, but with many limitations. The key ones are that the windows server cannot be the primary server, you cannot run Linux containers on the Windows server, and privileged containers are not allowed.

3. **What are the differences between AKS, EKS, and GKE?**

   For most practical purposes, they are the same except for costing and support for advanced features. AKS does not charge for the primary servers, whereas EKS, GCP charge for them. They all offer the ability to hook access control to their identity access management (IAM) system. GKE offers built-in support for Istio service mesh and an autopilot mode where GKE does all the updates for the worker nodes.

%

Chapter 11: Kubernetes performance optimization

1. **How do you approach performance issues in Kubernetes?**

   I approach them as solving any process performance issues, as running containers are processes. Kubernetes API server and etcd servers are also processed. Based on the information available and the symptoms, I create a hypothesis and check the resource usage using the Utilization, Saturation, and Errors methodology. If my hypothesis checks out, I apply the fixes for the short term to mitigate the issue. Refer to the following figure:

   Debugging Strategy:

   1. Have a hypothesis.
   2. Collect data that disproves it.
   3. Depending on the analysis:
      - if your hypothesis is correct fix it.
      - if not start at step (1)

   Figure 15.2: *Debugging strategy*

   I participate in retro meetings to figure out how the issue can be prevented from happening again.

2. **What will you do if certain Pods/Namespaces/Workloads use up more than their fair share of the resources?**

   I will review the CPU/Memory limits placed on pods/namespaces/workloads and see if fixing them helps. If certain workloads require more resources, I will move them to nodes with larger

resources using node tainting/node affinity. In rare cases, I will review if setting pod priorities via `PriorityClass` would help.

%

Chapter 12: Kubernetes troubleshooting tips

3. **How do you troubleshoot Kubernetes issues?**

   I have a structured approach to troubleshooting issues. Even though it might seem like a waste of time, this ensures that we have covered the bases and not gone after red herrings. The structured approach starts from the bottom of the heap, Pods, and goes up to DNS.

4. **I have an issue with a Service; how will you troubleshoot it?**

   After making sure that Pods are ok, I will check the Service configuration to ensure that the namespace and matching labels are correct. I will make sure that the IP/port of the service matches the expected IP/port. Using curl or Netcat, check that the service responds on the IP/port. I will start a diagnosis pod in the same namespace to ensure that I can reach the underlying Pod IPs/Ports plus that the Service IP/port is reachable.

%

Chapter 13: Kubernetes tools and extensions

5. **What is Helm, and why do we need it?**

   Helm is the *de facto* package manager for Kubernetes applications. To deploy a complex application, we need to deploy multiple underlying services. We might need to deploy database servers for easier testing and development. Having multiple individual manifests is possible. However, deploying them as a single unit and sharing them is hard. A standard mechanism for passing configuration is also required. Helm provides all the preceding services, including a hub for discovery.

6. **Would you choose Helm or Kustomize?**

   I would avoid choosing between the two and would like to leverage open-source helm charts for standard applications and have the option to use Kustomize for in-house applications. Another approach is to store the output of Helm as input to Kustomize, which gives us the best of both worlds. I would go with the standard used by the enterprise and watch for places where Helm or Kustomize would be a better choice.

%

Chapter 14: Kubernetes plugins

7. **Why do we need Plugins?**

   Plugins enable us to extend Kubernetes functionality and, in some cases, even replace certain parts like Network Management. Various plugins extend Kubernetes functionality for integration with enterprise and cloud provider systems.

8. **Why do we need a Service Mesh?**

   Whenever one needs more than the Kubernetes network policy provides, the Service Mesh option should be considered. Examples would be: Enforcing all communications must be through SSL, implementing a circuit breaker policy across the cluster, providing secure connections between multiple clusters., dynamically creating and maintaining policy-based access control, and traffic shaping (send 28% of traffic here and the rest over there).

9. **What is OPA, and how is it different from GateKeeper?**

   OPA is a general policy engine that uses custom rego language.

   Gatekeeper uses OPA as its authorization engine. It is specifically customized for Kubernetes access control. It is used as Admission Controller and can be extended to other Kubernetes access control/policy implementations.

10. **Can Kubernetes be used to manage VMs?**

    Yes, it can use plugins like KubeVirt.

Conclusion

We have come a long way, from the importance of culture and mindset to nitty-gritty details on Kubernetes. We performed multiple hands-on Kubernetes experiments using free tools such as **https://www.katacoda.com/**. Tools such as Helm, Kustomize, and advanced extensions such as Istio were also looked at.

Keep practicing, and you will rock the interview.

Index

A

attachment
  advantages 114
authorization plugins 203
AWS Elastic Kubernetes Service (EKS) 152
AWS Fargate 206
AWS Identity and Access Management (IAM) 152
Azure Kubernetes Service (AKS) 151

B

Balena 154
block devices
  limitations 114
block storage 113
blue-green deployment 98-101
  versus, canary deployment 103

C

canary deployment 101-103
  versus, blue-green deployment 103
career paths, DevOps/SRE/Kubernetes 2
  individual contributor (IC) path 3
  management path 3, 4
  selecting 5-7
central processing unit (CPU) 34
Certified Kubernetes Administrator (CKA) 13
Certified Kubernetes Application Developer (CKAD) 13
Cgroups 50, 51
Cilium eBPF-based network plugin 201, 202
Cloud Native Computing Foundation (CNCF) 13
CNCF landscape 13
Common Internet File Systems (CIFS) 114
computer architecture 34
computer performance 158
computer resources
  CPU/GPU 158
  memory 159
  network 159
  storage 159
computers components 30
  process 33

Profesora 31, 32
storage 32
workspace 33
ConfigMap 119, 120
secrets 120-125
Consul Connect 201
consumer expectations 95
availability, calculating 95-97
business cost 95
container images
managing 66
versus, running container instances 64, 65
container networking 52-54
Container Network Interface (CNI) 109, 201
container orchestration
need for 72-74
container registry 52
containers 51
pushing 66
removing 65
running 56, 57
runtime configuration 57
stopping 65
container storage 55, 56
culture fit
key elements 22
custom operators 206
custom container image
building 58-60
corollary 61
layers 60, 61
multi-stage builds 62-64
using, for Kubernetes 61

D

deployment object 79
rollout strategy 80
version deduction 80, 81
DevOps culture
benefits 21, 22
DevOps/SRE 1
career paths 2
demonstration 25
DevOps/SRE culture 18-20
benefits 22
key elements 20, 21
DevOps/SRE jobs 134
docker-based virtualization
disadvantages 49, 50

Docker Swarm 74

E

ENV variables 57, 58

G

Gatekeeper 206
GitOps 125-129
Google Kubernetes Engine (GKE) 152, 153

H

hard drives 114
Helm 186
  need for 186
  versus Kustomize 190-192
Helmfile 194
Helm repo 186
Helmsman 192, 193
hybrid-cloud 12, 13

I

Individual Contributor (IC) path 134
Ingress controllers 84-86
  health checks 86-88
  physical volumes 88
Ingress troubleshooting 178
  definition 178, 179
  DNS issues 181
  endpoints, confirming 180
  flowchart 181
  information, obtaining 178
  Ingress logs 180, 181
  Ingress Pod 180
  worst case scenario 181, 182
Istio 200
  disadvantages 201

K

K9s 194
key elements, culture fit
  collaborations 23
  continuous improvement 22
  intelligent risks 23
  learning 22

relentless focus, on customer 24
kubectl top command 160
Kubernetes 11, 74
applicability 43
basics 71
career paths 2
ecosystem challenges 13, 14
interview questions and answers 209-212
on Raspberry PI 154
performance optimizations 162
performance tools 160
plugins 197, 198
Windows support 150
Kubernetes Certified Service Provider (KCSP) 13
Kubernetes dashboard 161
Kubernetes deployments 97
blue-green deployment 98-101
canary deployment 101-103
Kubernetes/DevOps career map 134-136
Kubernetes ecosystem 195
Kubernetes' inclusive community 8
inclusiveness 8
Women/PoC, in DevOps/SRE 8
Kubernetes primitives 74
DaemonSets 81, 82
deployments 79
Ingress controllers 84
Persistent volume claims (PVC) 89, 90
Pods 75
Replica sets 78, 79
services 82, 83
Kubernetes troubleshooting flowchart 182
Kubernetes troubleshooting mental model 170
Ingress, troubleshooting 178
pods, troubleshooting 171
services, troubleshooting 175
YAML problems 171
KubeVirt 206
Kustomize 126, 189

L

Linkerd 201
LinuX Container runtime (LXC) 49

M

Mesos 74

N

Network-Attached Storage (NAS) 114
  limitations 115
Network File System (NFS) 114
networking 38-41
  exercise 41, 42
networking policy 109-113
network plugins 108, 201
node management 116-119

O

object storage 115
Open Policy Agent (OPA) 204, 205
Open Service Mesh (OSM) 201
operating system 35-38
operating system fundamentals 136
  containers 137
  deployments, with Kubernetes 138, 139
  GitOps 140
  Kubernetes basics and primitives 138
  Kubernetes services 139
Oslo 37

P

performance optimizations, Kubernetes
  cluster, moving closer to customer 165
  container image size 162
  features, configuring 164, 165
  minimal host, switching 165
  namespace-based resource limits 163
  node tainting/node affinity 163, 164
  pod priorities 164
  Pod resource limits 163
  resources for worker nodes, increasing 164
  storage, configuring 165
Physical Volume Claim (PVC) 115, 116, 202
plugins 198
Pods
  declarative specification 75
  specification 76, 77
Pods troubleshooting
  configuration 174
  double-check, by using events 173
  flow chart summary 175
  information, obtaining 171, 172
  log 173
  Pod definition 172

storage configuration 174
verify, using exec into Pod 174
Portable Operating System Interface (POSIX) 37
process 37, 48
advantages 49

R

Random Access Memory (RAM) 29, 34
Raspberry PI 153, 154
Redundant Array of Independent Disks (RAID) 113

S

security plugins 203
service mesh 199
need for 199
services troubleshooting 175
container port, matching target port 178
information, obtaining 175, 176
service definition 176, 177
service definition labels 177
Solid-State Disks [SSDs] 34
storage 113
storage plugins 202, 203

T

time to live (TTL) 42
Tokyo 37
Tom’s Obvious, Minimal Language (TOML) 193
total cost of ownership (TCO) 48

U

Utilization Saturation and Errors (USE) methodology 159, 160

V

VM management 206

W

WordPress
installing 187, 188
worker nodes 116