The KCNA Book

nigelpoulton

The KCNA Book

Pass the Kubernetes and Cloud Native Associate Exam in style

Nigel Poulton

Table of Contents

Getting started
- Who is the book for
- How is the book organised
- About the author
- Feedback

1: Setting the scene
- Virtualisation
- Containerisation
- Monolithic vs microservices
- Chapter summary
- Exam essentials
- Recap questions

2: Cloud native architecture
- Defining cloud native architecture
- Resiliency
- Autoscaling
- Serverless
- Community and governance
- Roles and personas
- Open standards
- Chapter summary
- Exam essentials
- Recap questions

3: Container orchestration
- Primer
- Container runtimes
- Container orchestration fundamentals
- Container security
- Container networking

Service meshes
Container storage
Chapter summary
Exam essentials
Recap Questions

4: Kubernetes Fundamentals

Primer
Simple Kubernetes workflow
Containers and pods
Augmenting pods
Kubernetes architecture
Scheduling
Kubernetes namespaces
The Kubernetes API and API server
Kubernetes networking
Chapter summary
Exam essentials
Recap questions

5: Cloud native application delivery

Primer
CI/CD
GitOps
Chapter summary
Exam essentials
Recap questions

6: Cloud native observability

Primer
Telemetry and observability
Prometheus
Cost management
Chapter summary
Exam essentials
Recap questions

The exam
- Exam domains and competencies
- About the exam
- Booking the exam
- Taking the exam
- Getting your result
- Staying connected

8: Sample test

Appendix A: Chapter quiz answers
- Chapter 1: Setting the scene
- Chapter 2: Cloud native architecture
- Chapter 3: Container orchestration
- Chapter 4: Kubernetes fundamentals
- Chapter 5: Cloud native application delivery
- Chapter 6: Cloud native observability

Appendix B: Sample Test answers

What next
- Other exams
- Books
- Video courses
- Let’s connect

Getting started

Kubernetes and cloud native technologies are all the rage and are shaping the world we work in. Building apps as small, specialised, single-purpose services that can self-heal, autoscale, and be regularly updated without downtime brings huge benefits. However, possessing the knowledge and skills to leverage these technologies is a huge career boost for you as an individual. For example, knowing how to design, build, and troubleshoot cloud native microservices applications running on Kubernetes can get you the best roles, on the best projects, at the best organisations. It can even earn you more money.

With all of this in mind, the Cloud Native Computing Foundation (CNCF) designed the KCNA exam and certification as a way for you to prove your competence with these technologies.

This book covers all of the exam objectives in one place in a well-organised and concise format. It's useful as both a revision guide and a place to start learning new technologies and concepts. For example, if you already know the basics of Kubernetes, the book will reinforce everything you already know, as well as test your knowledge with its extensive quizzes and explanations. However, if you're new to any of the topics on the exam, the book will get you up-to-speed quickly.

Who is the book for

The book is for anyone wanting to gain the KCNA certification.

As the exam tests your understanding of core technologies and concepts, it’s applicable to anyone working in technology. Examples include:

- Architects
- Management
- Technical marketing
- Developers
- Operations
- DevOps, DevSecOps, CloudOps, SREs etc.
- Data engineers
- More…

The book and exam are particularly useful if you come from a traditional IT background and want to learn the fundamentals of Kubernetes and cloud native.

If you’re brand new to Kubernetes, you should consider reading *Quick Start Kubernetes.* It’s only 100 pages long and will get you up-to-speed and 100% comfortable with the fundamentals of Kubernetes. It also has very easy hands-on examples that really help you grasp some of the concepts that might be new to you. It’s available on Amazon and Leanpub, and published in several languages, including French, Italian, Portuguese, Russian, Simplified Chinese, Spanish, and more about to be released.

How is the book organised

The technical content of the book is organised with one chapter per exam domain. There's a chapter dedicated to preparing you to take the exam, and there's a **full practice exam** with 60 questions just like the real exam.

Each technical chapter is organised as follows:

- Technical content
- Chapter summary
- Exam essentials
- Practice questions

The *exam essentials* is a recap of the major topics learned and can be used like flashcards when doing final revision and exam prep.

The practice questions test your mastery of the topics learned and are a similar style to the questions in the exam. However, they are not actual exam questions.

The chapter dedicated to preparing for, and taking the exam, explains exactly what it's like taking the exam so you don't have any surprises on the day.

The practice exam is a great opportunity for you to test your readiness for the real exam. Again, the questions are *like* the questions you'll see in the exam, but they're not actual exam questions.

About the author

OK, so I'm Nigel and I've trained over one million people on cloud and container technologies. I've authored several best-selling books and video training courses and dedicated my working life to helping people take their first steps with containers and Kubernetes. I'm also passionate about explaining things as clearly as possible so that you love my books and videos.

I actually wrote this entire book in draft form, took the exam, then came back and finely tuned the content to better prepare you for the exam. I considered failing the exam and re-taking it multiple times to get an even better feel for the style of questions and levels of detail being tested. However, that felt wrong and I wanted to get the book into your hands as quickly as possible.

I'd love to connect and would love to hear about your exam experience. You can reach me at all of the following:

- LinkedIn: https://www.linkedin.com/in/nigelpoulton/
- Web: nigelpoulton.com
- Twitter: @nigelpoulton

Feedback

Writing books is hard, and I worked tirelessly over holiday periods to get this book into your hands. With this in mind, I'd consider it a personal favor if you took a minute or two to write an Amazon review.

Also, if you have any feedback on the book, or your exam experience, ping me an email at `kcnabook@nigelpoulton.com` and I'll do my best to respond.

Enjoy the book, and good luck with your exam!

1: Setting the scene

This chapter doesn’t map directly to an exam objective. However, the things you’ll learn are in the exam and are important in setting the scene for why we have technologies like containers and Kubernetes. If you already know this, you can skip to the next chapter. Otherwise, stick around while we set the scene for the rest of the book.

We’ll cover all of the following at a high level.

- Virtualisation
- Containerisation
- The transition from monolithic apps to microservices

Don’t worry if you think we’re not covering things in enough detail. This is just an introductory chapter and we’ll get into the detail in later chapters.

Virtualisation

In the distant past we deployed one application per physical server. This was a huge waste of physical resources and company capital. It also delayed the rollout of applications while physical servers were procured, racked, patched into the network, and had an operating system installed.

Virtualisation technologies like VMware came along and opened the door for us to run multiple applications on a single physical server. This meant we didn’t have to buy a new server for every new app, we could deploy apps very quickly to virtual machines on existing servers and avoid all of the following:

- No more waiting for finance to approve server purchases
- No more waiting for the datacenter team to rack and cable servers
- No more waiting for the network team to authorise servers on the network
- No more waiting for sysadmins to install operating systems

Almost immediately we went from wasting money on over-powered physical servers that took ages to purchase and install… to a world where we could quickly provision virtual machines on existing servers.

However, the industry never sleeps, and innovation never stops.

Containerisation

In the early 2010's Docker gave the world the gift of easy-to-use containers.

At a high level, containers are another form of virtualisation that allow us to run even more apps on less servers and deploy them even faster.

Figure 1.1 shows a side-by-side comparison of server virtualisation and container virtualisation.

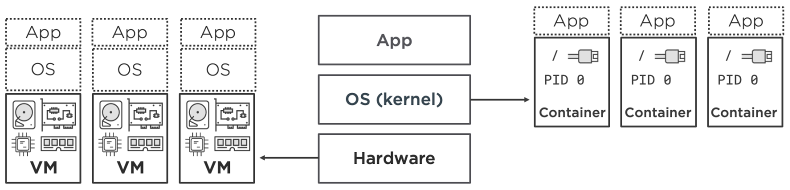

Figure 1.1

As the image shows, server virtualisation slices a physical server into multiple virtual machines (VM). Each VM looks, smells, and feels like a physical server, meaning each one has virtual CPUs, virtual memory, virtual hard drives, and virtual network cards. You install an operating system (OS) on each one and then install one app per VM. If a single physical server is sliced into 10 virtual machines, there will be 10 operating systems and you can install 10 apps.

Container virtualisation slices operating systems into virtual operating systems called *containers*. Each container looks, smells, and feels like a normal OS. This means each container has its own process tree, root filesystem, eth0 interface and more. You then run one app per container, meaning if a single server and OS is sliced into 50 containers, you can run 50 apps.

That's the view from 40K feet.

Containers vs VMs

Containers and virtual machines are both virtual constructs for running applications. However, containers have several important advantages.

One advantage is that containers are a lot more lightweight and efficient than virtual machines. This means businesses using containers can run even more apps on the same number of physical servers.

As an example, an organisation with 10 physical servers might be able to run 100 virtual machines and apps. However, if the same organisation chose containers instead of virtual machines, they might be able to run 500 containers and apps. One of the reasons is that every VM needs its own dedicated operating system (OS). This means a single physical server sliced in to 10 VMs requires 10 installations of Windows or Linux (other operating systems exist). Each OS consumes CPU, memory and disk space that can't be used for apps. In the container model every container shares the OS of the host it’s running on. This means there’s only one OS consuming CPU, memory, and disk space, resulting in more resources being available to run applications.

Containers are also faster to deploy and start than VMs. This is also because every VM contains an OS and an application. Operating systems can be large, making them bigger and bulkier to package. It also means that starting a VM bootstraps a full OS before the app can start. This can be time consuming.

To recap, when packaging an application as a container, you only package the application and dependencies. You do not package a full OS. This makes container images smaller and easier to share. It also makes them faster to start – you only start the app.

The following might help if the above is a little unclear.

Application developers write applications as they always have. The application and dependencies are then packaged into a container image. *Dependencies* are things like shared library files. Once the container image is created, you can host it in a shared repository where it can be accessed by the required systems and teams. A container host, which is just a server running a container runtime such as Docker, grabs a copy of the image and starts a container from it. The container host has a single copy of Windows or Linux that is already up and running, and the container runtime on the host quickly creates a container and executes the app that’s inside the image.

The smaller packaging used by the container model enables other benefits such as microservices, automated pipelines and more. We’ll cover all of these later in the book.

Before moving on, it’s important to acknowledge an advantage VMs have over containers.

The fact that every VM requires a dedicated OS is a disadvantage when it comes to packaging and application start times. However, it can be an advantage when it comes to security. As a quick example, if the shared OS on a container host gets compromised, every container is also compromised. This is because every container shares the OS kernel of the container host. In the VM model, every VM has its own OS kernel, this means compromising one kernel has no impact on other VMs.

Despite this security challenge, containers are generally considered the best solution for modern business applications.

So far, we've focussed mainly on physical infrastructure, such as servers, and how best to utilise them. Now let's change focus onto application development and management.

Monolithic vs microservices

In the past, we built monolithic applications that ran on dedicated physical servers.

Monolithic application is jargon for a large complicated application that does lots of things. For example, a monolithic application may have all of the following services bundled and shipped as a single program.

- Web front-end
- Authentication
- Shopping basket
- Catalog
- Persistent store
- Reporting

The important thing to understand is that all of these services were developed by a single team, shipped as a single program, installed as a single program, and patched and updated as a single program. This meant they were complex and difficult to work with. For example, patching, updating, or scaling the reporting service of a monolithic app meant you had to patch, update, or scale the entire app. This made almost all changes high-risk, often resulting in updates being rolled up into a single very high-risk update performed once a year over a long stressful weekend.

While this model was OK in the past, it's not OK in the modern cloud era where businesses are entirely reliant on technology and need their applications to react in-line with fast changing markets and situations.

Enter microservices applications…

The modern answer to the problem of monolithic applications is *cloud native microservices* apps.

At the highest level, "cloud native" and "microservices" are two separate things. *Microservices* is an architecture for designing and developing applications, and *cloud native* is a set of application features. We'll get into detail soon.

Consider the same application with the web front-end, authentication, shopping basket, catalog, persistent store, and reporting requirements. To make this a microservices app you develop each feature independently, ship each one independently, install each one independently, patch, update, and scale each one independently. However, they all communicate and work together to provide the same overall application experience for users and clients.

We call each of these features a "service", and as each one is small, we call them microservices (*micro* is another word for small in English). With this in mind, the same application will have the following 6 microservices, each of which is its own small independent application.

- Web front-end
- Authentication
- Shopping basket
- Catalog
- Persistent store
- Reporting

This microservices architecture changes a lot of things, including:

- Each microservice can have its own small and agile development team
- Each microservice can have its own release and update cycle
- You can scale any microservice without impacting others
- You can patch and update any microservice without impacting others

Generally speaking, adopting microservices architectures means you can deploy, patch, update, scale, and self-heal your business applications a lot easier and a lot more frequently than you can with monolithic apps – this is the cloud native element. Many organisations using microservices applications push multiple safe and reliable updates per day.

However, the distributed nature of microservices apps brings some challenges.

For example, the individual microservices of an application can all run on different servers and virtual machines. This means they need to communicate over the network, increasing network complexity as well as introducing security risks. It also becomes vital for the development teams of different microservices to communicate with each other and be aware of the overall app. That all said, the benefits of the microservices architecture far outweigh the challenges it brings.

In quick summary, *microservices* is the architecture pattern where you split application features into their own small, single-purpose apps. *Cloud native* is a set of features and behaviors. The *features* including self-healing, autoscaling, zero-downtime updates and more. The *behaviors* include agile development, and frequent predictable releases.

Chapter summary

In this chapter, you learned that we used to deploy one application per physical server. This was wasteful of capital, servers, and environmental resources. It also caused long delays in application rollouts while new servers had to be procured, delivered, and installed. VMware came along and let us run multiple applications per physical server. It reduced capital expenditure and allowed more efficient use of server and environmental resources. It also allowed us to ship applications a lot faster by deploying them to virtual machines on servers we already owned.

Containers are also a form of virtualisation. They virtualise at the operating system layer and each container is a virtual operating system. Containers are faster and more efficient than virtual machines, however, out-of-the-box they’re usually less secure. The advantages of containers made it possible for us to re-think the way we develop, deploy, and manage applications.

A major innovation, enabled by containers, is the microservices design pattern where large complex applications are broken up and each application feature is developed and deployed as its own small, single-purpose, application called a microservice. This architecture enables cloud native features such as, self-healing, scaling, regular and repeatable rollouts, and more.

All of this can be backed by cloud and cloud-like infrastructure that can scale on-demand.

Today, we live in a world where applications and infrastructure can self-heal, and individual application features (microservices) can automatically scale and be updated without impacting any other application features.

Exam essentials

This chapter doesn't map directly to an exam domain. However, the following exam topics were mentioned and will be covered in more detail later in the book.

Container runtimes

A server that runs containers is called a *container host*. They use a low-level tool called a *container runtime* to start and stop containers. Docker is the best-known container runtime and was the first container runtime supported by Kubernetes. However, it is being replaced in Kubernetes by a lighter-weight version called *containerd* (pronounced "container dee"). Many other container runtimes exist and some of them work differently. Some offer better performance at the expense of security, whereas others offer better security at the expense of size and performance. You'll learn more later in the book.

Container security

All containers running on a single host share the host's OS kernel. This makes them small, portable, and fast to start. However, if the host's kernel is compromised, all containers running on that host are also compromised. There are many container runtimes available, and not all work the same. For example, the Docker and containerd runtimes implement the shared kernel model making them lightweight and fast but susceptible to shared kernel attacks. Whereas gVisor implements a more secure model. You'll learn more later in the book.

Container networking

Microservices architectures implement application features as small standalone services called microservices. The different microservices making up an app have to use the network to communicate and share data. This increases the number of network entities, increases the complexity of networking in general, and increases network traffic. Communicating and sharing data across the network is also slower than communicating and sharing on the same server.

Autoscaling

Microservices architectures allow individual application microservices to scale independently. For example, platforms like Kubernetes can automatically scale up the reporting microservice of an application at month-end and year-end when they're busier than normal. Again, you'll learn more later in the book.

Recap questions

See Appendix A for answers. Page 157 in the paperback edition.

1. Which of the following are advantages of containers compared to virtual machines? Choose all correct answers.

- A. Smaller size
- B. Faster start times
- C. More secure out-of-the-box
- D. More apps per physical server

2. Which of the following is an advantage of virtual machines compared to containers?

- A. Virtual machines start faster than containers
- B. Virtual machines are more secure out-of-the-box
- C. Virtual machines are smaller than containers
- D. Virtual machines enable microservices design patterns

3. Which layer does container virtualisation work at?

- A. The hardware layer
- B. The infrastructure layer
- C. The application layer

- D. The operating system layer

4. Which of the following are components of a container? Choose all correct answers.

- A. Virtual process tree
- B. Virtual CPUs
- C. Virtual hard disks
- D. Virtual filesystems

5. Which of the following is true?

- A. You can normally run more containers than virtual machines on a server
- B. You can normally run more VMs than containers on a server

6. Why are containers potentially less secure than VMs?

- A. Container networking is always in plain text
- B. Container images are always available on the public internet
- C. You cannot encrypt container filesystems
- D. All containers on a host share the same OS kernel

7. Which of the following are part of a container image? Choose all correct answers.

- A. Application code
- B. Application dependencies
- C. Application snapshots
- D. A dedicated operating system

8. Which of the following best describes a monolithic application?

- A. An application that only runs on on-premises datacenters

- B. An application with all features coded in a single binary
- C. An application that implements cloud native features such as autoscaling and rolling updates
- D. An application with very small container images

9. Which of the following are disadvantages of monolithic applications? Choose all correct answers.

- A. Every feature has to communicate over the network
- B. Updates are high risk and complex
- C. You cannot scale individual application features
- D. You cannot deploy them to the public cloud

10. Which of the following are advantages of microservices applications? Choose all correct answers.

- A. Each microservice can have its own small agile development team
- B. Each microservice can have its own dedicated OS kernel
- C. Each microservice can be scaled independently
- D. Each microservice can be patched independently

11. Which of the following are potential disadvantages of microservices applications? Choose all correct answers.

- A. Container images can be very small
- B. They cannot run on public clouds
- C. Increased networking complexity
- D. Increased network traffic

2: Cloud native architecture

In this chapter, you'll learn everything needed to pass the *Cloud Native Architecture* section of the exam. These topics account for 16% of the exam and provide a foundation for the topics covered in later chapters.

The chapter is divided as follows.

- Defining cloud native architecture
- Resiliency
- Autoscaling
- Serverless
- Community and governance
- Roles and personas
- Open standards
- Chapter summary
- Exam essentials
- Recap questions

Defining cloud native architecture

The first thing to understand is that *cloud native* is a set of capabilities and practices. That's right, cloud native isn't about running on the public cloud, it's a set of capabilities and practices.

Cloud native capabilities include applications and infrastructure that is resilient, scalable, observable, highly automated, and easily updated. *Cloud native practices* is about building teams and processes that encourage co-operation, integration, and open governance. The end result is an environment that delivers high-quality applications that meet and respond to modern business demands.

Most cloud native apps are also *microservices apps*. As explained in the previous chapter, microservices is an architecture where every application feature is developed and deployed as a small independent application. As a simple example, an application with a web front-end and a database back-end will have two microservices – one for the front-end and one for the back-end. The most common model is to deploy each microservice as its own set of containers. For example, your initial deployment might have one container for the front-end and a different container for the back-end. If you need to scale-up the front-end, you add more identical front-end containers.

To avoid confusion, *microservices* is the application architecture such as developing each application feature as its own small app. *Cloud native* is the features such as resiliency, scalability, rollouts, etc. The two work together very well.

While it's true that cloud native isn't just running apps in the cloud, it's also true that many of the concepts originated in the public cloud and work best there. However, it's possible to be *cloud native* in your own on-premises datacenter, and doing so will bring many benefits to your business. For example, on-premises infrastructure and applications can be observable, automated, and even have elements of resiliency and scalability. They will also be easier to migrate to the cloud and even burst into the cloud. However, they'll always struggle to compete against the theoretically infinite resources of the public cloud.

Let's quickly clarify a few key terms. Anything not clarified here will be covered in detail in the following sections.

Public cloud

The *public cloud* is on-demand pay-as-you-use infrastructure and services offered by a 3rd-party. Popular examples include Amazon Web Services, Microsoft Azure, Google Cloud Platform, and Digital Ocean. The infrastructure and services are shared with other users and you don’t own or manage them. You just consume them and pay for what you use. Most cloud platforms are so large that we sometimes say they have infinite capacity.

On-premises datacenter

On-premises, on-prem and *datacenter* are all terms that refer to your own dedicated infrastructure and services that you own and manage. You may even own the facility that houses them, but this is not a requirement. They are usually very small compared to public clouds.

Infrastructure

Infrastructure is things such as servers, virtual machines, cloud instances, networks, and storage.

Cloud bursting

Cloud bursting is when you max-out your datacenter capacity and use the cloud as an overflow area. For example, an online retail app might max out on-prem datacenter capacity over a holiday weekend and scale into the public cloud to meet the temporary peak in demand. Users of the app may have requests serviced by instances running in your on-premises datacenter or the public cloud and should not know the difference. When demand decreases, the app stops using the public cloud.

In summary, *cloud native* is a combination of technologies and practices. Applications and infrastructure that are resilient, scalable, observable, and automated are “cloud native”. Organisations that embrace processes and standards that enable frequent and reliable deployments are “cloud native”. You can be cloud native if you run in the public cloud or your own on-premises datacenters.

Let’s dig deeper into the specific learning objectives for the exam.

Resiliency

Resilient applications and infrastructure can self-heal when things break. In fact, sometimes we call resiliency “self-healing”.

A simple infrastructure example is a node pool. Most cloud platforms let you create a pool of virtual servers called a node pool. If you configure one with 10 virtual servers, and one of them fails, the system can self-heal by spinning up a new node and automatically adding it to the pool.

Kubernetes offers self-healing capabilities for applications **and** infrastructure. For example, if Kubernetes knows you need 5 containers for a web front-end microservice, and one of them fails, it spins up a new one so you still have 5.

Although these are simple examples, combining resilient applications with resilient infrastructure is extremely powerful.

Loose coupling and building for failure

A key principle when designing resilient systems is to expect things to fail. This forces you to build stuff that can handle failures. A popular and battle-tested infrastructure example is disk mirroring. This is where two copies of the same data are written to two separate storage devices. When one of them fails, you can still access the data and no downtime or data loss is incurred. The key word from the previous sentence is *when*, not *if* – *when one fails* not *if one fails*.

Modern microservices applications should be designed and built in the sure knowledge that components will fail from time to time. Simple tactics to deal with this include loose coupling of microservices and consuming via APIs. That’s a lot of buzzwords, so this non-technical analogy should help.

Imagine you’ve bought a product that shipped with a missing part. An option is to contact the customer services team. To do this, you call a generic phone number and speak to Mia who opens a support case and ships the missing part. Two days later it still hasn’t arrived, so you call back. This time Ross answers the phone and you ask to speak to Mia. Unfortunately, she’s on vacation for a week, so you give your case number to Ross. He’s able to help you because your case details are stored on an external shared system the whole team has access to. In this example, you’re *loosely coupled* with the customer services team via a generic phone number and your case notes are stored on a separate system the entire team can access. This means you can speak to any member of the customer services team and they can help you with your case.

A *tightly coupled* version of the same scenario might look like this. You’ve bought a product that arrived with a missing part. You call Mia, from the customer service team, on her direct phone number. She records some case notes on her laptop and ships the missing part. Two days later it still hasn’t arrived, so you call her back. However, her phone keeps going straight to messages. After two days of trying to contact Mia you manage to find the phone number of another member of the customer support team called Ross. You call him on his direct phone number and he explains he can’t access your case notes because they’re on Mia’s laptop and she’s taken it home with her. In this scenario you were tightly coupled to a specific member of the support team and things fell apart when your person wasn’t there.

In the world of cloud native microservices applications you should loosely couple services and store data outside of the app so that when individual microservices fail and change, the overall application keeps functioning. A simple example might be microservice A talking to microservice B via network abstraction, such as a shared DNS name, and all state being stored in an external data store. This way, if any instance of microservice B goes down, you can reach other instances via the same DNS name and they can access the same state from the external store.

A tightly coupled version would be microservice A communicating directly with instance 1 of microservice B that stores state locally. When instance 1 of microservice B fails, the state is lost and none of the other instances can take over.

Self-healing

Another important feature of resilient applications and infrastructure is the ability to recover without human intervention – the system just heals itself.

Three concepts are key to the way this works.

- Desired state
- Observed state
- Reconciliation

Desired state is what you want. Observed state is what you have. Reconciliation is the process of keeping observed state in sync with desired state.

Consider the following example. You tell Kubernetes you want 5 web server pods running version 2.2 of your software. That’s your desired state. Kubernetes implements it and sets up a watch loop to observe things. At this point, desired state is 5 pods running version 2.2 and observed state is 5 pods running version 2.2. However, if something breaks and you lose a pod, observed state will drop to 4 pods but desired state will still be 5. Kubernetes reconciles the situation by starting a new 5th pod running version 2.2 so that observed state is brought back into sync with desired state. This process is called *reconciliation* and is key to self-healing without human intervention.

Health checks

One final thing on resiliency is health-checks and probes. These are an extra layer of checks that often include *application intelligence* – this is jargon for a set of checks that test your application is working as expected.

For example, the previous section on desired state and reconciliation has no *application intelligence.* It only makes sure there are 5 pods with version 2.2 of your software running. It does nothing to make sure the application inside each pod is working properly.

A simple non-technical example might be a quick visual check that you have milk in the fridge. Your desired state might be a single carton of red-top milk in the fridge. A quick visual check proves you have that. However, it does nothing to check if the milk is fresh or how much is in the carton. In this scenario, an *application health-check* might pick up the carton, check the date, and may be do a "sniff test" ;-) The point is… while the quick visual check might make sure observed state matches desired state (you want one carton of milk and you have one carton of milk) it does nothing to check the milk is good.

In the cloud native world, health-checks are usually specialised requests that test whether your application is working as expected. Sometimes we call them "probes" as they *probe* your applications.

Many web-based applications expose a dedicated health-check endpoint such as `/healthz`. For example, if the app is reachable at knca-app.kubernetes.local it would expose kcna-app.kubernetes.local/healthz. Sending requests to this causes the app to return data you use to determine app health. An example URL might be `myapp.k8s.local/healthz`.

In summary, loose coupling enables more fluid environments where things can break and be replaced without impacting the overall app and user experience. Systems like Kubernetes constantly observe their environment to ensure what they observe (observed state) is what you requested (desired state). If observed state doesn't match desired state, a process of *reconciliation* fixes the issue. Health checks and probes add an additional layer of checks that test whether applications are working as expected.

Autoscaling

Autoscaling is the ability of applications and infrastructure to automatically grow and shrink to meet demand.

Three things are vital for autoscaling to be truly cloud native.

- Applications **and** infrastructure need to be able to scale
- Scaling has to be automatic
- Scaling has to be up and down

Kubernetes has the ability to scale applications up and down based on current workload. It can use simple metrics, such as CPU and memory consumption, or it can use application-specific metrics such as queue size and request response times.

A simple example might be the reporting microservice of an application. If business units are running end-of-year reports and the reporting microservice is experiencing a 10x increase in demand, Kubernetes can spin-up additional reporting containers to meet demand. When demand decreases, Kubernetes can terminate the additional containers. If the application is running on a public cloud, Kubernetes can also scale-up the infrastructure to help support the additional containers. A common example is Kubernetes adding more nodes to the node pool.

Let’s take a closer look at the example.

Assume your business application is running on 5 nodes (cloud instances/virtual machines) and the reporting microservice is running on two containers. Multiple business units kick-off their year-end reports and load on the reporting system goes through the roof. Kubernetes observes this increased load and deploys 3 more reporting containers to handle the load. However, your 5 nodes are maxed out and don’t have the capacity to run the 3 additional containers. No problem, Kubernetes talks to your cloud and spins up an additional node to run the 3 new containers. Reports run quickly and on time, and when demand on the reporting microservice decreases, Kubernetes terminates the additional 3 reporting containers as well as the additional node.

It’s important to understand a few things from the example.

The scaling up and scaling down was automatic. This is possible because Kubernetes is constantly observing key performance metrics for your applications. When it sees things, such as increased load or longer response times, it scales the application and infrastructure to cope. It also scales things down when demand drops. The latter is important in controlling costs, especially in public cloud environments where you pay for what you use.

Microservices architectures are vital to this behavior as they allow Kubernetes to only scale microservices that need it.

Finally, the type of scaling described so far is referred to as *horizontal scaling*. At a high level, horizontal scaling is adding more instances of something you already have. The opposite is *vertical scaling* which attempts to increase the size of what you already have. The following non-technical example might help.

Assume you run a business that provides delivery services and you currently have one small delivery vehicle. As the business grows you also need to grow your capability to make more deliveries. You can horizontally scale your delivery capability by adding 2 more small delivery vehicles. Alternatively, you can vertically scale by selling your small delivery vehicle and replacing it with a single larger vehicle.

Switching back to the end-of-year reporting analogy. If one small reporting container cannot handle year-end demand, you can horizontally scale it by adding more of the same small containers. Alternatively, you can vertically scale by replacing the small reporting container with a single larger one. The same can be done for the infrastructure supporting it – you can horizontally scale by adding multiple small nodes, or vertically scale by adding a single huge node.

Generally speaking, Kubernetes and cloud native patterns prefer horizontal scaling.

Kubernetes has the following autoscalers.

- Horizontal Pod Autoscaler (HPA)
- Cluster Autoscaler (CA)

The horizontal pod autoscaler adds more *pods* when demand goes up and removes them when demand goes down. The cluster autoscaler integrates with your public cloud and adds or removes *nodes* according to demand.

For both autoscalers it's important you configure upper and lower bounds. For example, you might configure Kubernetes to limit a node pool to a maximum of 20 nodes. This can help you control costs and reduce the risk of runaway processes and malicious attacks growing the pool too large. Remember, nodes cost money in the cloud.

In summary, applications and infrastructure should be able to scale automatically based on metrics. Microservices architectures make it possible to selectively scale just the parts of an application that need it. Kubernetes has a horizontal pod autoscaler that scales application microservices to meet demand. It also has a cluster autoscaler that scales cloud infrastructure to meet demand.

Finally, it's important to understand that scaling is a constant attempt to balance performance with cost – providing a good user experience without consuming and paying for too much infrastructure.

Serverless

At a high level, serverless is an event-driven computing model where you write small functions that execute when an event triggers them. That's a lot of jargon, and we'll explain it all in a second, however, there's a couple of things we need to address before getting into the detail.

1. The terms *serverless* and *Function as a Service (FaaS)* refer to the same thing
2. Serverless uses servers

Point 2 raises the obvious question; if serverless uses servers, why do we call it serverless?

The answer is simple, the servers are hidden so well that we **never** have to think about them. Think about it like this… before cloud computing we spent a lot of time and energy working with servers. We had to buy them, rack and stack them, pay for maintenance contracts, and even replace them when they failed or got too old and slow. The cloud came along and took **most** of the server work away. Finally, along came serverless and took **all** of the server work away. So, in the serverless model, developers can focus entirely on writing code while the serverless platform takes care of all the compute and storage requirements.

Serverless is event-driven

The event-driven element is key to the serverless model. Consider the following serverless workflow.

1. Developer writes code
2. Code gets packaged as container image
3. Code gets uploaded to serverless platform
4. Code gets associated with an event
5. Event occurs and code runs

In this model, the code gets uploaded to the serverless platform and never executes until its associated event occurs. If the event never occurs, the code never runs. If the event occurs a lot, the code runs a lot. This has financial implications on fully-managed cloud platforms where you only pay when your code runs. Code that runs a lot will cost you more than code that runs less. Also, despite the seemingly low prices advertised by many serverless platforms, code that executes a lot can cost a lot of money.

A couple of other points from the list.

Most modern programming languages work with serverless, however, you should design and write serverless functions according to microservices design patterns – small, specialised functions that do a single well-defined job.

It’s also common to package serverless functions as containers as they are small, lightweight, and fit well with microservices architectures. However, not all serverless platforms support containers.

Serverless platforms

A lot of public clouds provide their own proprietary serverless platforms. These are usually easy-to-use and operate on flexible pay-as-you-go pricing models. However, they're usually proprietary and lack governance and standards. Popular cloud-based serverless platforms include.

- AWS Lambda
- AWS Fargate
- Azure Functions
- Google Cloud Functions

Popular open-source serverless platforms that support Kubernetes include.

- Knative (pronounced "kay native")
- OpenFaaS (pronounced "open faz")

At the time of writing, the serverless ecosystem is lacking strong governance and standards. However, the CloudEvents project is aiming to resolve this. It's currently an incubating CNCF project.

Serverless and scaling

Scaling is an integral part of most serverless platforms – the platform spins up the container and infrastructure required to execute your function and then automatically releases them when it completes. If a function executes one million times in a minute, the platform spins up the required containers and infrastructure and sends you the bill. It also releases them when your functions complete. As such, scaling is just part of the way serverless works.

In summary, serverless is often called function as a service (FaaS) and is an event-driven computing model where you upload small functions to the serverless platform and associate them with events. When those events occur, the functions execute. Behind the scenes, many serverless platforms execute functions inside of short-lived containers. Scaling is native to serverless, and many clouds have their own proprietary serverless platforms that operate a pay-as-you-use pricing model – you only pay for each time your functions execute. However, proprietary serverless platforms lack open standards and increase your risk of lock-in.

Finally, when serverless first hit the market there was a lot of speculation that it might become the dominant model for modern applications. However, the hype has faded and it's now clear that serverless will work together with containers and virtual machines in shaping the future of cloud computing.

Community and governance

Most cloud native technologies, including Kubernetes, are open source. This gives them access to large and passionate communities that develop and support them. However, strong governance is required to provide direction and order.

To address this requirement, the Cloud Native Computing Foundation (CNCF) was founded in 2015. It defines three maturity stages projects have to pass through on their journey to "graduation". The stages are.

1. Sandbox
2. Incubating
3. Graduated

This three-step process helps projects mature and eventually graduate as production-grade projects that can thrive and bring value to the community. Kubernetes was the first CNCF project and graduated in 2018. Since then, more than 15 other projects have graduated, and more than 100 are either in the sandbox or incubating stages.

Projects have to prove their maturity to the CNCF Technical Oversight Committee (TOC) if they want to progress through the stages. Each stage has its own set of requirements, but the following is an example of some of the requirements to graduate.

- Prove the project is in active use by real users
- Have people from at least two organisations contributing to the project
- Complete an independent 3rd-party security audit and publish the results
- Achieve and maintain a Core Infrastructure Initiative Best Practices Badge

There are other criteria, but these demonstrate the kind of governance provided by the CNCF – everything pushes projects towards maturity and good practices. However, CNCF governance is as lightweight and hands-off as possible so that projects learn to govern themselves.

Roles and personas

Being cloud native is more than the technologies you use. It's also about creating a co-operative culture and the right working environment. A large part of this is having the right set of roles and job titles that are fully supported by senior management.

The following are typical cloud native job roles and personas that you'll see in the real world. You'll also be tested on them in the exam.

Cloud Architect

A major responsibility of a Cloud Architect is choosing the right cloud platforms for the business and individual applications. This can include a mix of public clouds, private clouds, and hybrid clouds.

The cloud architect is also responsible for designing infrastructure and applications to be cloud native – resilient, scalable, observable, automated etc.

An awareness of costs and the differences between capital expenditures (capex) and operational expenditures (opex) is vital for this role.

Public cloud is usually an opex cost where you rent infrastructure on a pay-as-you-go basis. On-premises and private clouds are usually capex costs where you pay large amounts up-front to procure and own infrastructure.

DevOps Engineer

As the word suggests, "DevOps" engineers have a mix of developer and operational skills. This means some experience with programming languages as well as some experience with operational tools such as monitoring, logging, and deployment. It's common for a DevOps engineer to be 80% developer and 20% operator.

As well as this, the following things are expected of a DevOps Engineer.

1. Understand the entire application and application lifecycle
2. Be skilled with the right tools
3. Possess a willing mindset that's not afraid of change or failure

DevOps tools include continuous integration, continuous development (CI/CD), and GitOps tools that automate application development, testing, and deployment. A DevOps mindset means not being afraid to try new things and not being afraid to fail.

A major reason the DevOps role was created was to break down the barriers that often exist between development teams and operations teams – quite often they battled against each other which wasn't good for the business or customers. This is why the best DevOps Engineers have experience in both disciplines and are willing to see both perspectives.

DevOps Engineers must also be confident enough to push teams and organisations toward a cloud native culture where change is the new normal.

CloudOps Engineer

This is a more recent role and is similar to DevOps but has a slightly narrower focus on cloud infrastructure and tools. CloudOps Engineers may focus entirely on public cloud technologies, whereas DevOps Engineers need wider skills that manage on-premises infrastructure and apps.

Security Engineer

A Security Engineer must understand cloud native infrastructure and applications, as well as the specific threat vectors that apply. These threats can be very different from traditional monolithic applications running solely on on-premises datacenters.

A simple example is the increased reliance microservices applications have on sharing data over IP networks. In the past, monolithic applications implemented all features and functionality in a single binary that was deployed as a single large application on a single physical or virtual server. This meant all internal application communication was over secure local channels on the physical server. The microservices model breaks every application feature into its own small app and distributes them across multiple servers. This means all internal application communication now has to traverse IP networks and introduces totally new threat vectors.

Another simple example is the shared kernel model most container platforms implement. This is where every container shares the OS kernel of the host it's running on. If the host is compromised, so is every container running on that host. This introduces an entirely different set of threats to the virtual machine model where there is no shared kernel.

From a culture perspective, Security Engineers must embed themselves within other teams to ensure security is a priority at every step of application and infrastructure design, development, and deployment. They must ensure that security is always on the agenda and never an afterthought.

DevSecOps Engineer

A DevSecOps Engineer is a DevOps Engineer with security skills and a security-first mindset.

Many people consider this to be the same as a DevOps Engineer, as many organisations now expect their DevOps engineers to understand the importance of security, possess security skills, and possess a security-first mindset. This role is often considered one of the best ways of ensuring security is part of every discussion and part of every step of design, development, and deployment.

You may hear the term security *left-shift.* This is moving security to the very start of the development process (all the way to the left of the workflow). DevSecOps engineers can make that happen.

Data Engineer

Data is the lifeblood of many organisations – if you lose it or it gets corrupted, the entire organisation might go out of business.

The role of the Data Engineer is to ensure data is stored and protected appropriately, and making sure it’s always available when needed. This includes data locality (storing it in the right places), performance, availability, and protection. It also involves knowledge and experience with tools to analyse data.

Full Stack Developer

A Full Stack Developer is a multi-skilled developer that understands, and can code, the front-end (client-side) and back-end (server-side) components of a system. They need hands-on experience with multiple programming languages as well as infrastructure components such as cloud platforms and data stores.

A typical Full Stack Developer will be proficient in at least two programming languages, have skills with at least one cloud platform and its tools, and have experience with at least one data store or database technology. These back-end data store technologies are usually open-source.

Full Stack Developers are expected to be highly productive as they can make fast decisions and “just get stuff done” thanks to their broad skillset. A full stack developer is almost always faster and more agile than having to pull an entire team together – the Full Stack Developer gets the job done herself.

Site Reliability Engineer (SRE)

The main role of a Site Reliability Engineer (SRE) is to create, maintain, and optimise cloud native systems. It’s sometimes considered DevOps Engineer 2.0, but usually has more of an operational focus than a DevOps engineer. For example, a typical SRE might be 20% developer, 80% operations.

Like DevOps Engineers, SREs have a mindset of continual change and optimisation. However, it goes beyond simple DevOps engineering by focussing on key site reliability metrics such as Service Level Objectives (SLO), error budgets, Service Level Indicators (SLI), and Service Level Agreements (SLA).

It’s often said that the goal of an SRE is to automate themselves out of a job.

Open standards

Open standards are a big deal in cloud native. They help users avoid vendor lock-in and product lock-in, and they give developers confidence the products they build will work and integrate well with the wider ecosystem.

A common non-technical example is trains and train tracks. The majority of train tracks across the world are the same size and configuration (1,435 mm standard gauge). This means service providers can buy trains from one supplier today and switch to another supplier tomorrow with full confidence the trains from both suppliers will work on the same tracks. It also gives train manufacturers confidence the trains they build will work on tracks all over the world. The net result is confidence, and confidence enables progress.

The Open Containers Initiative (OCI) provides the following important standards in the cloud native ecosystem.

- Image spec
- Runtime spec
- Distribution spec

The image-spec defines a standard way to build and package container images, the runtime-spec standardises container execution environments and container lifecycles, and the distribution-spec standardises content distribution via registries. When combined, they act like the standardisation of train tracks and provide the confidence developers and users require for a strong ecosystem to emerge. Consider the following examples.

The image-spec gave developers and startups confidence to create their own *build tools* to create OCI container images. Prior to the OCI image-spec, everybody used the Docker tools and the `docker build` command to build container images. Now there are a lot of tools to choose from.

The runtime-spec gave developers and startups confidence to create their own container runtimes in the knowledge they'd work with other tools and safely run images created against the OCI image-spec. Prior to the OCI runtime-spec everybody used Docker and the `docker run` command to run containers. Now there are a lot of container runtimes and CLI tools to choose from.

Finally, the distribution-spec gave developers and startups the confidence to build their own container registries. Again, Docker Hub was originally the only container registry. Now there are lots to choose from.

Consider the following real-world success story made possible by the OCI standards. Kubernetes originally used Docker as its container runtime (the low-level tool to start and stop containers). Over time, Docker grew and became overkill for what Kubernetes needed. As a result, Kubernetes made the runtime layer pluggable so that users can pick the best container runtime for their requirements. And the best part about it… as long as the new container runtime works with OCI images, which almost all of them do, everything keeps working and no changes are needed to existing container images.

If the OCI image spec didn’t exist, users and projects would lack the confidence to make changes like this as the risk of interoperability issues would be too high.

The OCI operates under the umbrella of the Linux Foundation and is made up of volunteers from major cloud native and container-focussed companies. Its goal is to provide lightweight governance and standards for low-level container-related technologies.

Other standards that have been instrumental in the Kubernetes ecosystem include the following:

- Container Runtime Interface (CRI)
- Container Network Interface (CNI)
- Container Storage Interface (CSI)
- Service Mesh Interface (SMI)

These are all instrumental in standardising important low-level components of Kubernetes so that ecosystem partners can develop products and be confident they’ll work as expected. It also creates environments and tools that are swappable and prevent vendor and product lock-in. You’ll learn about each of these later in the book.

Chapter summary

In this chapter, you learned that being *cloud native* is a lot more than running your apps in the public cloud. In fact, cloud native is a combination of technologies and behaviors. The technologies include platforms like Kubernetes, as well as microservices applications that run as containers and serverless functions. The behaviors include a mindset that’s open to change, as well as a culture that’s open to acting quickly and investing in the right people and technologies. Open standards are also key bringing the kind of stability and confidence required for a strong ecosystem of tools to emerge.

Exam essentials

Autoscaling

Autoscaling is the ability for applications and infrastructure to automatically grow and shrink to meet demand. When demand goes up, more application instances and more infrastructure can be dynamically added to the environment. When demand drops, application instances and infrastructure can be removed. Kubernetes has a horizontal pod autoscaler to add and remove application instances to meet demand. It also has a cluster autoscaler that integrates with cloud platforms to add and remove nodes to meet infrastructure requirements. Metrics are used to make scaling decisions. Basic metrics include CPU and memory, whereas custom application metrics include things such as queue length and request response time.

Serverless

Serverless is a modern event-driven computing model where small functions are associated with specific events and only execute when those events occur. Sometimes we call it Function as a Service (FaaS) and many public clouds offer proprietary serverless platforms where you upload your functions, associate them with events, and only pay for the time your functions execute. Open-source serverless platforms include OpenFaaS and Knative.

Community and governance

The cloud native community is a large and thriving open-source community that actively develops and maintains important cloud native technologies such as Kubernetes. While these communities are amazing, the projects need strong governance to keep the community focussed and the projects heading in the right direction. The major source of governance in the cloud native ecosystem is the Cloud Native Computing Foundation (CNCF). It was founded in 2015 and defines three stages of project maturity – sandbox, incubating, and graduated. Projects have to prove their maturity to progress through the stages, and each stage is designed to guide projects towards further maturity. Kubernetes, and over 15 other projects, have graduated the CNCF.

Roles and personas

An important step towards being cloud native is hiring the right people into the right roles. People need to have strong open mindsets and not be afraid to try new things and occasionally fail. Hopefully the failures will be in test and development environments, but it's vital that people are open to trying lots of new things and failing along the way. As well as hiring the right people, they have to be hired into the right roles with the full support of senior management. For example, DevOps Engineers need to be free and authorised to implement DevOps practices and use DevOps tools. There's no point hiring the right person and giving them the title of DevOps Engineer if your company culture and senior management won't allow them to carry-out their role.

Open standards

Open standards are vital in two very important areas. They give developers confidence the products they build will work as expected and integrate well with other tools. They also help prevent technology lock-in by encouraging large ecosystems with lots of compatible product choice. The Open Containers Initiative (OCI) has created three container-related standards that have helped the container ecosystem grow significantly. Thanks to the OCI there is a lot of choice for container runtimes, container image build tools, and container registries.

Recap questions

See Appendix A for answers. Page 159 in the paperback edition.

1. Which of the following is another term for resiliency?

- A. Automation
- B. Self-healing
- C. Robustness
- D. Autoscaling

2. Which of the following can be resilient?

- A. Only applications
- B. Only infrastructure
- C. Applications and infrastructure

3. Which of the following is true about designing cloud native applications?

- A. They should be designed to expect failures
- B. They should be designed so failures cannot happen
- C. They should be designed to shut down when failures happen
- D. They should become read-only when failures happen

4. Which of the following design principles is key to resilient apps?

- A. Tight coupling
- B. Over coupling
- C. Loose coupling
- D. Mesh coupling

5. What is the name for the process of synchronising observed state with desired state?

- A. Re-distribution
- B. Realignment
- C. Retribution
- D. Reconciliation

6. Which of the following is commonly used as an application health-check endpoint?

- A. /healthz
- B. /health
- C. /statusz
- D. /status

7. What do health-checks normally do?

- A. Return info on whether an application has a virus
- B. List the known vulnerabilities of an application
- C. Test whether an application is coping with demand and needs scaling up
- D. Test whether an application is working as expected

8. Which of the following best describes “desired state”?

- A. What you want
- B. What you have
- C. Cluster quorum
- D. The current state of your cluster

9. Which of the following is true of cloud-native autoscaling? Choose all correct answers.

- A. It's always based on memory consumption
- B. Scaling must be up and down
- C. Scaling must be automatic
- D. Apps and infrastructure can scale

10. Why is scaling down so important?

- A. Running unnecessary infrastructure in the cloud incurs costs
- B. Running too many application instances can increase response times
- C. Running too much infrastructure causes over-distribution of apps
- D. Unused infrastructure can cause duplicated IP addresses

11. Which of the following is an advantage of microservices applications when scaling?

- A. They can scale up and down
- B. All services can be scaled together
- C. Individual services can be scaled
- D. They simplify networking

12. Horizontal scaling is best described as?

- A. Adding more of something you already have
- B. Increasing the size of something you already have
- C. Increasing the memory of existing instances
- D. Scaling by a factor of 2 every time

13. Which type of scaling do cloud native apps normally prefer?

- A. Vertical scaling

- B. Horizontal scaling
- C. Diagonal scaling
- D. 3D scaling

14. Which of the following are popular Kubernetes autoscalers?

- A. Horizontal Pod Autoscaler (HPA)
- B. Multi-dimensional Autoscaler (MDA)
- C. Flux Autoscaler (FA)
- D. Cluster Autoscaler (CA)

15. Which of the following is true of serverless? Choose all correct answers.

- A. There are no servers
- B. It's a form of event-driven computing
- C. It uses servers
- D. It will replace the need for containers

16. Which of the following is a common serverless packaging format?

- A. Virtual Machine template
- B. COM package
- C. Dynamic link library (DLL)
- D. Container image

17. Which of the following application architectures is best-suited for serverless?

- A. Microservices
- B. Monolithic

18. Why are containers a common choice for serverless workload packaging? Choose all correct answers.

- A. They’re small
- B. They’re secure
- C. They fit well with microservices architectures
- D. They don’t need servers to run

19. Which of the following is most likely to be true?

- A. Serverless functions that execute a lot will cost more
- B. Serverless functions that execute less will cost more
- C. Serverless functions that execute a lot will cost less
- D. Serverless functions are usually free to execute

20. Which CNCF project is aiming to bring standards for serverless functions?

- A. The Open Container Initiative
- B. CloudEvents
- C. The Open Serverless Initiative
- D. The Serverless Foundation

21. Which of the following cloud native features is inherent in most serverless platforms?

- A. Zero downtime updates
- B. Observability
- C. Autoscaling
- D. Telemetry

22. Which of the following is another common name for serverless?

- A. Servergone
- B. Function as a Service (FaaS)
- C. Just-in-time cloud (JiC)
- D. Containerisation

23. Which of the following organisations was founded in 2015 to provide governance for open-source container-related projects?

- A. The Linux Foundation (LF)
- B. The Open Container Foundation (OCF)
- C. The Open Compute Project
- D. The Cloud Native Computing Foundation (CNCF)

24. What are the three maturity stages of the CNCF?

- A. Incubating
- B. Sandbox
- C. Maturing
- D. Graduated

25. Which of the following are examples of things projects have to provide in order to graduate the CNCF? Choose all correct answers.

- A. A completed security audit performed by an independent 3rd-party
- B. Proof the project is being used by users
- C. Proof of committers from at least two organisations
- D. Proof of strong financial backing from at least two investors

26. Which of the following are features of the public cloud? Choose all correct answers.

- A. Opex cost model
- B. Capex cost model
- C. Pay-as-you-use
- D. You own the infrastructure

27. Which of the following are elements of a DevOps persona (role)? Choose all correct answers.

- A. Team management skills

- B. Excel skills
- C. Have developer and operations experience
- D. Understand the entire app

28. What is a potential difference between a DevOps Engineer and a CloudOps Engineer?

- A. The CloudOps Engineer needs more skills than a DevOps Engineer
- B. CloudOps Engineers get paid a lot more
- C. CloudOps Engineers have more of a developer focus
- D. CloudOps Engineers focus mainly on cloud skills and cloud tools

29. Which of the following is an example of a DevOps tool?

- A. CI/CD
- B. Microsoft Excel
- C. Oracle
- D. VMware

30. Which of the following is a potential advantage of hiring a full stack developer?

- A. They're cheaper than traditional developers
- B. They have more people skills than traditional developers
- C. They make fast decisions and just get stuff done
- D. They write shorter code

31. What is commonly said about Site Reliability Engineers?

- A. They should automate themselves out of a job
- B. They shouldn't be trusted
- C. They should have previously worked at Google
- D. They are the best scripters in the industry

32. What are some of the benefits of open standards? Choose all correct answers.

- A. They force projects to be open-source
- B. They help avoid lock-in
- C. They provide confidence for developers
- D. They introduce security risks

33. Which of the following are OCI specs? Choose three.

- A. image-spec
- B. distribution-spec
- C. runtime-spec
- D. pod-spec

34. You're building a business case to re-factor an existing application to make it cloud native. What are some of the benefits this will bring? Choose all correct answers.

- A. Application resiliency
- B. Automatic scaling
- C. Less human interaction with changes
- D. Better visibility into what's going on inside the app

3: Container orchestration

This chapter covers everything you'll need to pass the *Container orchestration* section of the exam. The topics account for 22% of the exam and build on what you learned in the previous chapter. They also and provide a foundation for the topics covered in upcoming chapters.

The chapter is divided as follows.

- Primer
- Container runtimes
- Container orchestration fundamentals
- Container security
- Container networking
- Service meshes
- Container storage
- Exam essentials
- Recap questions

Primer

Containers are similar to virtual machines – both allow you to run multiple applications on a single physical server. However, there are important differences.

At a high level, containers are more lightweight than virtual machines. This means the same server can run more containers than virtual machines. As they're both technologies for running applications, containers let you run more applications on your servers. For example, a server that can run 10 virtual machines might be able to run 50 containers. This enables huge cost savings for organisations – you need less infrastructure to run your apps.

Being lightweight also means containers start faster than virtual machines and are easier to package and share.

Container workflow

The following list shows the common steps involved in creating and running an application as a container.

1. A developer writes an application
2. The application is packaged as a *container image*
3. The container image is uploaded and shared in a registry
4. A server downloads the image
5. The server runs the image as a container

Let’s look a little closer.

Developers write applications in their favourite languages – there’s no need to learn new languages for containers.

The application, and all dependencies, are packaged into a container image that we normally just call an *image*. As the app **and its dependencies** are packaged in the same image, you shouldn’t have situations where an application works on a developer’s laptop but doesn’t work in production. This used to happen because production systems and developer laptops were running different versions of dependencies. Containers fix this problem by bundling the app **and all dependencies** in the same container image.

It’s normal practice to host container images in a centralised registry so they’re accessible from different environments (dev, test, production, etc.) and can be securely shared with different teams. Many container registries exist. Some are designed to run inside your firewall on your own private network, whereas others exist on the public internet. *Docker Hub* was the first major container registry and is available on the internet. However, like all registries, it's divided into repositories that you can hide and restrict access to.

Once an image is shared to a registry it can be downloaded to a server and ran as a container. You can restrict who can access and download your images, and the servers that run them can be physical servers, virtual machines, and even cloud instances.

When an image is executing we call it a *container*. This means a container is a running instance of an image. If you've worked with VMware, it might be useful to think of images as being like VM templates, and containers as being like running VMs. However, remember that images and containers are smaller and more lightweight than VM templates and VMs.

Servers that run containers need a special piece of software called a *container runtime*. A later section is dedicated to learning about these.

Containers and microservices

Containers were vital in shaping microservices architectures.

As mentioned in previous chapters, microservices applications are made of lots of small independent services that work together to form a useful application.

Each microservice is normally responsible for a single feature and packaged as its own container image. For example, a microservices application with front-end functionality, middleware functionality, and back-end functionality might be divided into the following three microservices/container images.

- Front-end
- Middleware
- Back-end

In this example, the front-end service is developed by a dedicated team, packaged as its own container image, and deployed independently of the other two microservices. The same is true for the other two. If demand on the front-end increases, you can scale it up by adding more front-end containers. If demand drops, you can scale back down by terminating the additional containers.

This demonstrates the relationship between microservices and containers. It also shows how each microservice can be scaled independently by simply adding more copies of that microservice's container.

Container architecture

Containers are sometimes referred to as *virtual operating systems*. Let's compare them to virtual machines to make this clear.

Hypervisor virtualisation, such as VMware, virtualises hardware. This means it takes physical servers and slices them into virtual servers that we call *virtual machines (VM)*. For example, CPUs are sliced into virtual CPUs, storage is sliced into virtual storage, and network cards are sliced into virtual network cards. These virtual resources are combined into VMs that look, smell, and feel like regular physical servers. You then install a single OS and application in each VM.

Container virtualisation, such as Docker, virtualises operating systems (OS). This means it takes operating systems like Linux and Windows and slices them into virtual operating systems called *containers*. For example, process trees are sliced into virtual process trees, filesystems are sliced into virtual filesystems, and network interfaces are sliced into virtual interfaces. These virtual resources are combined into containers that look, smell, and feel like regular operating systems. You then run a single application in each container.

To recap, hypervisor virtualisation slices physical servers into virtual machines that look, smell, and feel like physical servers. Container virtualisation slices operating systems into containers that look, smell, and feel like a regular OS. Figure 3.1 shows a comparison.

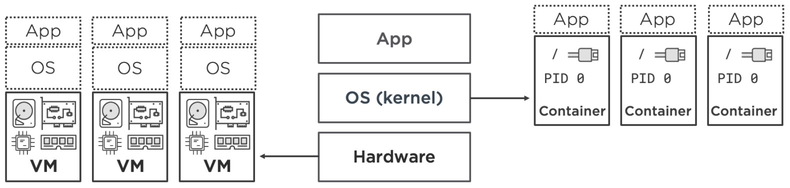

Figure 3.1

In summary. Containers are a form of virtualisation that is lighter weight than hypervisor virtualisation like VMware. This means containers waste less infrastructure and are more portable. Application code and dependencies are packaged into container images that are stored in centralised registries and downloaded by servers that want to run them. Servers use a container runtime to download container images and run them as containers. You also learned that containers virtualise operating systems – each container is essentially a virtual operating system running a single app. A server running 10 containers can run 10 apps.

Container runtimes

Containers run on physical servers, virtual machines, and cloud instances. As they all look the same to a container we'll refer to them generically as *nodes.*

Every node that wants to run containers needs specialised software called a *container runtime.* This is responsible for downloading container images from registries, as well as creating, starting, stopping, and deleting containers on the node. Docker is the best-known container runtime, but lots of others exist.

Let’s revisit the workflow mentioned earlier to show where container runtimes fit in.

Developers write applications that get built into container images and stored in registries. To run a container, a node needs to download the image and start a container from it. The software that does this is the container runtime. The act of downloading a container image is usually referred to as *pulling* the image.

If you’ve worked with VMware, it might be helpful to compare container runtimes with ESXi. This is because both are low-level tools that perform tasks such as starting and stopping containers or VMs. In the VMware world, higher-level tools like vCenter add more advanced functionality. In the container world, higher-level tools like Kubernetes add more advanced functionality.

The Open Containers Initiative (OCI)

Docker was the original container runtime and is still widely used. However, the Open Containers Initiative (OCI) created a set of industry standards that made it simple and safe for other developers and companies to create their own container runtimes. In fact, Docker, Inc. was a founding member of the OCI and was heavily involved in creating the standards. Some of the most popular container runtimes include:

- Docker
- containerd (pronounced “container dee”)
- runC (pronounced “run see”)
- CRI-O (pronounced “cry oh”)
- gVisor (pronounced “jee visor”)
- Kata containers

As you’d expect, each one has advantages and disadvantages.

Docker has been around a long time and is very widely used. However, it does a lot more than just the low-level stuff. For example, Docker does advanced scheduling and implements a lot of higher-level features normally provided by platforms such as Kubernetes. As a result, it's overkill for a lot of projects and scenarios.

containerd (pronounced "container dee") is just the low-level runtime code stripped out of Docker and donated to the CNCF in 2017. Thanks to its lightweight nature, it's fast becoming the most popular container runtime used by Kubernetes. It's based on OCI specs and works with OCI images.

> **Jargon:** The OCI created three specifications that have defined low-level industry standards for containers. One of these specs is the *image-spec* that defines what a container image should look like and how it's constructed. Container images that conform to the OCI image-spec are often referred to as *OCI images*. Almost all container runtimes work with OCI images. Put in more simple terms, *Docker images* and *OCI images* are the same.

Types of container runtimes

There are three main types of containers and container runtimes:

- Namespaced (shared kernel)
- Sandboxed (proxied kernel)
- Virtualised (dedicated kernel)

That's a lot of jargon that boils down to size and security.

Namespaced containers are the fastest and most lightweight, but they're the least secure. This is because every container on the same node shares the node's kernel. Docker, containerd, and CRI-O are common examples of container runtimes that create namespaced containers.

Virtualised containers are the slowest to start and consume the most resources, however, they’re the most secure. This is because every container runs inside its own virtual machine with its own kernel. Windows containers and Kata Containers are examples of container runtimes that create virtualised containers.

Sandboxed containers are an attempt at a middle-ground. They’re faster and smaller than virtualised containers but bigger and slower than namespaced containers. They’re also in the middle when it comes to security. Implementation details are beyond the scope of this book and the exam, but they often implement a proxy layer between the container and the host’s kernel. gVisor is an example of a container runtime that creates sandboxed containers.

Kubernetes and containers

Even though Kubernetes is a container orchestrator, it cannot run containers without the help of a container runtime.

Figure 3.2 shows the relationship between Kubernetes and container runtimes. Kubernetes manages a cluster of nodes, each running a container runtime, and tells them when to start and stop containers. It also implements high-level intelligence such as self-healing and autoscaling etc.

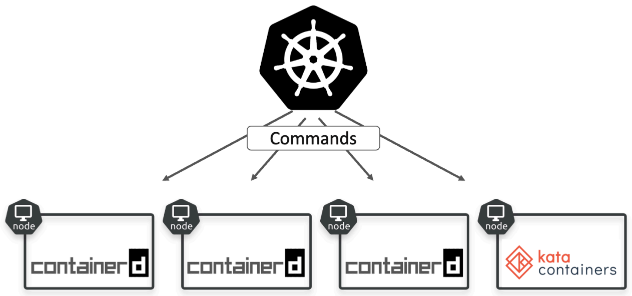

Figure 3.2

In the early days, Kubernetes used Docker as it's runtime and a lot of Kubernetes installations still do. However, Kubernetes implements a technology called the Container Runtime Interface (CRI) that allows users to pick their favourite container runtimes. As things stand, containerd is the default container runtime on most new Kubernetes clusters.

The OCI specs were instrumental in enabling the CRI and allowing Kubernetes to make container runtimes swappable – as long as container runtimes work with OCI images, you can swap one for another.

> **Jargon.** The OCI maintains low-level industry specifications for things such as container runtimes and container images. The CRI is what makes the container runtime layer of Kubernetes pluggable. Don't get the two mixed up.

In summary, container runtimes are low-level tools that pull images, start, stop, and delete containers. They come in different shapes and sizes to fit different business and technical requirements. For example, some are fast and lightweight but less secure, whereas others are bigger and slower but more secure. The OCI created specifications that brought important standards to the container ecosystem making it possible for a wide range of container-related tools and platforms to be built. Kubernetes is a higher-level tool that manages clusters of container runtimes and tells them which containers to run as well as bringing autoscaling and other cloud native features. Kubernetes also implements a pluggable layer, called the Container Runtime Interface, that allows you to choose the best container runtime for your requirements. Finally, as container runtimes are low-level tools, they usually operate in the background, and you normally interact with them indirectly via higher level tools such as Kubernetes.

Container orchestration fundamentals

As soon as we started using containers it became obvious we needed help managing them. This is because we were breaking large applications into lots of small loosely coupled microservices that were scaling up and down and we were rolling out new versions and updates all the time. As an example, a single large monolithic application that rarely changed would be replaced with a sprawling microservices application that changes all the time. Help was needed.

Tools like Docker Swarm, Kubernetes, and Mesosphere DCOS were created to help solve the problem. To cut a long story short, Kubernetes emerged as the most popular of those tools and is now the most important cloud native technology on the planet.

At a high level, Kubernetes is a *container orchestrator.* This means it runs and manages applications that run in containers. Part of this includes providing cloud native features such as scheduling, self-healing, and autoscaling.

Let’s consider a simple example. You have a business application with a front-end and back-end. Your developers have designed it with two microservices – one for the front-end and one for the back-end. Each one has been written and packaged as a container image and hosted in a container registry. You need a platform to run the application and provide cloud native features, so you choose Kubernetes. Your operations team builds a Kubernetes cluster with 3 nodes running the containerd container runtime. When the cluster is built, you tell Kubernetes to run your app with two containers for the front-end and two containers for the back-end – this is called your *desired state*. Kubernetes deploys the application and configures *watch loops* to make sure what is actually running on the cluster matches what you asked for. We call this *observed state* and *desired state*.

Let's assume that one of the cluster nodes fails and takes a front-end container with it. Watch loops will notice that *observed state* is now one front-end container but desired state is two. They will automatically fix this by starting a second front-end container on a surviving node so that observed state is back in sync with desired state. Also, as you've configured your node pool to have three nodes and it currently only has two, a new third node will be automatically created and added to the pool. We call this *self-healing* or *resiliency.*

Let's also assume you configured scaling metrics for the app and Kubernetes notices the front-end is struggling to cope with current demand. To resolve this, it automatically scales the number of front-end containers up from two to four. This resolves the issue, and when demand drops off Kubernetes scales back down to two.

This demonstrates the cloud native capabilities of Kubernetes and is called *container orchestration*.

Kubernetes architecture basics

Kubernetes is a cluster of *control plane nodes* and *worker nodes.*

Control plane nodes run all of the intelligence that implements cloud native features – the stuff that makes Kubernetes Kubernetes.

Worker nodes are where user applications run.

A non-technical analogy might be Amazon. Amazon is basically a web service and distribution service. The web service runs all the intelligence that implements things like the product catalog, searches, your shopping basket and wish lists – the stuff that makes Amazon Amazon. The distribution part is how stuff gets to users and is a bit like the Kubernetes worker nodes.

It’s vital that Kubernetes control plane nodes are high performance and highly available. As a quick example, placing three control plane nodes in different datacenter halls or cloud zones is a good practice. This means if one zone or datacenter hall experiences something like a power outage, not all control plane nodes will fail. A bad example would be putting all three control plane nodes in the same datacenter rack under a leaky air-conditioning unit and all on the same network switch and same power supply – if any of those things break, all three control plane nodes could fail.

It’s usually recommended to run 3 or 5 control plane nodes for high availability.

In summary, containers are used to run complex microservices applications and we need help managing them. Kubernetes is the most popular tool for doing this and sits in a class of tools called *container orchestrators*. The job of a container orchestrator is to *deploy* and *manage* containerised applications. Deploying includes selecting the right nodes to run containers, whereas managing includes making sure they’re resilient, scale when required, and can be easily updated and rolled back. Kubernetes clusters are made up of control plane nodes and worker nodes. The control plane nodes implement the Kubernetes intelligence, whereas worker nodes are where user apps run. Worker nodes rely on container runtimes to start and stop containers, but Kubernetes makes the overall decisions which nodes should run which containers. Kubernetes also makes the decisions to self-heal and scale etc.

Container security

There are several security risks that are particularly relevant to containers. We’ll cover the following.

- Shared kernels
- Root containers
- Unsecured networks
- Untrusted code

Containers and shared kernels

Before going any further, *kernel* is a technical term for the core functionality of an operating system. For example, the core of the Windows operating system is the Windows NT kernel, and the core of Linux operating systems is the Linux kernel. We often use the terms *operating system* and *kernel* to mean the same thing.

Namespaced containers are the most popular type of container and operate a shared kernel model. This means all containers running on the same node share the node’s kernel. As an example, 25 containers running on the same node will all share the node’s kernel. This is a big part of why containers are small and start fast, but it’s a security risk. For example, if the node’s kernel is hacked or compromised, every container on that node is also compromised. Also, a single container that gets hacked and compromised might be able to compromise the node and every other container on that node.

This means namespaced containers are inherently less secure than virtual machines.

Examples of container runtimes that create namespaced containers are Docker, containerd, runC, and CRI-O.

Virtualised containers and *sandboxed containers* were created to address the shared kernel issue. Virtualised containers run every container in its own lightweight virtual machine where each one gets its own kernel. This resolves the shared kernel issue but comes at the cost of each container being larger and slower to start. Sandboxed containers attempt to protect the host kernel without giving each container its own VM and kernel.

Other technologies exist to secure containers but are usually complex to configure. Some of these include *seccomp, capability dropping, SELinux,* and *AppArmor*.

Root containers

"Root" is the name of the most powerful user account on a Linux system. This is why running containers as *root* gives them too much power and is a huge security risk. As a result, you should avoid writing applications that need to run as the root user inside containers.

If you absolutely have to run containers as root, you should consider *user namespaces* so the root user inside a container is automatically mapped to a different user account on the shared node.

Unsecured networks

We've mentioned several times that containers run microservices apps that are highly distributed and talk to each other over the network. This makes networks a potential weak point and popular area to attack. As a result, it's vital that container-to-container communication is encrypted. A common way to do this is with a service mesh.

You should also consider Kubernetes Network Policies to restrict which microservices can communicate with other microservices.

Untrusted code

Containers evolved while the public cloud was a relatively new concept. As a result, we made a lot of mistakes as we figured things out. For example, a habit that became popular was running containers from so-called *community images.* These were images provided by unknown and un-trusted 3rd parties. Consider the following quick example.

You're leading a project to deploy a new application that implements a technology you've never worked with before. Instead of figuring out the technology yourself and building your own trusted images, you realise somebody else has already done the hard work and uploaded a "working" image to a container registry. You decide to download and use that image as it will save you a lot of effort. The obvious risk is that the community image could contain malicious code.

With this example in mind, it's vital that you only run trusted code.

In summary, cloud native security is often said to consist of four C's:

- Code
- Containers
- Clusters
- Cloud

We've just said you should always trust the *code* you run. This includes knowing where it came from and exactly what it contains and what it does. You should also ensure your *containers* are safe by not allowing root containers to run, and choosing the right container runtime. You should also mitigate the potential dangers of namespaced containers and their shared kernel model. You should also make sure your Kubernetes clusters are up-to-date with security fixes and properly secured with RBAC (user accounts and access control). Finally, you should do everything in your power to secure your datacenters and cloud infrastructure.

Figure 3.3 shows the four C's and how you normally have less and less control the further to the right you go.

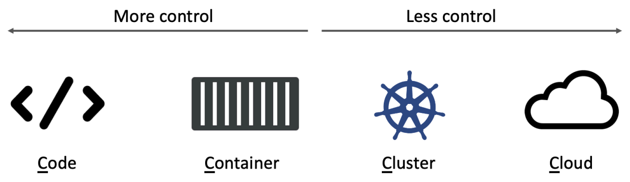

Figure 3.3

Container networking

Containers and microservices bring all of the following networking challenges.

- Increased east-west traffic
- Service discovery
- Tracing requests
- IP churn
- Security

East-west network traffic is jargon for containers in the same application talking to each other on the same networks. The other type of traffic flow is called north-south and is applications talking to external services. Consider a simple example of a microservices application with a web front-end service and a shopping cart service. Traffic between the front-end and the shopping cart containers is east-west traffic on the same network. The traffic between the front-end and external client, such as a customer's mobile phone app, is north-south and over untrusted networks such as the internet.

Microservices architectures split complex applications into lots of small applications that all talk to each other over the network. This massively increases east-west traffic and requires traffic to be encrypted.

It's also common for microservices applications to run on platforms like Kubernetes and have cloud native capabilities such as self-healing, scaling, and rollouts. All of these operations add and remove containers from the IP network and cause IP addresses to come and go a lot more frequently than most networks were designed for – this is sometimes called *IP churn*.

All of this requires a new container networking model and a new generation of networking tools.

Kubernetes and container networking.

Kubernetes implements an open flat network called the *pod network*. This is where containers run, and it spans all nodes in a Kubernetes cluster. Let's take a closer look at some of its capabilities.

The Kubernetes pod network is flat and open. That's jargon meaning all containers can see each other and can communicate with each other. This keeps things simple and makes container-to-container traffic easy. However, you should use Kubernetes network policies to control traffic flow. This includes restricting which containers and services can communicate with each other – letting everything communicate with everything is a major security risk.

Kubernetes also implements and automates service discovery. All new applications automatically have their names and IP addresses registered with internal DNS. They're also configured to use the internal DNS to find other applications. Kubernetes makes this automatic so that applications don't have to worry about how to implement it.

Kubernetes also implements IP address management (IPAM) that takes care of assigning IP addresses to new containers and garbage collecting the IP addresses of terminated and failed containers. This keep things fresh by ensuring DNS never resolves requests to the IP addresses of containers that are no longer running.

Kubernetes does not have strong capabilities for network observability and tracing. Observability is the ability to see what is going on a network and potentially troubleshoot or optimise traffic flow. Tracing is the ability to track requests as they pass between the different microservices of an application. For these capabilities you should consider a service mesh.

Finally, the Kubernetes *pod network* is implemented by a networking plugin and you can choose one that best fits your requirements. For example, not all network plugins support IPv4 and IPv6 networking. Kubernetes implements the Container Network Interface (CNI) to make the pod network pluggable. This is similar to the way it makes container runtimes pluggable with the Container Runtime Interface.

In summary, containers and microservices applications bring a new set of challenges to networking. There's a lot of east-west network traffic that needs securing and needs to be observable. Cloud native features, such as self-healing and scaling, mean containers are frequently added and removed from the network causing a lot of IP churn. This requires strong IP address management and service discovery tools. Kubernetes implements an open flat pod network that you should secure with network policies. It also provides strong

IPAM and service discovery tools. For container-to-container traffic encryption, as well as deep visibility into traffic flow, you should use a service mesh.

Service meshes

Service meshes automatically add network security, observability, and reliability to cloud native applications and are important for large-scale production environments.

A key aspect of service meshes is the ability to add these features *automatically*. They do this by implementing them at the *platform layer* instead of the application layer. This is jargon that means you just install the service mesh and every application that runs on it "just gets the features". It's a powerful model that means developers and individual applications don't need to even think about these services.

Consider a large microservices application that needs to encrypt traffic. Without a service mesh you'd need to code every microservice with the intelligence and certificates to encrypt and decrypt traffic. This adds complexity to every microservice and distracts developers from developing features that benefit customers and the business. The alternative is a service mesh that automatically does all of the encrypting and decrypting without the application even knowing it's happening – the service mesh provides the encryption "as a service".

Let's look at how service meshes are usually implemented on Kubernetes.

Kubernetes is a cluster of control plane nodes and worker nodes that run containerised applications. However, Kubernetes actually runs containers inside of another construct called a *pod*. This might sound confusing, but it's not – a pod is just a lightweight wrapper that let’s Kubernetes schedule multiple containers as a single unit. Service meshes exploit this by injecting a lightweight service mesh container into the same pod as every application container. The service mesh container transparently takes over networking in the pod and does things such as encrypting/decrypting all traffic ingressing and egressing the pod. It also exposes advanced networking telemetry that can be used to troubleshoot and optimise network performance. A key feature is that application containers are unaware of the existence of the service mesh and just operate as normal. See Figure 3.4

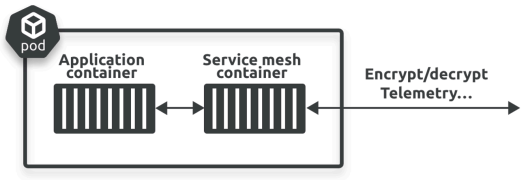

Figure 3.4

Popular service meshes include Istio, Linkerd, and Consul. Others exist, but they all provide similar features. Linkerd is a graduated CNCF project, and the Service Mesh Interface (SMI) is a specification aimed at to standardising fundamental service mesh capabilities in Kubernetes.

In summary, service meshes offer out-of-the-box network security and observability at the platform layer. This means applications can focus on being applications and not have to care about the network. They accomplish this in Kubernetes by injecting a special service mesh container into applications pods. This service mesh container handles network traffic entering and leaving the pod, enabling them to transparently secure traffic and expose advanced telemetry for troubleshooting and traffic optimisation. There are lots of service mesh options for Kubernetes and the SMI is an industry specification that defines common service mesh capabilities.

Container storage

Containers are designed to be ephemeral and stateless. Ephemeral means they’re designed to be short-lived, and if they fail you replace them with new ones instead of fixing them. Stateless means they were never designed to store data. In fact, if a container fails, all data inside it is lost.

With this in mind, containers should store data in external systems. It’s a very simple model where application code runs in containers and application data is stored outside of them. A simple cloud example running on AWS might have business application code running inside containers that store the data they generate in AWS elastic block store (EBS) volumes. This way, if any of the application containers fail, the data still exists in the EBS volumes. A similar on-premises example might have application code running inside containers and the data they generate in volumes on a shared NetApp storage system. Other clouds and storage systems exist.

This model decouples the lifecycle of data from the lifecycle of containers and allows you to pick and choose the right storage backend for each containerised application. The architecture is shown in Figure 3.5.

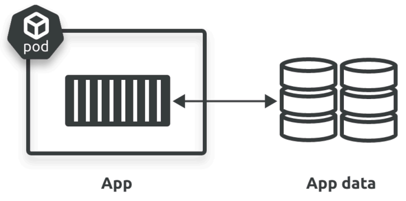

Figure 3.5

Consider the following simple example. You have a crucial business application that creates and manages very important customer data. The data needs to be protected and the app needs fast access to it. You run the business application on containers that read and write data to a high-performance storage system that replicates data offsite and takes regular snapshots. As the data is stored in an external storage system, application containers can fail and be replaced by new containers running on different nodes that can access the same data. This is an example of loosely coupling application code and logic with the application data.

Kubernetes has a pluggable storage layer based on the Container Storage Interface (CSI). It provides a common storage interface that external systems can plug into and provide their storage services to applications running on Kubernetes. For example, if you have a NetApp storage system that you want to use with Kubernetes, you'll install NetApp's CSI plugin on your Kubernetes cluster and connect the storage.

In summary, containers are designed to run small, specialised business applications and be stateless. This means you shouldn't store data in them because it disappears any time the container fails or is deleted. Instead, you use CSI plugins to mount volumes from external systems into containers. These volumes look and feel exactly the same as local volumes and apps read and write data to them as normal. This way, if, and when, containers fail or are replaced, the data survives. It's common to expose volumes into Kubernetes from enterprise-grade storage systems that provide advanced data management and data protection services. You can make block storage, file storage, and object storage available to Kubernetes.

All of this is transparent to the app that just reads and writes data.

Chapter summary

In this chapter, you learned the basics of containers and how they’re like smaller faster virtual machines that are better suited for microservices applications. Container runtimes are low-level applications that start and stop containers and are usually managed by higher-level applications like Kubernetes. Kubernetes *orchestrates* containerised applications, which is jargon for providing features such as scheduling, self-healing, and autoscaling.

You also learned that containers and microservices design patterns introduce new security threats such as shared kernels, root containers, unsecured networks, and untrusted code. Containers also provide new and unique challenges for network services such as IP address management and service discovery. They also have an impact on overall network security. Kubernetes provides native IPAM and service discovery, but you need a service mesh for container-to-container encryption and deep visibility into network traffic flow.

Finally, you learned that containers are designed to be stateless and ephemeral. This means you should always store data outside of the container in an external storage system.

Exam essentials

Container runtimes

Container runtimes are a class of infrastructure application that specialises in low-level container operations such as creating, starting, stopping, and deleting containers. They tend to operate in the background and are managed by higher-level tools like Kubernetes. For example, you or your team might deploy applications to Kubernetes on a daily basis. However, you won’t run the commands to actually start and stop containers – Kubernetes looks after all of that for you behind the scenes. Docker is the best-known container runtime, but others include containerd and CRI-O.

Containerisation

Containerisation is the process of building an application and its dependencies into a container image. For example, when a developer has written an application and packaged it as a container image you can say the developer has “*containerised* the app”.

The Open Containers Initiative (OCI)

The OCI is an open organisation that develops low-level container standards such as the container runtime-spec and container image-spec. Both have been instrumental in growing the number of container runtimes and build tools. Container images that are built to the OCI image-spec are often referred to as *OCI images.* Docker, containerd, CRI-O, and most other container runtimes work with OCI images.

Images and containers

Developers write applications and use build tools to package them as *images*. Container runtimes download images and start *containers* from them. As such, a container is a running instance of an image, and on the flip side, images are like stopped containers. However, you can start multiple containers from a single image. For example, you can download a single copy of the MySQL image and run multiple containers from it.

Container orchestration

Container orchestration tools, such as Kubernetes, automate most of the day-to-day operational aspects of deploying and managing containerised applications. For example, Kubernetes will schedule containers on nodes with enough resources to run them, scale them up and down based on demand, replace them when they fail, and manage rollouts and rollbacks.

Kubernetes

Kubernetes is the industry-standard container orchestrator. It runs as a cluster of one or more control plane nodes and one or more worker nodes. You package microservices applications as containers and run them on Kubernetes clusters. However, as you'll learn later, they have to run inside pods. Kubernetes has the intelligence to schedule each container workload to appropriate cluster nodes, it scales them when required, heals them when failures occur, and enables frequent rollouts. Control plane nodes implement the orchestration intelligence and worker nodes are where user apps run.

Container Runtime Interface (CRI)

The CRI is an open standard implemented by Kubernetes to make the container runtime layer pluggable. This means you can pick the best container runtimes for your requirements, and each worker node in a Kubernetes cluster can run a different container runtime. Docker was the original container runtime used by Kubernetes, but it's being replaced by containerd as the default on most new Kubernetes clusters.

Shared kernels

The most popular type of containers are *namespaced containers* and they all share the kernel of the node they're running on. They're the most popular and lightweight, but they're the least secure. Docker, containerd, CRI-O and many others container runtimes create namespaced containers. Virtualised containers and sandboxed containers are more secure but not as lightweight and portable.

Root containers

Don't run applications as root in your containers. This is because they can pose a significant risk to the node they run on, as well as every other container on the same node. You should design your applications so they don't need to run as root, and you should implement policies that prevent containers from running as root. If your apps absolutely have to run as root, you should implement *user namespaces* to map the root user inside the container to a non-root user on the node. This means a compromised root container won't have root access on the node.

Unsecured networks

Microservices applications are heavily distributed over multiple nodes meaning they have to share sensitive data over IP networks. This is a huge security risk if those networks are open and data is sent in plain text. With this in mind, you should encrypt all traffic between containers. One of the simplest ways to do this is by implementing a service mesh.

Untrusted code

You should only run applications that you know and trust. A simple way to accomplish this is to build and test all code yourself internally. Do not use and trust code downloaded from the internet such as community images.

Container networking: IP address management (IPAM)

What was once a large monolithic application with a single IP address that never changed is now a cloud native microservices application with 15 IP addresses. No, wait, it just scaled up and is now 21 IP addresses. No, wait, 23… Every container in a microservices application has its own IP address, and as things scale up and down, and as new versions are rolled out, IP addresses are constantly being added and removed from the network. This requires modern IPAM tools. Kubernetes provides native IPAM for highly dynamic networks.

Service discovery

Kubernetes uses a dedicated internal DNS for service registration and discovery. All new services on a Kubernetes cluster are automatically registered with DNS, and every container is configured to use this internal DNS for service discovery. This means every service on a Kubernetes cluster can find every other service.

The Kubernetes pod network

Kubernetes implements a network, called the "pod network", that all containers get connected to. It's a flat open network meaning all containers can connect to all other containers. It's also where IPAM and service discovery are implemented. However, you should use Kubernetes network policies to lock it down and control which services are allowed to communicate.

Service meshes

Service meshes are very useful in production Kubernetes environments. They provide network security and observability at the *platform layer.* This means applications can focus on being applications without having to think about securing traffic and exposing telemetry – the service mesh automatically provides all of this. Popular service mesh technologies include Istio, Linkerd, and Consul. The Service Mesh Interface (SMI) is a Kubernetes-related specification defining a standard set of service mesh resources.

Container storage

Containers are designed to be stateless. This means you shouldn't store data in them as it will be lost when the container is deleted or lost. Kubernetes implements a technology called the Container Storage Interface (CSI) that enables 3rd-party storage resources to be exposed in Kubernetes and used by applications running on Kubernetes. It makes it possible for you to expose enterprise-grade storage to apps running on Kubernetes. It also means that when containers are deleted or fail, the data is still available.

Recap Questions

See Appendix A for answers.

1. Which of the following is true of containers? Choose all correct answers.

- A. Containers are like lightweight virtual machines
- B. You cannot run user applications in containers
- C. Containers always have a dedicated kernel
- D. You normally run one application per container

2. Which of the following requires the least infrastructure to run the same number of apps?

- A. Containers
- B. Virtual machines
- C. Physical servers

3. Which is easier to package and share?

- A. Containers
- B. Virtual machines

4. You're recommending your business adopts Kubernetes and containers. Will your developers need to learn any new programming languages?

- A. Yes
- B. No

5. Which of the following best describes the contents of a container image?

- A. Operating system kernel
- B. Just application code
- C. A list of application dependencies
- D. Application code and dependencies

6. Which of the following problems does containers solve?

- A. Shared kernel
- B. Malicious code injection
- C. Works on laptop, breaks in production
- D. Encryption of network traffic

7. What is the name of the software that starts and stops containers?

- A. Service mesh
- B. Container runtime

- C. Virtual machine
- D. Sandbox

8. Which of the following is the job of a container runtime? Choose all correct answers.

- A. Download (pull) container images
- B. Choose which node to run containers on
- C. Start containers
- D. Stop containers

9. What is the role of the Open Containers Initiative (OCI)?

- A. To define Kubernetes standards
- B. To define low-level container standards
- C. To ensure containers are always open source
- D. To help ecosystem projects progress to maturity

10. What is an OCI image?

- A. A community image available on Docker Hub to be used as a standard base image
- B. Any container image complying with the OCI image-spec

11. What is the major difference between a Docker image and an OCI image?

- A. Docker images are smaller
- B. OCI images are smaller
- C. OCI images are slower
- D. There is no difference

12. Which of the following are the main types of containers? Choose 3.

- A. Sandbox containers

- B. Virtualised containers
- C. OCI containers
- D. Namespaced containers

13. Which of the following containers share the node's kernel? Choose all correct answers.

- A. Kata containers
- B. Docker containers
- C. CRI-O containers
- D. containerd containers

14. What type of containers does Kubernetes natively create?

- A. Namespaced containers
- B. Sandboxed containers
- C. Kubernetes can create namespaced, virtualised, and sandboxed containers
- D. Kubernetes can't natively create containers

15. Which of the following is the default container runtime on most modern Kubernetes clusters?

- A. containerd
- B. CRI-O
- C. Docker
- D. gVisor

16. Which of the following technologies allows Kubernetes to run different container runtimes?

- A. The OCI runtime spec
- B. The Container Runtime Interface (CRI)
- C. The Virtual Container Layer (VCL)
- D. Hypervisors

17. Which of the following is the OCI responsible for?

- A. Low-level container-related standards
- B. Making the container runtime layer of Kubernetes pluggable

18. Which of the following are jobs a container orchestrator does? Choose all correct answers.

- A. Self-healing containers
- B. Autoscaling containers
- C. Creating container images
- D. Scheduling containers to nodes

19. What is the most popular container orchestrator in use today?

- A. Docker
- B. containerd
- C. Kubernetes
- D. The CRI

20. Which of the following describes a Kubernetes cluster? Choose one.

- A. One or more control plane nodes
- B. One or more worker nodes
- C. One or more control plane and worker nodes
- D. One or more containers

21. Which of the following are potential security risks for containers? Choose all correct answers.

- A. Root containers
- B. Unsecured networks
- C. Untrusted "community" code or images
- D. Shared kernels

22. Which of the following technologies might help if you have apps that need to run in root containers? Choose one.

- A. Rootkits
- B. user namespaces
- C. pid namespaces
- D. Encrypted networks

23. What are the four Cs of cloud native security?

- A. Code, certificates, clusters, cloud
- B. Code, containers, clusters, certificates
- C. Code, containers, certificates, cloud
- D. Code, containers, clusters, cloud

24. Which of the following might be network-related challenges that come with containers? Choose all correct answers.

- A. Traffic security
- B. Service discovery
- C. IP churn
- D. Tracing requests

25. Which of the following significantly increases with microservices applications?

- A. North-south traffic
- B. East-west traffic
- C. Shaped traffic
- D. Free CPU cycles

26. Which of the following can increase IP churn (IP addresses being added and removed from the network)? Choose all correct answers.

- A. Self-healing

- B. Autoscaling
- C. Rollouts
- D. Traffic encryption

27. Which of the following is true of the Kubernetes pod network?

- A. It’s open and flat
- B. It’s open and hierarchical
- C. It’s locked-down by default
- D. It’s IPv6 only

28. What is the purpose of the CNI?

- A. Make container runtimes swappable
- B. Create standards for container runtimes and image formats
- C. Make Kubernetes networking pluggable
- D. Help cloud native technologies grow to maturity

29. What is another term for east-west network traffic?

- A. IPAM
- B. North-south traffic
- C. Service mesh
- D. Container-to-container traffic

30. Which of the following is true of service mesh technologies?

- A. They implement features at the application layer
- B. They implement features at the platform layer
- C. They implement features at the container layer
- D. They implement features at the data-link layer

31. Which of the following are popular service meshes?

- A. Kubernetes
- B. Istio

- C. Linkerd
- D. Prometheus

32. What does the CSI enable?

- A. Deep forensic analysis of crime scenes
- B. External storage systems to integrate with Kubernetes
- C. Low-level container standards such as runtimes and image formats
- D. Rootless containers

33. Which of the following technologies might help secure namespaced containers? Choose two.

- A. Seccomp
- B. SELinux/AppArmor
- C. Secure shell
- D. KubeVirt

4: Kubernetes Fundamentals

This chapter covers the topics in the *Kubernetes fundamentals* section of the exam. It accounts for 46% of your mark, making it by far the most important section of the exam. The Kubernetes-related questions are also the most detailed in the exam.

With this in mind, this chapter will go into more detail than other chapters. It also has a small set of review questions after most chapter sections. This will re-enforce what you've learned and test your knowledge without waiting until the end of the chapter. It still has the normal recap questions at the end as well.

If you're new to Kubernetes, you should be prepared to find some of this content difficult. However, it's a major part of the exam and you'll need to learn it. Be sure to:

- Take notes
- Thoroughly test yourself on all of the review questions
- Revisit any topics you struggle with

If you feel you need to learn more about Kubernetes before taking the exam, you should seriously consider the following additional resources by the author.

- **Book:** Quick Start Kubernetes
- **Video course:** Docker and Kubernetes: The Big Picture (Pluralsight)
- **Video course:** Getting Started with Kubernetes (Pluralsight)

All of these resources are short, have excellent reviews, and will help you learn the fundamentals of Kubernetes. For example, the book is only ~100 pages.

Before reading this chapter, you should be familiar with containers and microservices, as well as cloud native features such as resiliency and autoscaling. The previous chapters have covered these topics, so if you’ve completed those, you’re ready for this chapter.

The chapter is divided as follows.

- Primer
- Simple Kubernetes workflow
- Containers and pods
- Kubernetes architecture
- Scheduling
- Kubernetes namespaces
- The Kubernetes API and API server
- Kubernetes networking
- Chapter summary
- Exam essentials
- Recap questions

Primer

Google has been running Search and Gmail on billions of containers per week for a lot of years. This is massive scale and they needed help. So, they built a couple of in-house tools called Borg and Omega to help. The rest of the world encountered similar challenges when Docker made containers popular. Even though we weren't doing things at the same scale as Google, we still needed tools to help. At around the same time that Docker made containers popular, a group of technologists at Google were building a new container orchestration tool. It was based on lessons learned from Borg and Omega, and in 2014 they released it as an open-source project called "Kubernetes".

So, Kubernetes is a platform for scheduling and managing containers at scale. It brings features such as self-healing, autoscaling, and zero-downtime rolling updates. It was originally built inside of Google and released the community as an open-source project in 2014. Since then, it's become the industry-standard container orchestrator, it's the second biggest open-source project on GitHub, and it was the first project to graduate the CNCF.

Although Kubernetes is primarily a *container orchestrator* it can also orchestrate virtual machines and serverless functions. The KubeVirt project allows Kubernetes to manage virtual machines, and the Knative and OpenFaaS projects enable serverless workloads to run on Kubernetes.

Kubernetes is currently the central project at the heart of a thriving ecosystem with many other projects plugging into it, expanding its capabilities, and providing additional services. Some examples include.

- KubeVirt
- Knative
- OpenFaaS
- The Container Network Interface (CNI)
- Prometheus
- Linkerd
- Harbor

As already mentioned, KubeVirt extends Kubernetes so it can manage virtual machines, whereas Knative and OpenFaaS make Kubernetes able to manage serverless functions.

The CNI makes Kubernetes networking pluggable so that other technologies, such as Cilium, can bring advanced networking. Prometheus adds advanced monitoring, Linkerd brings service mesh capabilities, and Harbor implements a container registry. There are hundreds of other examples.

Finally, the name "Kubernetes" comes from the Greek word for helmsman. This is the person who steers a ship and is the reason the Kubernetes logo is the wheel of a ship. You'll also see the name shortened to "K8s" which is pronounced "kates". In this abbreviation, the number “8” represents the eight letters removed from the name – K~~ubernete~~s >> K8s.

Kubernetes primer quiz

1. Which of the following is a reason for deploying Kubernetes? Choose one answer.

- A. To manage on-premises networks and storage
- B. As a DevOps tool for build and test pipelines
- C. To help manage containerised applications at scale
- D. As an open-source alternative to proprietary cloud platforms

2. Which of the following features can Kubernetes bring to containerised applications? Choose all correct answers.

- A. Encrypt network traffic
- B. Self-healing
- C. Zero-downtime rolling updates
- D. Autoscaling

3. How does the KubeVirt project extend Kubernetes?

- A. Allows you to slice a single cluster into multiple virtual clusters
- B. Allows Kubernetes to schedule virtual machine workloads
- C. Lets you run serverless workloads
- D. Lets you run Kubernetes in on-premises environments

4. Which of the following community projects provides Kubernetes-native monitoring and logging?

- A. Linkerd
- B. Docker

- C. Harbor
- D. Prometheus

Answers. 1:C, 2:B,C,D, 3:B, 4:D.

Simple Kubernetes workflow

Application developers write code that gets packaged as container images and hosted in a registry.

For a container to run on Kubernetes it must be wrapped in a *pod.* This is done by describing the container in a pod manifest file that you post to the Kubernetes API server where the request is authenticated and authorised. Assuming these checks pass, the pod definition is persisted to the cluster store and the scheduler allocates the pod to a node. The Kubelet service on the assigned worker node tells its container runtime to download the required image and start it as a container.

There's quite a lot of jargon in those paragraphs and we'll explain it all throughout the rest of the chapter.

Kubernetes workflow quiz

1. True or false, Containers can be directly scheduled to Kubernetes?

- A. True
- B. False

2. Which Kubernetes node component downloads container images and starts containers?

- A. Kubelet
- B. Container runtime

- C. API server
- D. Scheduler

Answers. 1:B, 2:B.

Containers and pods

Let’s clear up some terminology before getting into the detail.

We sometimes use the terms *container* and *pod* to mean the same thing. However, a pod is actually a thin wrapper around one or more containers and is a mandatory requirement for any container that wants to run on Kubernetes. This means containers **must** be wrapped in a pod if they want to run on Kubernetes. The following extremely simple YAML file defines a pod with a single container.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: kcna-pod
5    labels:
6      project: kcna-book
7  spec:
8    containers:     <<==== Container is defined below here
9      - name: kcna-container
10       image: <container-image-goes-here>
```

Lines 1-7 give the pod and name and a label. Lines 8-10 show the container definition embedded within the overall pod definition. As the container is embedded within the pod definition, we say the pod “wraps the container”.

As well as enabling containers to run on Kubernetes, pods also augment containers with all the following features and capabilities:

- Co-scheduling
- Shared memory
- Shared volumes

- Shared networking
- Policies
- Kubernetes intelligence

Co-scheduling allows you to run multiple containers in the same pod and guarantees they'll always run on the same node. The common model is to have a main application container and a helper container. The helper container is usually called a *sidecar*. Let's consider a couple of examples.

A *logging sidecar* is a special container that runs in the same pod as an application container. It collects the application's logs and ships them to a central location for aggregation. This allows the application container to focus on running the application while the logging sidecar focuses on handling the logs.

Service meshes also use pods for co-scheduling. They inject a *proxy sidecar* into the same pod as an application container. The sidecar intercepts all network traffic and can do things such as encrypt and decrypt traffic, expose detailed network telemetry, and optimise network traffic flow. As with the logging sidecar example, the application container focusses on running the application while the service mesh sidecar focuses advanced network features.

In both cases, the application container doesn't even know the sidecar exists.

Pods are also a *shared execution environment.* This means every container in the same pod has access to the same memory, volumes, and network stack. This enables helper patterns such as a main web container and a synchroniser sidecar. In this example, the web container focusses entirely on serving static content from a volume, while the sidecar focusses entirely on syncing the latest content from a remote repository and putting it into the shared volume where the web server can access it. This works because both containers are in the same pod and have access to the same volumes. See Figure 4.1.

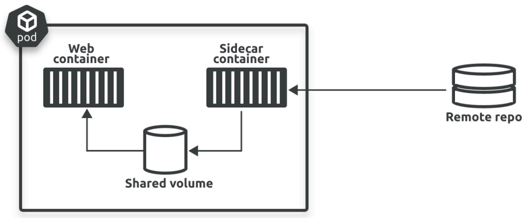

Figure 4.1

Pods also allow you to attach polices and things like resource requests and resource limits. *Resource requests* specify the minimum amount of CPU and memory a pod requires to run properly, whereas *resource limits* let you set maximum values. They're extremely important in helping the scheduler assign pods to nodes with sufficient resources.

Figure 4.2 shows two pods. The one on the left is scheduling a single container and the one on the right is scheduling two. Both containers in the pod on the right have access to the same shared memory, the same shared volumes, and the same shared network stack. This means both containers in the pod on the right share the same IP address and are accessible via dedicated ports. They're also guaranteed to be scheduled to the same worker node.

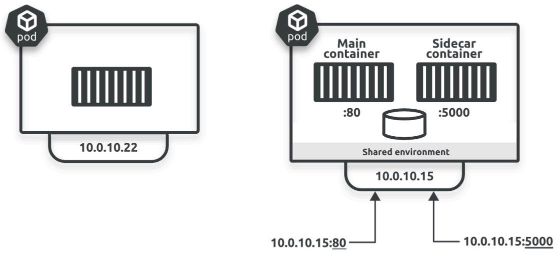

Figure 4.2

The following declarative YAML file defines a multi-container pod with some advanced features. Don't be overwhelmed if it looks complex, you don’t have to understand the detail for the exam. However, you will need to understand the concepts. It’s also been annotated to highlight the important parts.

```
1  apiVersion: v1
2  kind: Pod             <<==== This file is defining a pod
3  metadata:
4    name: kcna-pod      <<==== Pod name
5  spec:
6    volumes:            <<==== Define a volume both containers can
access
7    - name: html        <<==== Give the shared volume a name ("html")
8      emptyDir: {}      <<==== Make it an empty volume
9    runtimeClassName: level1       <<==== Schedule to a node with a
specific runtime
10   containers:                    <<==== This block defines two
containers
11     - name: main-container       <<==== This is the main app
container
12       image: <app-image>         <<==== Main app container image
13       volumeMounts:              <<==== Mount a volume into main app
container
14       - name: html               <<==== Mount the shared “html”
volume
15         mountPath: /usr/share/nginx/    <<=== Mount it here in the
container
16     - name: sidecar              <<==== This is the sidecar
container
17       image: <sidecar-image>     <<==== Sidecar container image
18       volumeMounts:              <<==== Mount a volume into the
sidecar container
19       - name: html               <<==== Mount the same shared “html”
volume
20         mountPath: /tmp/git      <<==== Mount it here in the sidecar
container
```

Lines 1-9 define the pod, whereas lines 10-20 describe two containers. As with the previous example, both containers are nested/wrapped within the pod definition.

Lines 6-8 define a new empty volume called “html”. As it’s part of the pod environment it’s accessible to both containers. Line 9 tells Kubernetes to schedule the pod to nodes with a particular container runtime. The detail isn’t important, you just need to recognise that the pod is bringing features the containers don’t natively have.

Lines 11-15 define the main application container. It mounts the shared *html* volume into its local filesystem at `/usr/share/nginx.`

Lines 16-20 define the sidecar container and mount the same shared *html* volume at `/tmp/git.`

To summarise the YAML file. It describes a multi-container pod. The pod includes a runtime class reference to help the scheduler and it defines a shared volume. Both containers mount the shared volume and can use it to share data.

There's a lot more we could say about pods, but for now, it's enough to know they're a lightweight wrapper around one or more containers and that you cannot run a container on Kubernetes without wrapping it in a pod. We often use the terms *container* and *pod* interchangeably, especially when there's no need to draw a distinction. Also, containers in a pod share the same environment and are guaranteed to be scheduled to the same node.

Containers and pods quiz

1. Which of the following is the atomic unit of scheduling on Kubernetes?

- A. Docker
- B. Container
- C. Pod
- D. Kubelet

2. Which of the following are advantages of deploying containers inside pods? Choose all correct answers.

- A. Pods allow co-scheduling of containers
- B. Pods allow multiple containers to share the same execution environment
- C. Pods give dedicated kernels to containers

- D. Pods let you apply policies to containers

3. What is a common name for a helper container?

- A. Sidecar
- B. Sideshow
- C. service mesh
- D. Supporter

4. What format are Kubernetes manifest files usually written in?

- A. Python
- B. YAML
- C. XML
- D. Rust

Answers. 1:C, 2:A,B,D, 3:A, 4:B.

Augmenting pods

You’ve just learned that pods wrap one or more containers and add capabilities. However, pods don’t self-heal, they don’t autoscale, and they don’t do things like zero-downtime rollouts. Kubernetes has higher-level controllers that wrap around pods and implement these features. As a result, you’ll almost always deploy pods via higher level controllers such as *deployments* and *statefulsets.*

You’ll learn more about controllers later in the chapter. But for now, let’s consider a simple example of deploying a stateless web server.

Your development team writes a stateless web app and packages it as a container. You know you need 5 instances to meet expected demand, and as you’re running a Kubernetes environment you need to wrap each container in a pod. However, pods aren’t resilient. For example, if you deploy 5 standalone pods and a node hosting 2 of them fails, there's no intelligence to recreate them and ensure you always have 5. This is where higher-level controllers come into play.

Kubernetes has a *deployment controller* that accepts a pod definition and ensures a desired number are always running.

With this in mind, you wrap your pod in a deployment and tell Kubernetes to run that. Once all 5 pods are up and running, the deployment controller keeps a watch on things and responds to events such as failures. For example, if a node fails and takes two of the pods with it, the deployment reacts by starting two new pods to keep you at 5.

You'll learn more about controllers later, but the following YAML shows a simple deployment. Notice how the container is nested within the pod, which in turn is nested within the deployment. As such, the deployment wraps the pod, which in turn wraps the container.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: kcna-deployment
5  spec:
6    replicas: 5
7    <Snip>
8    template:
9      spec:
10       runtimeClassName: level1
11       containers:
12       - name: kcna-pod
13         image: <container-image>
```

Lines 1-6 are the deployment and define the desired state of 5 pod replicas always running. Lines 8-10 define the pod and include the runtime class requirement you saw before. Lines 11-13 define the container. This is shown in Figure 4.3.

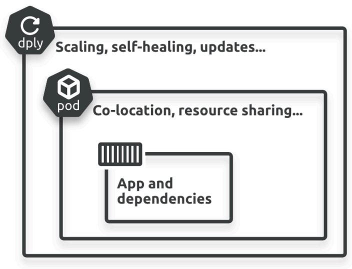
dply
Scaling, self-healing, updates...
pod
Co-location, resource sharing...
App and dependencies

Figure 4.3

Augmenting pods quiz

1. Why does Kubernetes implement higher-level controllers instead of deploying static pods?

- A. They provide advanced network telemetry
- B. They add features such as resiliency, scaling, and updates
- C. They allow pods to be migrated between different cloud platforms
- D. They run pods faster

2. Which of the following Kubernetes controllers ensures a specific number of containers are always running?

- A. Deployment controller
- B. Job controller
- C. Endpoints controller
- D. Cloud controller

3. Which of the following best describes the architecture of pods and controllers?

- A. Pod specifications are wrapped in a controller specification
- B. Controller specifications are embedded inside pod specifications

Answers. 1:B, 2:A, 3:A.

Kubernetes architecture

We'll cover a lot of things in this section, so we'll split things out into sub-sections to make things easier to digest and refer back to. We'll cover all of the following:

- High level Kubernetes architecture
- The control plane
- Controllers
- Worker nodes
- kubectl
- Hosted Kubernetes

High level Kubernetes architecture

At a high level, Kubernetes is a cluster of nodes that run containerised applications. It runs a number of *controllers* that implement features such as self-healing and automated releases.

There are two types of nodes in a Kubernetes cluster.

- Control plane nodes
- Worker nodes

Both can be virtual machines, physical servers, or cloud instances, and you can mix-and-match different types in the same cluster.

Control plane nodes are the brains of Kubernetes. They run the API server, the scheduler, and the cluster store. They also run the controllers that implement the intelligence.

Worker nodes are where user applications run. Every worker node has a kubelet and a container runtime. The kubelet talks to the control plane and tells the container runtime when to start and stop containers.

The Kubernetes command line utility is called `kubectl`.

Let's look at everything in more detail.

The control plane

The Kubernetes control plane is a collection of specialised independent services that run on every control plane node. These implement the intelligence that makes Kubernetes so special.

As the control plane is a vital part of every Kubernetes cluster, it needs to be high performance and highly available. In the real world, production Kubernetes clusters normally run 3 or 5 control plane nodes spread across multiple failure domains. This means failure of a control plane node, or any of the supporting infrastructure, won't take the entire control plane down. You should also ensure control plane nodes are connected by low-latency reliable networks.

It's important that you run an odd number of control plane nodes, and running 1 is better than running 2. This is because an even number has more chance of resulting in a split-brain scenario where a network failure occurs and there are an equal number of control plane nodes on either side of the network failure. If this happens, neither side can be sure they have a majority.

If the control plane ever does go down, applications that are already running will continue to work but you won't be able to make any changes or updates.

The major control plane services include all of the following, and they all run on every control plane node.

- The API server
- The scheduler
- The cluster store
- Controllers

The API server is the main gateway in and out of Kubernetes. When you send commands to Kubernetes, you send them to the API server. When you query the state of something, you send the query to the API server. Also, the kubelet on every worker node watches the API server for new work assignments and reports status back to the API server. Basically, everything in Kubernetes goes through the API server. This requires high-performance control plane nodes so the API server can respond to requests quickly.

The scheduler is responsible for deciding which worker nodes to run pods on. It can schedule pods to nodes with enough CPU and memory, it can distribute them across availability zones, and it can even target them to particular nodes or keep them away from particular nodes.

The cluster store is where the state of the cluster is stored and is the only stateful component of the entire control plane. This means it's the only component that creates and stores important cluster-related data that must not be lost. For example, if you deploy an application that requires 5 instances of a particular pod, this will be stored in the cluster store and used by controllers and other components as a reference. The cluster store is based on the *etcd* distributed database.

Controllers watch the current state of the cluster and take actions to ensure it's always in sync with desired state. Again, as a simple example, you might request 5 pods running a web server container – this is desired state. If something fails and drops the number of pods down to 3, a controller observes the change and starts 2 new pods to bring observed state back into sync with desired state. We'll explain this in more detail shortly.

As a quick recap, the control plane runs critical services and needs to be highly available. For most scenarios you should run 3 control plane nodes and spread them across different availability zones so that local failures don't impact all three. You should also ensure control plane nodes are high-performance so that requests to the API server aren't slow.

Controllers

Kubernetes runs a set of specialised controllers that implement most of the intelligence that makes Kubernetes so powerful. Each one runs as *reconciliation loop* that watches the cluster and ensures *observed state* is in sync with *desired state.*

We'll briefly mention some of the more important controllers and then walk through an example.

Deployment controller. The Deployment controller implements the intelligence and features required to run stateless apps on Kubernetes. For example, it can ensure 5 replicas of a web service are always running. If one fails, the controller starts a new one. It also implements the intelligence for zero-downtime rolling updates and more.

StatefulSet controller. The StatefulSet controller is similar to the deployment controller but it specialises in **stateful** workloads. For example, it deploys and self-heals a desired number of pods just like the deployment controller does. However, it also implements things like *ordered startups* and *ordered shutdowns* that might be important to stateful apps that write to shared data stores.

DaemonSet controller. The DaemonSet controller ensures a single instance of a particular pod is always running on every node of the cluster. A common example is logging, where it's common to need an agent running on every cluster node to collect and ship logs. The DaemonSet controller will ensure an instance of the agent pod runs on every node in the cluster. This includes automatically starting pods on any new nodes added to the cluster. It can also restart or replace pods that fail.

There are a lot more controllers, but they all operate as *reconciliation loops* that watch the cluster and make sure *observed state* matches *desired state.*

Consider the following example. You have a new app that needs 5 stateless web pods running v1.3 of your code. You describe this requirement in a declarative manifest file that you post to the API server. This gets recorded in the cluster store as your *desired state* and the scheduler assigns the 5 pods to worker nodes. The deployment controller operates on the control plane as a background reconciliation loop making sure observed state matches desired state – in this example it's making sure the 5 web pods are always running. If observed state matches desired state it just keeps watching the cluster. However, if a node running one of the pods fails, the deployment controller will observe 4 web pods and realise it doesn't match the desired state of 5. It will fix this by instructing the scheduler to start a new pod on a surviving node. This will bring observed state back into sync with desired state and is an example of *resiliency* or *self-healing.*

Sticking with the same example, assume you want to update all 5 web pods from v1.3 to v1.4. You do this by updating the same declarative manifest file to specify the new version and post it to the API server again. This registers a new desired state of 5 web pods running v1.4. The deployment controller will observe 5 web pods running v1.3 and compare that to desired state which now wants 0 pods running v1.3 and 5 pods running v1.4. It will respond by replacing the 5 pods running v1.3 with 5 running v1.4. This is an example of a rollout.

Jargon busting. We often say that controllers operate as *"control loops", "watch loops"* or *"reconciliation loops"*. They all mean the same thing. We also use the terms *"current state"*, *"actual state"* and *"observed state"* to mean the same thing. Finally, the process of bringing current state back into sync with desired state is called *reconciliation*.

Worker nodes

Worker nodes are where user applications run. They can by physical servers, virtual machines, or cloud instances, and a single Kubernetes cluster can run a mix of any.

All workers run three important services:

- Kubelet
- Container runtime
- Kube-proxy

The kubelet is the main Kubernetes *agent* on the node. It watches the API server for new work assignments, tells the container runtime which containers to start and stop, and reports status info back to the API server. It also does other things such as making container logs available to the `kubectl logs` command.

Older Kubernetes clusters used Docker as their container runtime. However, that layer is now pluggable, meaning cluster administrators can choose the best container runtime for their requirements. At the time of writing, most new Kubernetes clusters use containerd (pronounced "container dee") as their container runtime but many others exist. In fact, a single cluster can run different runtimes on different worker nodes as shown in Figure 4.4.

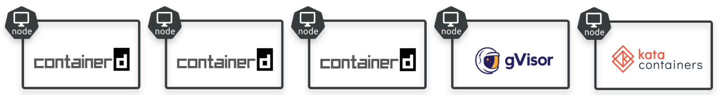

Figure 4.4

The technology making the container runtime layer pluggable in Kubernetes is the *Container Runtime Interface (CRI).* The following list is some of the more common container runtimes.

- Containerd (popular on new clusters)
- CRI-O (default on RedHat OpenShift clusters)
- gVisor (provides additional isolation between container and host OS)
- Kata containers (runs every container in a lightweight VM)
- Docker (the original, but deprecated since K8s 1.20)

The final piece of the worker node is the kube-proxy. This implements networking so that traffic can successfully reach pods.

kubectl

kubectl is the Kubernetes command-line tool and is available for most platforms, including Linux, MacOS, and Windows.

You install it on a machine, configure its config file, and use it to manage remote Kubernetes clusters. It can do all of the following:

- Get cluster information
- Deploy objects such as pods and services
- Query the state of objects
- Read logs
- Execute commands inside of containers

It has a configuration file called `config` that's normally located in your home folder in a hidden directory called `.kube`. You'll often hear it referred to as your *kubeconfig* file, and it contains a list of known clusters and user credentials. For example, if you have two clusters, they'll both be listed in your kubeconfig file. As will any usernames and credentials required to connect to the clusters and manage them.

The following are common kubectl commands:

- `kubectl get <object>`
- `kubectl describe <object>`
- `kubectl apply <filename>`
- `kubectl delete <filename | object>`
- `kubectl api-resources`

`kubectl get` commands return brief information about a particular resource type. For example, the following command returns info about a pod called web-fe.

```
$ kubectl get pod web-fe
NAME       READY   STATUS    RESTARTS   AGE
web-fe     1/1     Running   0          29m
```

The following `kubectl describe` command returns more detailed information about the same pod.

```
$ kubectl describe pod web-fe
Name:          web-fe
Namespace:     default
Priority:      0
Node:          prod-cluster/192.168.65.4
Start Time:    Wed, 9 Feb 2022 21:43:01 +0000
Labels:        project=kcna-book
Annotations:   <none>
Status:        Running
IP:            10.1.0.107
<Snip>
```

The preferred way to deploy objects to Kubernetes is to describe them in a YAML manifest file and deploy them with `kubectl apply` or `kubectl create`. The following command deploys the objects defined in a file called `web-deploy.yml`.

```
$ kubectl apply -f web-deploy.yml
deployment.apps/kcna-deploy created
```

The `kubectl delete` command deletes objects. You can give it an object type and name, or you can give it a manifest file. If you give it an object type and name, it will delete that object. If you give it a YAML manifest file, it will delete everything defined in the file. See the following examples.

```
$ kubectl delete pod web-fe
pod "web-fe" deleted

$ kubectl delete -f web-deploy.yml
deployment.apps "kcna-deploy" deleted
```

The following command displays a list of all resources you can deploy on your cluster.

```
$ kubectl api-resources
NAME            APIVERSION      NAMESPACED   KIND
pods            v1              true         Pod
deployments     apps/v          true         Deployment
daemonsets      appsv1          true         DaemonSet
jobs            batch/v1        true         Job
<Snip>
```

It’s important that your version of `kubectl` is no more than one minor version older or newer than your Kubernetes API server. For example, if you’re running Kubernetes 1.23, you can use `kubectl` version 1.22 or 1.24.

Hosted Kubernetes

Most cloud platforms offer a *hosted Kubernetes* service. This is where the cloud platform owns and manages the Kubernetes cluster, and you pay to use it.

Many hosted Kubernetes services own and manage the control plane but let you manage worker nodes. In this model, the cloud platform is responsible for the performance and availability of the control plane, as well as performing upgrades. You’re responsible for adding and removing worker nodes and configuring node pools.

Hosted Kubernetes is very popular as it removes the pain and lead-time of building and managing your own high-performance highly-available cluster. However, costs that might initially seem small can easily become big if not managed well.

Popular hosted Kubernetes services include:

- AWS Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Google Kubernetes Engine (GKE)
- Linode Kubernetes Engine (LKE)

Others also exist.

Kubernetes architecture quiz

1. Which open-source distributed database is the cluster store based on?

- A. Consul
- B. Oracle
- C. MongoDB
- D. etcd

2. Which of the following container runtimes is default on Red Hat OpenShift clusters?

- A. CRI-O
- B. Docker
- C. containerd
- D. kata

3. Which of the following statements is true of recent releases of Kubernetes?

- A. containerd is unsupported
- B. containerd is deprecated
- C. The Docker runtime is default
- D. The Docker runtime is deprecated or unsupported

4. What is a common name for the kubectl configuration file?

- A. kubernetes.config
- B. kubeconfig
- C. kubectl.yml
- D. Kubelet

5. What is the job of the Kubernetes API server?

- A. To serve the API over HTTPS
- B. The schedule work to cluster nodes
- C. To start and stop containers
- D. To define Kubernetes objects

6. Which of the following are Kubernetes control plane services? Choose two.

- A. Docker
- B. Kubelet
- C. API server
- D. Scheduler

7. Which of the following are types of Kubernetes cluster node? Choose two.

- A. Worker nodes
- B. Secondary nodes
- C. Control plane nodes

- D. Primary nodes

8. What is hosted Kubernetes?

- A. Where your cloud platform owns and manages the Kubernetes control plane
- B. All Kubernetes clusters that run on a public cloud platform
- C. A name for on-premises Kubernetes clusters
- D. Lightweight Kubernetes distributions running on IoT and edge devices

9. Which Kubernetes controller manages workloads that work with persistent data?

- A. CronJob
- B. Job
- C. StatefulSet
- D. Deployment

10. Which of the following best describes how Kubernetes controllers operate?

- A. As key-value stores
- B. Client-server architecture
- C. As RESTful objects exposed over HTTPS
- D. As background reconciliation loops

Answers. 1:D, 2:A, 3:D, 4:B, 5:A, 6:C,D, 7:A,C 8:A, 9:C, 10:D.

Scheduling

Kubernetes has a built-in scheduler that runs as part of the control plane. It uses advanced logic to schedule pods to the right worker nodes.

Scheduling starts when Kubernetes is asked to run a new pod. This might be you sending a new manifest file to the API server that asks for a new pod, it might be the result of an autoscaling event, or it might even be a self-healing action replacing a failed pod. Either way, as soon as a new pod is requested, it goes into the pending state while the scheduler picks the best node to run it. As soon as a node is identified the pod is scheduled. However, if the scheduler can't find a suitable node, the pod will stay pending.

Consider the following example. You're running a Kubernetes cluster with 6 worker nodes and all of your pods have been configured with resource requests and resource limits – resource requests tell the scheduler the minimum amount of CPU and memory a container needs in order to run, whereas resource limits set maximums. Assume you're scheduling two new pods and each one requires 2GB of memory and 2 CPU cores. If the scheduler can't find a worker node with enough resources, they'll go into the pending state until a suitable node becomes available. As soon as a suitable node becomes available they'll be scheduled.

The scheduler can also do advanced things such as:

- Affinity and anti-affinity
- Pod topology spread constraints
- Match workloads with container runtimes
- Schedule work to nodes with the right storage

Affinity rules let you schedule pods onto particular nodes, or nodes that are running particular pods. *Anti-affinity* does the opposite – it lets you tell the scheduler not to run pods on particular nodes, or nodes already running particular pods. Think of affinity and anti-affinity rules as being like magnets that can either attract or repel pods from particular nodes – affinity rules attract pods to particular nodes, whereas anti-affinity rules repel pods from particular nodes.

Pod topology spread constraints let you balance pods across user-defined failure domains. They're an important tool for achieving application high availability.

The scheduler can also target pods at nodes with specific container runtimes. For example, you can tell Kubernetes that certain workloads should only run on nodes with the gVisor runtime. It can also schedule containers to nodes with access to certain storage volumes.

Scheduling quiz

1. What is the job of the Kubernetes scheduler?

- A. To assign work to worker nodes
- B. To run batch jobs at specific times
- C. To maintain accurate time on the control plane
- D. To keep observed state in sync with desired state

2. What happens to a pod if the scheduler can't find a worker node to run it?

- A. The pod will be rejected and won't run
- B. The pod will sit in the pending state
- C. It will temporarily run on a control plane node

Answers. 1:A, 2:B.

Kubernetes namespaces

Kubernetes namespaces are a way to divide a single Kubernetes cluster into virtual clusters called *namespaces.*

Each namespace can have its own set of resource quotas and user accounts. This makes them a good way to share clusters among different teams or environments. For example, you might use namespaces to divide a single cluster into dev, test, and qa namespaces. However, you shouldn’t use namespaces as a workload boundary to try and isolate hostile or potentially dangerous workloads. For example, namespaces cannot prevent a compromised container in one namespace from affecting containers in other namespaces. For example, a compromised container in the Pepsi namespace can easily take down containers in the Coke namespace. Currently, the only way to guarantee one workload won’t impact another is to put them on separate clusters.

Kubernetes namespaces quiz

1. Which of the following is a good use-case for Kubernetes namespaces?

- A. Dividing a cluster among competing workloads
- B. Assigning friendly names to Kubernetes clusters
- C. Combining multiple Kubernetes clusters under a single DNS name for simpler management
- D. Dividing a cluster among different non-competing departments

Answers. 1:D.

The Kubernetes API and API server

If you’re new to the concept of APIs, you should think of the *Kubernetes API* as a catalog listing every possible Kubernetes object and their properties. This catalog (API) is accessed via the *API server.*

For example, Kubernetes defines a lot of objects such as pods, deployments, replicasets, statefulsets, cronjobs, services, ingresses, network policies, and more. All of these are defined in the API, along with all of their properties that can be used to configure them. When you deploy an object, such as a deployment, you define it in a declarative manifest that you send to the *API server* where it's authenticated, authorised, and scheduled to the cluster.

If you try to deploy an object that isn't defined in the API, the operation will fail. Also, if you try and configure a property of an object that no longer exists, it will fail.

The Kubernetes API is divided into named groups to make it easier to understand, navigate, and expand. For example, workload-related objects, such as deployments and statefulsets, are defined in the "apps" API group, whereas storage-related objects are defined in the "storage.k8s.io" API group.

New features enter the API as *alpha* features, progress through *beta*, and eventually graduate to *stable*. We sometimes refer to stable features as *generally available (GA)* or *v1*. Stable objects can also be *v2, v3* etc.

The following `kubectl` command shows a couple of the objects in the Kubernetes API. The deployments object is a stable v1 object in the *apps* named group. CSIStorageCapacity is a beta object in the storage.k8s.io named group.

```
$ kubectl api-resources
NAME                    APIVERSION                NAMESPACED   KIND
deployments             apps/v1                   true         Deployment
csistoragecapacities    storage.k8s.io/v1beta1    true
CSIStorageCapacity
<Snip>
```

Kubernetes has a deprecation policy that guarantees all stable objects will be supported for at least 12 months, or two releases (whichever is longer), after deprecation.

At the time of writing, a new Kubernetes version is released every 4 months, resulting in three releases per year. For example, v1.21, v1.22, and v1.23 were all released in 2021.

The *API server* exposes the API over a RESTful HTTPS endpoint. All requests to the API server are subject to authentication (authN) and authorisation (authZ).

API and API server quiz

1. Where are Kubernetes objects defined?

- A. The cluster store
- B. The API
- C. Individual controllers
- D. The Kubernetes GitHub repo

2. Which Kubernetes API group are workload objects, such as deployments and statefulsets, defined?

- A. apps
- B. core
- C. workloads

3. Which of the following are other ways to refer to a v1 API object? Choose two.

- A. Stable
- B. Beta
- C. GA
- D. Live

4. How long does Kubernetes guarantee stable objects will be supported for?

- A. At least one month following deprecation

- B. At least 6 months following deprecation
- C. At least 12 months following deprecation
- D. At least 18 months following deprecation

Answers. 1:B, 2:A, 3:A,C, 4:C.

Kubernetes networking

In this section we'll cover the following network-related topics:

- Kubernetes services
- The pod network
- Service registration and discovery

Kubernetes services

You already know that pods are typically deployed via higher-level controllers like the deployment controller and the statefulset controller. These implement cloud native features such as self-healing, autoscaling, rollouts, and rollbacks.

These features make individual pods extremely unreliable. Consider the following examples.

Every time a node or pod fails, the missing pod is replaced with a new pod with a new IP address. If a client was connecting directly to the failed pod, future connections will time-out and won't re-establish to the new pod. Scale-up events add new pods with new IP addresses, whereas scale-down events remove pods. Again, clients connected to a pod that is removed as part of a scale-down operation will lose their connection. Finally, rolling out a new version of an app removes all existing pods and replaces them new ones with new IP addresses.

All of this makes pods unreliable and other apps should never connect directly to pods.

To resolve this issue, Kubernetes provides a *service* object that sits in front of pods and provides them with reliable networking. For example, you define a service object and tell it to redirect traffic to all pods with a particular label. The service gets a stable name and IP address that Kubernetes guarantees will never change. Clients connect to the service, and the service redirects the traffic to the pods. The service is also intelligent enough to know when pods are added or removed, so it guarantees to always load-balance the traffic to the latest set of running pods.

Figure 4.5 shows a service called *kcna-svc* with an IP address of 10.0.0.10. The name and IP are stable, and clients should connect to these instead of directly to pods. All connections hitting the service are redirected to a healthy pod with the `project=kcna-book` label. The service makes sure it always knows when pods are added and removed from the network.

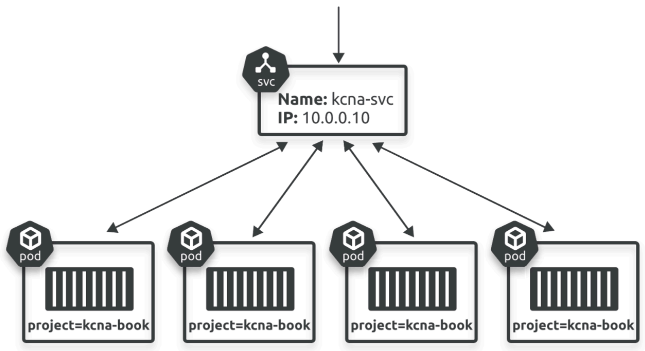

Figure 4.5

The pod network

Every Kubernetes cluster implements a network called the *pod network* that all pods run on. It's a large, flat, open network, meaning all pods can communicate with all other pods. This makes deployment easy but is a major security risk. As a result, you should implement Kubernetes network policies to control pod-to-pod communication.

Behind the scenes, it’s actually a network plugin that implements the pod network. Kubernetes uses the *Container Network Interface (CNI)* to expose a pluggable network layer, and 3rd parties create CRI plugins that actually implement the pod network. This has resulted in a lot of innovation and a strong Kubernetes networking ecosystem. Some of the more well-known CRI plugins include:

- Cilium
- Canal
- Flannel
- Weave

Cilium is open-source, based on eBPF, and it’s currently the most popular CRI plugin for Kubernetes.

Service registration and discovery

Kubernetes also implements native service registration and service discovery based on DNS. *Service registration* is the act of registering the name and IP of new services with DNS, whereas *service discovery* is the act of using a service name to learn its IP address.

> **Clarification.** When saying “the name and IP of new services gets registered with DNS” we mean the name and IP address of *Kubernetes service objects* that sit in front of pods. These were explained in an earlier section.

Kubernetes service discovery operates on two core principles:

1. All new services are automatically registered with DNS

2. All pods and containers are automatically configured to use DNS for discovery

To make this work, every Kubernetes cluster runs an internal DNS service called the *cluster DNS.* This runs as a Kubernetes-native application that watches the API server for new service objects. When it sees one, it automatically registers its name and IP in the cluster DNS. This means services don't need to implement any logic to register themselves – the cluster DNS is always watching the cluster and registers all services automatically.

Also, every container on Kubernetes is automatically configured to use the cluster DNS when it needs to discover new services. This means apps running in containers don't need to implement any special logic to find other services – the containers they're running in do all of that automatically.

Kubernetes networking quiz

1. Which Kubernetes object provides stable networking for pods?

- A. LoadBalancer
- B. Ingress
- C. Service
- D. The pod network

2. How is the Kubernetes pod network implemented?

- A. By an external physical network
- B. By assigning worker nodes IPv6 addresses
- C. Via 3rd-party CNI plugins
- D. Via a service mesh

3. How does Kubernetes implement service registration and discovery?

- A. Via a service mesh
- B. CNI
- C. IPv6
- D. DNS

Answers. 1:C, 2:C, 3:D.

Chapter summary

Kubernetes is a container orchestrator originally developed at Google and open-sourced and donated to the CNCF in 2014. It was the first project to graduate the CNCF and is now the industry-standard orchestrator and second biggest project on GitHub. It can also orchestrate virtual machines and serverless functions.

A Kubernetes cluster consists of one or more control plane nodes and one or more worker nodes. Control plane nodes run the API server, cluster store, scheduler, controllers, and the API itself. Worker nodes are where user applications run and have a kubelet, container runtime, and kube-proxy.

The API server exposes the API over a RESTful HTTPS interface and requires high-performance control plane nodes. All internal and external Kubernetes traffic goes through the API server. The cluster store is the only stateful part of the control plane and is where the state of the cluster and applications is persisted. The scheduler assigns pods to worker nodes, and controllers ensure the observed state of the cluster always matches desired state.

Every worker node runs a kubelet that watches the API server for new work and tells the container runtime which containers to start and stop. The kube-proxy implements networking rules to ensure traffic to Kubernetes services reaches the right pods and containers.

Namespaces are a great way to divide a single cluster into virtual clusters for sharing between different teams and departments. Every namespace can have its own resource quotas and user accounts, but you shouldn’t use them as workload boundaries.

Finally, Kubernetes uses 3rd party plugins to implement an open flat pod network that you should lock down with network policies.

Exam essentials

Kubernetes

Kubernetes is the industry-standard orchestrator for containerised apps. It was initially developed at Google and is now an open-source project under active development on GitHub. It’s deployed as a cluster of control plane nodes and worker nodes that provide scheduling, self-healing, autoscaling, and more. Third-party projects have extended its capabilities to include orchestration of virtual machines, serverless functions, and much more.

Pods

Pods are the atomic unit of scheduling on Kubernetes. Developers write applications, package them as containers, and then wrap those containers in pods to run on Kubernetes. A Pod can wrap one or more containers, and all containers in a pod are guaranteed to be scheduled on the same node. The pod also acts as a shared execution environment for containers. This means two containers in the same pod have access to the same volumes and networking. If you want to scale an app, you add more pods.

Sidecar containers

The multi-container pod model usually has a main application container and a helper container. The helper container is called a *sidecar* and has enabled powerful patterns and innovations. For example, service mesh technologies work by injecting a sidecar container into the same pod as applications containers. The sidecar can intercept all network traffic and transparently encrypt and decrypt it. It can also expose rich network telemetry that can be used to better understand and optimize traffic flow. Many other solutions use sidecars.

Control plane nodes

Control plane nodes are where all control plane services run. In fact, every control plane service runs on every control plane node for high availability. They can be physical servers, virtual machines, and even cloud instances, and you should run either 3 or 5 for high availability. You should also spread them across failure domains. If you’re running a hosted Kubernetes cluster, the control plane nodes will be managed by your cloud platform.

The Control plane

The control plane is the collection of services that comprise the intelligence of Kubernetes. The services include the API server, scheduler, cluster store, API, and controllers. Each service is highly specialised and does one thing. All control plane services run on all control plane nodes.

API server

The API server exposes the Kubernetes API as a RESTful endpoint over HTTPS. It’s a service that runs as part of the control plane and is the central hub of Kubernetes communication. For example, all commands and requests issued to Kubernetes are sent to the API server. Even internal control plane services can only send messages to each other via the API server.

Scheduler

The scheduler is a control plane service that assigns pods to worker nodes. It implements advanced logic that let it make intelligent decisions about which nodes are the best choice for pods.

Cluster store

The cluster store is the only stateful part of the control plane and is where the state of the cluster and apps is kept. It’s usually based on the **etcd** distributed database, and a copy runs on each cluster node.

Controllers

Controllers run as part of the control plane as background reconciliation loops that ensure the observed state of the cluster matches desired state. That’s jargon for making sure what is running on the cluster is what you asked for. Kubernetes implements several controllers, and each one is highly specialised. Common controllers include the deployment controller, the statefulset controller, the DaemonSet controller, and the job controller.

Deployment controller

The deployment controller is probably the most widely used Kubernetes controller. It deploys and manages stateless workloads and provides cloud native features such as resiliency, autoscaling, and zero-downtime rolling updates.

StatefulSet controller

The StatefulSet controller manages stateful workloads and provides cloud native features such as resiliency, autoscaling, and updates. It also implements things like ordered startups, ordered shutdowns, and persistent naming. These might be useful to stateful apps.

DaemonSet controller

The DaemonSet controller can be used to guarantee an instance of a particular pod will always be running on every cluster node. If new nodes are added to the cluster, the DaemonSet controller will notice this and automatically start an instance of the pod on the new node. A typical use case is ensuring a logging agent is running on every cluster node.

Worker nodes

Worker nodes are clusters nodes where user applications run. They don’t run any control plane services, but they do run a kubelet, container runtime, and kube-proxy. You should spread worker nodes across failure domains to help the scheduler keep applications highly available.

Kubelet

The kubelet is the main Kubernetes agent running on every worker node. It watches the API server for new pods to run and reports back on status. It also makes container logs available to the `kubectl logs` command.

Container runtime

Container runtime software takes instructions from the local kubelet to start and stop containers on the local node. Older Kubernetes clusters always used Docker as the container runtime, but newer clusters implement the Container Runtime

Interface that lets administrators choose the best container runtimes to suite their requirements.

Kube-proxy

Kube-proxy is a service on every cluster node that implements the networking rules that enable traffic sent to services to reach pods.

kubectl

`kubectl` is the Kubernetes command line utility. It's available on many operating systems and platforms.

Kubeconfig

The kubeconfig file tells `kubectl` which clusters it can talk to and which user accounts to authenticate with. It's normally just called `config` and lives in the hidden `.kube` folder of you home directory.

CRI

The Container Runtime Interface (CRI) is a low-level technology implemented by Kubernetes to make the container runtime layer pluggable. It enables cluster admins and cluster operators to choose the right container runtime for their applications and environment.

containerd

containerd (pronounced "container dee") is the default container runtime on most modern Kubernetes clusters. It's a stripped-down version of Docker with only the bits that Kubernetes needs. The code for containerd was donated to the CNCF by Docker, Inc. It's a graduated CNCF project and stable container runtime.

Namespaces

Namespaces let you split a Kubernetes cluster into multiple virtual clusters called "namespaces". Each namespace gets its own set of user accounts and resource quotas, and they're a good way to share a cluster among different teams or departments in the same organisation. However, they're not a secure workload boundary, so you shouldn't use them to isolate production environments from test or dev. You also shouldn't use them to divide a cluster between competing clients. This is because compromised containers in one namespace can easily impact containers in other namespaces.

Pod network

Every Kubernetes cluster implements a special network called the *pod network.* This is the network all pods and containers are deployed to, and it’s always implemented by a 3rd party network plugin. Lots of plugins exist, and they all plugin via the Container Network Interface (CNI). Every Kubernetes cluster has exactly one pod network.

CNI

The Container Network Interface (CNI) is a low-level technology that allows 3rd-parties to build the pod network with advanced technologies and features. For example, some CNI plugins implement dual stack IPv4 and IPv6 networking, whereas others might only implement IPv4.

Recap questions

Some of these questions may be similar to the section review questions from earlier in the chapter. This isn’t a problem, and repetition helps you learn and remember things.

See Appendix A for answers.

1. What year was Kubernetes released to the community as an open-source project?

- A. 2020
- B. 1970
- C. 2018
- D. 2014

2. Which of the following features can Kubernetes add to containers? Choose all correct answers.

- A. Self-healing
- B. Autoscaling
- C. Automated image scanning
- D. Zero-downtime rolling updates

3. Which of the following workload types can Kubernetes orchestrate? Choose all correct answers.

- A. Containers
- B. Virtual machines
- C. iOS apps
- D. Serverless functions

4. Which of the following projects enables Kubernetes to orchestrate virtual machine workloads?

- A. OpenFaaS
- B. KubeVirt
- C. Cilium
- D. Knative

5. Which two projects allow serverless functions to run on Kubernetes?

- A. OpenFaaS
- B. KubeVirt
- C. Cilium
- D. Knative

6. Which of the following projects implements a service mesh on Kubernetes?

- A. Prometheus
- B. Linkerd
- C. OpenFaaS
- D. Harbor

7. Which of the following best describes a Kubernetes pod?

- A. The Kubernetes name for a Docker container
- B. A controller running on the control plane keeping observed state in sync with desired state
- C. A network abstraction for load-balancing
- D. A thin wrapper around one or more containers

8. Which of the following is true?

- A. Pods automatically provide service mesh capabilities to containers
- B. Containers run slower if they’re wrapped in a pod
- C. Containers run faster if they’re wrapped in a pod
- D. Containers cannot run on Kubernetes unless they’re wrapped in a pod

9. Which of the following do pods provide for containers? Choose all correct answers.

- A. Co-scheduling of multiple containers
- B. Shared memory
- C. Shared volumes
- D. Shared images

10. Which of the following is true about containers running in the same pod? Choose all correct answers.

- A. They’ll always be scheduled to the same worker node
- B. They’ll have access to the same volumes
- C. They’ll all use the same container image
- D. They’ll share the same IP address

11. What are the two node types in a Kubernetes cluster?

- A. Service nodes

- B. Control plane nodes
- C. Worker nodes
- D. Primary nodes

12. Which of the following Kubernetes components run on control plane nodes? Choose all correct answers.

- A. kubectl
- B. The API server
- C. Kube-proxy
- D. Scheduler

13. Which of the following statements is true?

- A. Only stateful control plane services run on every control plane node
- B. Kubernetes controllers only run on the active control plane node
- C. All control plane services run on all control plane nodes

14. You're planning the deployment of a production Kubernetes cluster. How many control plane nodes should you deploy?

- A. 1
- B. 2
- C. 3
- D. 4

15. What is the name of the Kubernetes command line utility?

- A. docker
- B. cri
- C. kube
- D. kubectl

16. Which control plane service is the only stateful component of the control plane?

- A. The API server
- B. The cluster store
- C. The scheduler
- D. The controller manager

17. What will happen to existing workloads if the control plane goes down?

- A. They'll keep running
- B. They'll crash
- C. They'll drop to 50% performance
- D. They'll perform automatic backups

18. Where does kubectl send commands to?

- A. The scheduler
- B. The API
- C. The cluster store
- D. The API server

19. What is the role of a Kubernetes controller?

- A. To control which pods run on which nodes
- B. To keep observed state in sync with desired state
- C. To keep desired state in sync with observed state
- D. The keep observed state in sync with a GitHub repo

20. Which of the following Kubernetes controllers implements the logic to run stateless applications on Kubernetes?

- A. The Endpoints controller
- B. The StatelessSet controller
- C. The StatefulSet controller

- D. The Deployment controller

21. Which of the following Kubernetes controllers implements the logic to run stateful applications on Kubernetes?

- A. The Endpoints controller
- B. The StatelessSet controller
- C. The StatefulSet controller
- D. The Deployment controller

22. Which of the following Kubernetes controllers will ensure an instance of a particular pod will run on every cluster node?

- A. The DaemonSet controller
- B. The Deployment controller
- C. The StatefulSet controller
- D. The Node controller

23. Which of the following are alternative terms for "observed state"?

- A. Desired state
- B. Acquired state
- C. Current state
- D. Actual state

24. What is the technical term for the process of bringing observed state into sync with desired state?

- A. Retribution
- B. Recalibration
- C. Reconciliation
- D. Reconstitution

25. How do Kubernetes controllers operate?

- A. As a buffer between containers and host kernels
- B. As background reconciliation loops
- C. As stateless key-value stores
- D. As a distributed database

26. What is the role of the kube-proxy on each worker node?

- A. Implement networking rules
- B. A reverse proxy to the API server
- C. Start and stop containers
- D. To run virtual machines inside pods

27. What is the name of the technology that enables Kubernetes to run different container runtimes?

- A. Docker
- B. Runtime agents
- C. Oepncontainers
- D. The CRI

28. What is the most common container runtime on modern Kubernetes clusters?

- A. Containerd
- B. Kata containers
- C. Docker
- D. gVisor

29. Which of the following container runtimes is deprecated or unsupported in recent Kubernetes versions?

- A. Containerd
- B. CRI-O
- C. Docker
- D. gVisor

30. What is the role of the kubeconfig file?

- A. To configure the control plane
- B. To configure workers
- C. To configure kubectl
- D. To configure the API server

31. Which of the following commands will return detailed info about a pod called kcna?

- A. kubectl get pod kcna
- B. kubectl inspect pod kcna
- C. kubectl parse pod kcna
- D. kubectl describe pod kcna

32. You have a Kubernetes cluster running version 1.23. Which of the following versions of kubectl are supported? Choose all correct answers.

- A. 1.21
- B. 1.22
- C. 1.24
- D. 1.25

33. What happens to a newly created pod if the scheduler can't identify a node to run it?

- A. It will remain pending until a suitable node becomes available
- B. It will remain pending for 24 hours
- C. It will remain pending until a cluster administrator releases it
- D. It will remain pending until a redeploy is attempted

34. Which of the following technologies will ensure the Kubernetes scheduler distributes workloads across user-defined fault zones?

- A. The API server
- B. Pod HA policies
- C. Pod topology spread constraints
- D. Network policies

35. Which Kubernetes technology is appropriate for dividing a single cluster so that different teams and departments have their own resource quotas and user accounts?

- A. KubeVirt
- B. The gVisor container runtime
- C. A quota policy
- D. Namespaces

36. Where are Kubernetes objects such as pods, deployments, services, and ingresses defined?

- A. The cluster store
- B. The API
- C. The cluster catalog
- D. The deployment controller

37. How frequently are new versions of Kubernetes released?

- A. Once per year
- B. Quarterly
- C. Every four months
- D. Every month

38. What is the name of the network Kubernetes pods get deployed to?

- A. The pod network
- B. The container network
- C. The overlay network
- D. The default network

39. Which of the following technologies makes the Kubernetes network layer pluggable?

- A. eBPF
- B. OpenShift
- C. OpenNetworking
- D. The Container Network Interface

40. Which popular Kubernetes network plugin is based on eBPF?

- A. Cilium
- B. Calico
- C. Kubenet
- D. Weave

5: Cloud native application delivery

This chapter covers the topics in the *Cloud native application delivery* section of the exam. The topics account for 8% of the overall exam.

The chapter is divided as follows.

- Primer
- CI/CD
- GitOps
- Chapter summary
- Exam essentials
- Recap questions

Primer

Cloud native application delivery is often referred to as *code to cloud* and *code to production,* and we usually automate it with continuous integration (CI) and continuous delivery (CD) pipelines as well as GitOps tools. We'll talk more about all of this later in the chapter, but for now you should know the major automation steps.

1. Build code
2. Test code
3. Release code

Automating these steps enables fast, frequent, and high-quality releases.

Finally, *release* is jargon for rolling out a new version of an application or an environment. For example, updating an application from version 1.1 to version 1.2 is accomplished via a *release*. We often use the terms *rollout, update,* and *release* to mean the same thing.

CI/CD

CI/CD is an acronym for continuous integration, continuous delivery. It’s a set of tools and processes that automates the building, testing, and provisioning of software. It enables frequent reliable releases and is an integral part of the wider DevOps movement.

As the acronym suggests, CI/CD is two separate processes.

- Continuous Integration (CI)
- Continuous Delivery (CD)

CI automates the process of *building* and *testing* software changes. It usually involves developers frequently merging code to a central repository where it’s built and subjected to extensive tests. The building and testing must be fully automated and are highly effective at identifying issues before they’re released into live environments. The whole process is automated and ensures only high-quality code is passed to the CD process.

CD automates the later process of releasing that high-quality code into live environments such as dev, test, and production. It’s often triggered after a build successfully passes CI testing.

The overall CI/CD process is referred to as a *pipeline* as it resembles the operation of a pipe – you put something into one end of the pipe and don’t touch it again until it pops out the other end. The automated nature of CI/CD processes means once a developer uploads a change to a repository, the entire process is automated until the built and tested artefact pops out the other end.

Automation of these steps makes all the following possible.

- Faster software delivery
- More frequent software delivery
- Fewer bugs
- More productive developers

Automation always makes things faster and more repeatable. This is why automated CI/CD pipelines allow developers to test and deploy software faster and more frequently. The fact that testing is automated and mandatory means fewer bugs will make it through to live production environments. Also, the fact that developers aren't involved in the building and testing (they put code into the pipeline and the system does the rest) means they can spend more time being productive building new apps and features.

You might sometimes see CD split into two slightly different things.

- Continuous delivery
- Continuous deployment

Continuous delivery requires human approval before deploying to production, whereas *continuous deployment* automates everything, including deploying to production.

It's normally considered good CI/CD practice to build, test, and release as frequently as possible. Doing this tests the pipeline, keeps developers and management comfortable with the process, keeps changes and updates small and potentially less impactful, and speeds up the delivery of features and fixes. The opposite would be saving up lots of changes and implementing them once or twice per year as big jobs. This is higher risk and delays the release of features and fixes.

Popular CI/CD tools include:

- Jenkins
- CircleCI
- GitLab CI
- Travis CI

In summary, CI/CD is about automating the entire process of application building and testing, all the way through to deployment in production environments. A special set of tools are used to implement CI/CD pipelines and enable frequent and reliable software releases. CI automates building and testing, and uploads the fully tested code to a repository. CD takes the completed code and deploys it to environments such as dev, test, and prod. Many organisations leverage CI/CD tools and processes to make multiple safe software releases per day, often during busy business hours. Automating these tasks also frees up developers to work on new applications and features instead of performing manual builds and tests.

GitOps

The idea of GitOps became popular in 2017 following Alexi Richardson's blog post on the Weaveworks website titled *Operations by Pull Request.*

At a high-level, it takes the battle-tested DevOps practices we've been using for years in software development and brings them to the world of infrastructure and Kubernetes. If you've just read the section on CI/CD, GitOps builds on these by adding version control and pull requests.

Keeping it high-level, you describe the desired state of your infrastructure in configuration files that you store in a Git repo for version control. Every time you make a change to your environment you check the files out of Git, update them with your changes, and submit a Pull Request (PR) to merge the changes into the repo. As soon as the PR is merged, the process of deploying the changes to your live environment kicks in.

Although there are lots of GitOps tools, such as Argo, Flux and Jenkins X, GitOps is actually a collection of tools, workflows, and practices. These include.

- Infrastructure as Code
- Pull requests and merges
- Continuous delivery
- Zero tolerance for manual changes

Let’s take a closer look at each.

Infrastructure as Code (IaC)

IaC is the practice of defining infrastructure in declarative configuration files. This means instead of using GUIs and scripts to deploy things like virtual machines, networks, and policies, you define them in configuration files that you feed to your GitOps tools and let them do the hard work of provisioning. It’s a key component of fast and repeatable infrastructure deployments.

> **Terminology.** *Declarative configuration* is a pattern where you describe the *desired state* of something in a configuration file and let a tool or process implement it. A common real-world example is defining Kubernetes objects in YAML configuration files that you feed to the Kubernetes API server and Kubernetes provisions the objects for you.

Using configuration files to deploy infrastructure works well on cloud platforms where there’s a pool of available infrastructure you can claim and configure. For example, an IaC config file might define cloud instances, network configurations, firewall and ACL rules, routing tables, network policies, and more. You send the config file to your cloud and your cloud provisions it. However, it isn't as powerful in on-premises environments as they don't usually have large pools of configurable resources. Also, IaC cannot perform common on-premises provisioning tasks such as physically racking servers and connecting physical network cables.

One area where IaC is extremely common and powerful is Kubernetes. This is because Kubernetes describes everything in configuration files that are used for provisioning. As an example, Kubernetes pods, services, ingresses, network policies, load-balancers, and more are all declaratively described in YAML configuration files that you post to the API server and Kubernetes provisions them. The following YAML file is an example of IaC – it describes a Kubernetes load-balancer service that will listen on port 9000 and forward the traffic on port 8080 to pods with the `env=prod` label. You can use `kubectl apply` to send the file to a Kubernetes cluster and Kubernetes will talk with your cloud platform and provision the load-balancer.

```
apiVersion: v1
kind: Service
metadata:
  name: test-lb
spec:
  type: LoadBalancer
  ports:
  - port: 9000
    targetPort: 8080
  selector:
    env: prod
```

Pull requests and merges

GitOps combines IaC with well-established Git workflows to create automated continuous delivery pipelines. Over the past few years, it's become a very popular way to manage Kubernetes environments and has inspired a new generation of continuous delivery (CD) tools.

Consider the following example. You build a Kubernetes cluster and install a GitOps tool such as Flux, Argo, or Jenkins X. You configure the GitOps tool to monitor a Git repo that you use to store your Kubernetes configuration. You describe a new application in a Kubernetes YAML file that comprises several Kubernetes objects such as pods, service accounts, secrets, network policies and an ingress. You create a Pull Request (PR) to merge the YAML file into the Git repo. The PR includes a description of the changes and evidence of any testing. Teammates comment on the PR and it gets approved and merged to the repo. Your GitOps tool notices the merge and initiates a continuous deployment process that provisions the app and infrastructure on your Kubernetes cluster.

Sometime later, you need to make a small update to the config. Instead of manually editing the live environment, you create a branch of the Git repo and make the configuration changes there. Once you're happy with them you create a PR to merge them with the master branch of the repo. Teammates once again comment on the PR and eventually approve the merge. Your GitOps tool observes the merge and kicks-off another continuous delivery process to apply the update.

As you can see, *PRs* and *merges* are critical GitOps operations. In fact, there are four important GitOps rules.

1. Store **all** of your app and infrastructure configuration in Git
2. Use PRs and merges for approvals and triggering deployments
3. Automate deployment
4. Stop making manual changes

Let's clear up some of the jargon.

Git is a popular open-source version control system created by Linus Torvalds (the inventor of Linux). Its core functionality is tracking changes and enabling rollbacks, and it's used to manage large and small projects. Huge projects include Linux and Kubernetes, small projects include things such as the lab files for my other books and video training courses.

GitHub is cloud-based code repository that uses Git for version control. It organises files into folders called *repositories* and you normally give each project its own repository. For example, all the files for my Quick Start Kubernetes book are on GitHub in the `nigelpoulton/qsk-book` repository, whereas all the files for The Kubernetes Book are in the `nigelpoulton/TheK8sBook` repository – one project per repository.

Git lets you create *branches* of a repository as areas to work on changes. You work on changes in your own branch and use pull requests to merge your changes into a special branch called *master*. This master branch is the source of truth and all changes and updates from other branches have to be *merged* into master to become part of the project. As mentioned, the process of merging changes to the master branch is initiated via *Pull Requests (PR)* that notify team members the changes are ready to be reviewed and potentially *merged*. PRs provide a comments forum where changes can be discussed and reviewed before being approved and merged. PR comments and approvals are saved and available for future recall, making them ideal for rollbacks and audit requirements.

In the example, a GitOps tool such as Flux or Argo is watching the master branch of the Git repository. Every time it observes a merge, it kicks off a continuous delivery (CD) process to deploy the updated configuration to the live environment.

Continuous delivery

As mentioned earlier in the chapter, CI/CD is two distinct processes.

- **CI** builds and tests software

- **CD** deploys to live environments

GitOps is primarily focussed on the CD aspects and doesn't handle the earlier CI steps. This means you'll still need a CI tool to handle application building, testing, and sharing. GitOps kicks in after those steps, adding version control, change management, and CD.

Zero tolerance for manual changes

Once you adopt GitOps, you can no-longer make manual changes to live environments as they'll be lost with every PR and successful merge.

Consider the following example. You're using GitOps to manage a live Kubernetes environment. However, a team member who isn't fully onboard with the GitOps process makes a manual change to the live environment. This is an *anti-pattern* and causes the state of the live environment to drift from what is defined in the Git repo. Later the same day, you follow the GitOps process of testing a change in a feature branch and creating a PR to merge it. When the PR is approved, your changes are merged and the CD pipeline automates deployment to the live environment. The process will apply the entire configuration to the live environment and overwrite the manual change made by your teammate.

Let's look a bit closer.

Assume the config files in your GitHub repo specify Kubernetes objects such as pods, services, and network policies. You've been following your GitOps process and the live environment exactly matches the configuration stored in the GitHub repo. However, an application starts experiencing an issue, so a teammate logs on to a live system and manually updates a live network policy that resolves the issue. While this might seem like a good idea at the time, the live environment no longer matches the configuration in the GitHub repo and can become a problem later.

Later the same day, you follow the GitOps process to deploy a new set of application pods and you create a PR to merge the change. The PR is approved and merged to the master branch, your GitOps tool observes the change and triggers the CD workflow. Even though you only added new application pods, the entire configuration from the repo will be re-applied to the environment. This means network policies will also be re-applied and will overwrite your teammate's manual fix. This is why it's extremely important everybody buys into the GitOps process and you stop allowing manual changes to live environments.

> **Jargon busting.** An anti-pattern is something that seems like a good idea at the time but is actually a bad idea.

Important GitOps tools and projects

As previously mentioned, GitOps inspired a new generation of continuous delivery tools that integrate with Git workflows. We'll briefly describe the following and you may be tested on them in the exam.

- Flux
- Argo
- Jenkins X

Flux (sometimes called FluxCD or Flux CD) is an easy-to-use Kubernetes-native application that synchronises Kubernetes state with config files in a Git repo. It started as a Weaveworks project and is now an incubating CNCF project. You install it on a Kubernetes cluster and configure it to watch a Git repo for changes. "Watching" a repo is a pull-based model where Flux periodically checks the repo and synchronises any changes with the live Kubernetes environment. Flux is known for being extremely simple to use and is entirely focussed on CD to the Kubernetes cluster (it doesn't do CI). It can only watch one Git repo and can only deploy to one Kubernetes cluster and namespace. This makes it easy to install and configure but means it’s less configurable than some other tools.

Argo (sometimes called ArgoCD or Argo CD) is a lot like Flux. It’s a declarative GitOps tool designed for Kubernetes that was originally developed at Intuit and is currently an incubating CNCF project. It’s also installed as a Kubernetes-native app, is pull-based, and focusses on CD (doesn’t do CI). An advantage it has over Flux is that a single Argo installation can monitor multiple Git repos and deploy to multiple Kubernetes namespaces. This can make it feel like the “Pro” version of Flux.

Jenkins X is a Kubernetes focussed GitOps tool that covers the entire CI/CD process. Under-the-hood it’s a collection of other open-source tools that are nicely packaged and relatively simple to deploy. However, it’s a lot more complex than Argo and Flux.

In summary, GitOps combines infrastructure as code, Git workflows, and CD pipelines to automate infrastructure provisioning. In fact, GitOps is often considered “infrastructure as code done properly”. You define the desired state of your infrastructure in declarative configuration files that you store in a Git repo. The repo becomes the source of truth for your environment, meaning any time a PR is merged to the master branch of the repo the CD pipeline deploys it. It enables frequent and reliable deployments as well as ensuring manual changes are overwritten with each merge. The native version control keeps a history of every change that can be used for fast and reliable rollbacks as well as auditing. This audit history includes all pull requests, comments, and approvals. Finally, large numbers of people can work on changes in their own feature branches, but you can limit the number of people who can approve merges. It’s also vital that everyone adopts the GitOps process and stops making manual changes to live environments.

Chapter summary

In this chapter, you learned that CI/CD automates the building, testing, and deployment of software and infrastructure. CI deals with automating the early steps of building and testing code, and then publishing it to repositories. CD takes the built and tested artefact and automates the process of deploying it to live environments. GitOps builds on this by adding version control, pull requests, and approvals that trigger the CD process. GitOps is particularly powerful in Kubernetes environments where everything is declaratively described in YAML configuration files that are used to configure the environment. Popular GitOps tools include Argo, Flux, and Jenkins X.

Exam essentials

Continuous integration, continuous delivery (CI/CD)

CI/CD is a set of tools and practices focussed on automating the steps taking software from code to production. It's a key part of DevOps practices and enables fast, repeatable, and reliable testing and deployment. It also gives developers more time to work on features and fixes. The automated steps are often referred to as a *pipeline.*

Continuous integration (CI)

CI is focussed on automating software builds and tests. A developer will typically merge code to a repo with a particular tag that kicks-off an automated pipeline that builds the software and runs tests against it. It's common to spin-up the build in a container and perform the tests. The code that passes the tests gets passed on to later CD stages.

Continuous delivery (CD)

CD is focussed on automating the delivery of software and infrastructure to live environments. It typically runs after CI processes and only deploys "good code". Good code is code that's passed CI tests.

Infrastructure as Code (IaC)

Infrastructure as Code is the practice of defining infrastructure in configuration files and using a platform to provision it. It's extremely popular in Kubernetes environments where all Kubernetes objects can be defined in YAML manifest files that are used to provision Kubernetes apps and infrastructure. IaC is particularly powerful in Kubernetes environments running on public clouds as the entire infrastructure stack can be described and deployed from config files.

GitOps

GitOps takes infrastructure as code and adds version control, approvals, and CD pipelines. A typical GitOps workflow will have a GitOps tool monitoring a Git repo for merges. Any time software or infrastructure configs are merged, the GitOps tool kicks-off a CD pipeline that deploys the update to the live environment. A new breed of CD tools, including Argo, Flux, and Jenkins X are popular GitOps tools. Many people consider GitOps to be IaC done properly.

Recap questions

See Appendix A for answers.

1. Which of the following terms refers to the process of updating an application from version 1.1 to 1.2? Choose all correct answers.

- A. Release
- B. Rollup
- C. Rollout
- D. Rollback

2. Which of the following is a description of continuous integration (CI)?

- A. Automating the deployment of updates to dev and test environments
- B. Automating the deployment of updates to production environments
- C. Automating building and testing of application code
- D. Automating the deployment of infrastructure

3. What is a goal of continuous integration (CI)?

- A. Identify issues before they reach the live environment
- B. Integrate software deployment with well-known Git workflows
- C. Integrate on-premises and public cloud networks
- D. Integrate secrets management into workflows

4. Which of the following is a description of continuous deployment?

- A. Automating the deployment of updates to live environments
- B. Automating building and testing of application code
- C. Automating the testing of infrastructure

5. Which of the following is usually considered a good CI/CD practice?

- A. Run more tests than releases
- B. Run more releases than tests
- C. Frequent releases
- D. Infrequent releases

6. Which of the following is a popular CI/CD tool?

- A. Kubernetes
- B. Docker
- C. Swarm
- D. Jenkins

7. Which of the following is a positive side-effect of an automated CI/CD pipeline?

- A. Smaller container images
- B. Developers with more time to develop apps and features
- C. Binary-level vulnerability scanning
- D. Developers become skilled at running tests

8. Which of the following are part of a typical GitOps framework? Choose all that apply.

- A. Infrastructure as Code
- B. Containers as a service
- C. Continuous Delivery (CD)
- D. Pull requests (PR)

9. Which of the following can infrastructure as code define? Choose all correct answers.

- A. Policies
- B. Kubernetes objects
- C. Server cabinets
- D. Virtual networks

10. Which of the following environments does Infrastructure as Code work best with?

- A. On-premises datacenters
- B. Public clouds

11. What language are Kubernetes objects normally described in?

- A. Rust
- B. YAML
- C. SOAP
- D. Python

12. What is the typical continuous delivery trigger in a GitOps workflow?

- A. Pull request and merge
- B. Git rebase
- C. Docker pull
- D. Committing a file with the deploy tag

13. Which of the following are GitOps tools? Choose all correct answers.

- A. Argo CD
- B. Flux CD
- C. Docker CD
- D. AWS

14. Which of the following is an anti-pattern in a GitOps environment?

- A. Commenting on pull requests
- B. CD pipelines
- C. Using GitHub repositories
- D. Making manual changes to live environments

15. Which of the following is an open-source version control system created by Linux creator Linus Torvalds?

- A. GitHub
- B. Kubernetes
- C. BitBucket
- D. Git

16. Which elements of CI/CD do GitOps tools and processes usually implement?

- A. CI

- B. CD

17. Which of the following is a feature of GitOps?

- A. Manual changes are overwritten with every merge
- B. Environments experience short downtime with every merge
- C. Every merge spins up a brand-new additional environment
- D. Frequent tests

18. Which of the following best describes an anti-pattern?

- A. Declaratively defining infrastructure in config files
- B. Scaling applications based solely on CPU and memory usage
- C. Something that seems like a good idea but isn’t
- D. Frequent deployments

19. Which of the following GitOps tools is well-known for being simple to configure and use?

- A. Flux CD
- B. Jenkins
- C. Jenkins X
- D. Travis CI

6: Cloud native observability

This chapter covers the topics required to pass the *cloud native observability* section of the exam and accounts for 8% of your exam grade.

The chapter is divided as follows.

- Primer
- Telemetry and observability
- Prometheus
- Cost management
- Chapter summary
- Exam essentials
- Recap questions

Primer

Cloud native applications are deployed as lots of small moving parts that connect over the network and are frequently updated and replaced. This makes it more important than ever to be able to see how applications are connected and have the ability to track requests as they traverse the various distributed microservices of an application.

To assist with this, every microservice needs to output high-quality telemetry in the form of metrics and log data. You then need the right tools to collect the telemetry, aggregate it, and analyse it for troubleshooting and making adjustments. The process is a feedback loop that looks like Figure 6.1 – observe > analyse > adjust.

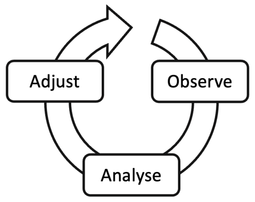

Figure 6.1

Even though we've been doing things like this for years with traditional monolithic apps, all of the following are either new or far more important with cloud native apps.

- Tracing requests
- Application-specific metrics
- Network latency
- Scaling decisions
- The volume of logs and metrics
- Collating and analysing telemetry

Telemetry and observability

Telemetry is jargon for logs, metrics, and traces. This is *observed* by remote monitoring platforms that use it to learn how systems work, diagnose problems, and measure and optimise performance. A highly observable system is one that generates high-quality telemetry.

Consider the following quick example. You have an in-house cloud native application comprising lots of small microservices. You also have a centralised monitoring platform. Each microservice outputs telemetry that is collected, stored, and analysed by the monitoring platform.

As mentioned already, cloud native telemetry comes in three major flavours that we sometimes call *signal types, verticals,* or *classes*.

- Logs
- Metrics
- Traces

Each serves a different purpose, but collectively they provide detailed insights that can be vital in understanding, troubleshooting, and optimising microservices applications.

The OpenTelemetry project is an incubating CNCF project that aims to standardise the way we collect telemetry and export it to centralised systems for analysis. The project consists of specifications and instrumentation. *Instrumentation* is jargon for coding telemetry into applications.

Finally, the sheer volume of telemetry emitted by cloud native apps can resemble a firehose and be overwhelming. This is because every microservice generates its own telemetry, resulting in cloud native apps often generating 10x or even 100x the amount of telemetry as traditional apps. This has driven the development of a new generation of cloud native monitoring and monitoring-related tools. These include Prometheus, Fluentd, Grafana, Jaeger, and more.

Log data

Log data is usually application-related *events* that are great for debugging problems and one of the first places you should check when something goes wrong.

Each event, such as a dropped connection or a service restart, generates a log entry that is timestamped and often formatted as JSON for better organising and querying. They're also categorised according to severity levels such as *info, warning, error,* and *critical. Debug* is another level that is usually the most verbose, often including all the data from all other levels.

Many applications write logs to the standard out (stdout) and standard error (stderr) streams. However, some applications write logs to arbitrary locations.

> **Jargon.** Very old computer systems had three standard *streams* relating to input and output channels. The *standard in (stdin)* stream handled input coming from a keyboard, *standard out (stdout)* handled output going to the screen, and *standard error (stderr)* allowed error information to be streamed independent of output to the screen. Linux continues to implement these streams, and it's common practice for log data to be sent to stdout and stderr. This architecture still exists in containers running on Kubernetes.

In Kubernetes environments the kubelet on each node watches the stdout and stderr streams of every container and makes the log messages available to the `kubectl logs` command. It also keeps the logs from the most recently terminated instance of every container in case you need to go back to them. Logs that are not written to a container's stdout and stderr cannot be viewed using `kubectl logs`.

A drawback to the kubelet on each node keeping logs is that if the node fails all log messages are lost. This is why it's considered a better practice to store logs in a centralised location where the lifecycle of the log data is separate to the lifecycle of the container and the node it's running on. Kubernetes calls this architecture *cluster-level logging* and offers the following options.

- Node-level logging agent
- Sidecar with logging agent

The *node-level logging agent* installs an agent on each node that gathers all container logs from the kubelet and ships them to a central location. It's common practice to use a Kubernetes DaemonSet to deploy and manage the agent on each node.

A logging *sidecar* is a support container that runs in the same pod as an application container. It captures the logs from the application container and either streams them to its own stdout and stderr, or forwards them directly to a central location.

Consider the following example. You have an application that writes log data to a bespoke location in the container's filesystem. This means the local kubelet can't access them. However, the application provides a sidecar container that knows where the logs are written to and forwards them to its own stdout and stderr streams. Doing this allows the kubelet on the node to capture them from the sidecar and make them available to a node-level agent that can forward them on for aggregation. It also makes them available to the `kubectl logs` command. It's a simple solution that means applications don't need to change where they write logs to – the sidecar knows the location and puts them in the place Kubernetes expects them.

Logging sidecars can also forward logs directly to a central location. Doing this bypasses the kubelet and means you won't be able to view them with `kubectl logs`. This is because `kubectl logs` works with the local kubelet and shipping them straight to a central location bypasses the kubelet.

Application-level logging is another pattern that pushes logs straight to the central store in the required format. This also skips the kubelet and requires applications to be configured with the location of the central store. This is an anti-pattern in cloud native apps as it's not usually good practice to code environment-specific configuration into applications.

As already mentioned, the `kubectl logs` command returns the logs from a running container as long as they're sent to stdout or stderr. It can also return the logs from the most recently terminated instance of a container. If a pod has more than one container, you have to specify the container name with the `-c` flag. If the pod has only one container you can omit the `-c` flag.

The following command retrieves logs from the only container in a pod called “kcna”.

```
kubectl logs kcna
```

If the pod is failed or terminated, you can use the `-p` or `--previous` flag. The following example retrieves the logs from the ctr1 container that ran in the terminated kcna pod.

```
kubectl logs -p -c ctr1 kcna
```

Metrics

Metrics are performance-related values gathered and measured over a period time. Each metric is time-stamped, has a name, and can have labels to help with grouping and querying. They’re a great way to see historical trends and predict future changes.

Examples of metrics include CPU utilisation, queue length, number of requests, error rate, and more. For example, if the number of requests has been averaging 150 per minute for the last 6 months and suddenly spikes to 10,000, you know something has changed and the system may need to react. Likewise, if dropped connections has been averaging less than 2 per minute since the system was deployed, but suddenly jumps to 5,000, you know something’s wrong.

Generic system metrics such as CPU utilisation and memory usage are considered basic and lack any application intelligence. Custom application metrics are usually more useful. Examples include, queue length, response time, requests per second, and a lot more. The point is, they’re specific to an application and therefore a lot better at determining application performance and whether or not scaling operations need to be initiated.

Speaking of scaling. Cloud native microservices applications will normally autoscale to meet demand. However, autoscaling decisions are only as good as the metrics being observed. So, it’s not only vital you monitor the right metrics, you also need to be able to quickly analyse them for scaling decisions. Consider the following simple example.

You have a cloud native app with 3 microservices and autoscaling configured to watch memory usage. Requests traverse all three microservices and you’re suddenly experiencing slow response times. You look at your monitoring dashboard and see memory usage is within acceptable limits. However, one of the microservices has an abnormally large queue size. When discussing with the application developers you realise that memory usage isn’t a good indicator of application performance and therefore not a good metric to scale on. You rectify the immediate problem by adding an extra instance, but you decide to implement custom metrics that monitor queue performance so that when this happens again the system has the right metrics to make automatic scaling decisions.

Network-related metrics, such as latency, are also more important than ever as more and more requests are reliant on the network.

Tracing

Tracing is a technical term for tracking the end-to-end journey of requests as they pass through the different components of a distributed system.

Individual requests can be traced, and each trace reports things such as which system components processed the request, any events, and how long each event took to process. They’re more important than ever in understanding traffic flow and potential bottlenecks in distributed microservices environments. For example, a request that might previously have hit a single server and been handled locally might now hit several servers and traverse the network several times as it's passed between microservices. With this in mind, it's vital you can observe the end-to-end journey of every request so you have the full picture.

Jaeger is a popular tool for storing and analysing tracing data.

In summary, effective observability requires high-quality telemetry. Log data is created by applications and is a good place to look when something goes wrong. Metrics are a great way of seeing historical trends and spotting when trends change. Traces are vital in understanding request flow in distributed microservices applications. The OpenTelemetry project is trying to make it as simple as possible to instrument applications with high-quality telemetry.

Prometheus

Prometheus is considered the *industry standard* monitoring tool for Kubernetes and cloud native environments. It was created by SoundCloud in 2012 when they realised existing monitoring tools weren't good enough for what they needed. It was donated to the CNCF as an open-source project in 2016 and became the second project to graduate the CNCF in late 2018.

It's a common pattern to run cloud native applications on Kubernetes, use Prometheus for monitoring and alerting, and use Grafana for dashboards and visualisation.

How Prometheus works

At a high level, every microservice should generate high-quality telemetry in the form of logs and metrics. Prometheus collects these and stores them where they can be queried and analysed to help with troubleshooting and optimisation. It can also feed into Grafana for high-quality dashboards and visualisations.

Digging a little deeper…

Prometheus expects applications to expose metrics via the `/metrics` HTTP endpoint. That's jargon for a web address that ends in `/metrics`. For example, if a microservice is reachable at `store.myapp.k8s.local` it should expose Prometheus metrics on `store.myapp.k8s.local/metrics`. Prometheus periodically connects to this endpoint to *scrape* the metrics and store them. Scraping is jargon for "collecting" the metrics. This is a *pull* model where the application is only responsible for making the metrics available and it's the responsibility of Prometheus to do the hard work of scraping them. The frequency at which Prometheus scrapes metrics is called the *scrape interval*.

Once Prometheus has the metrics, it stores them in a timeseries database. Events are stored in plain text and each event is stored on its own line in the database. *Timeseries* means each event has a timestamp. In fact, each Prometheus metric has a name, a timestamp, and various labels. This makes them very easy to group, query, and analyse. On the topic of queries, Prometheus has its own powerful query language called PromQL. You may also hear the Prometheus data model referred to as *multi-dimensional*. This means metrics can be grouped, queried, and organised on name, timestamp, and labels.

Prometheus provides basic dashboards but is usually coupled with Grafana for more powerful visualisations.

Some applications, such as short-lived batch jobs, don't work well with pull models where scraping is done at regular intervals. A simple example is a short-lived batch job that might exit before Prometheus scrapes its metrics. In these scenarios, you can configure the application to *push* metrics to the Prometheus *pushgateway* where they’re stored until Prometheus comes along and scrapes them.

Applications that don’t natively support Prometheus metrics

You have two main options when an application doesn’t natively expose Prometheus timeseries metrics on the `/metrics` endpoint.

- Client libraries
- Exporters

If you own the application, you can use a *client library* to add Prometheus support. Client libraries implement the required timeseries metrics and usually expose them via the expected `/metrics` endpoint. They exist for applications written in most languages, including Python, Java, Go and a lot more. Doing this is called *direct instrumentation*.

If you’re using 3rd-party application, your developers won’t be able to directly instrument them with client libraries. In this scenario, the best option is usually an *exporter.* The exporter runs as a sidecar in the application pod, captures that application’s native metrics and logs, converts them to Prometheus timeseries format, and exposes them on a `/metrics` endpoint. A common example is the NGINX web server. NGINX is an extremely popular application but doesn’t support Prometheus metrics and doesn’t expose a `/metrics` endpoint. However, a Prometheus exporter for NGINX exists that sits alongside NGINX, captures its metrics, converts them into timeseries format, and exposes them on a `/metrics` endpoint.

Prometheus and alerting

Prometheus can also do alerting. This is where it pings your phone or sends a message about a condition.

For this to work, a condition needs to persist long enough for an alert to be triggered. The alert is then sent to the Prometheus alert manager which deals with how and where to send it. Possible actions include alerting a pager, sending an email, sending SMS or Slack messages, and more.

In summary, Prometheus is the most popular cloud native monitoring tool and is frequently deployed in Kubernetes environments. It has built-in support for Kubernetes, including the ability to automatically discover services. It expects metrics and log data to be exposed in timeseries format on the well-known `/metrics` endpoint that it scrapes at regular intervals. Client libraries are available in most programming languages to add Prometheus support to your in-house applications. This is called *direct instrumentation.* You can use exporters if you're using 3rd-party applications and can't directly instrument them with client libraries. Finally, Prometheus is often paired with Grafana for best-of-breed dashboards and visualisations.

Cost management

Effective cost management is a key part of managing cloud native apps and infrastructure, and you should always consider the following three things.

1. Choosing the right infrastructure
2. Rightsizing
3. Turning unused infrastructure off

Choosing the right infrastructure

Choosing the right cloud platform is a vital decision. However, even after you've chosen your cloud, there are still a lot of decisions that have a high impact on costs and cost management.

For example, most clouds offer the following instance categories – *instance* is the technical term for a virtual server on a cloud.

- On-demand
- Reserved
- Spot

On-demand instances are the most common and most flexible, however, they're the most expensive. You can create and delete them "on demand" and they're yours for as long as you need them.

Reserved instances come at a discounted price in exchange for a long-term commitment. For example, if you commit to a use an AWS EC2 instance for 3 years, and pay part of the cost upfront, you may get it up to 50-70% cheaper than if you consume it as an on-demand instance. However, you're tied into paying for the duration of the agreement, even if you stop needing it part way through.

Spot instances are usually the cheapest but can be deleted at any point. All of the big clouds carry spare capacity to cope with surges in customer demand. To offset the cost of this spare capacity sitting idle, most clouds offer it as *spot instances* at significant discount (sometimes up to 90% discount). However, if the cloud needs that capacity, it can simply delete your spot instances. And while some clouds send capacity rebalancing signals to alert you when spot instances are being removed, there's no guarantee they'll be sent or received in time for you to react.

Rightsizing for cost

Rightsizing is making sure you're running the right amount of infrastructure to meet requirements and is a delicate balance between performance and cost. If you provision too much infrastructure, performance will be great but you'll be paying too much. If you provision too little, you'll save money but performance will be poor. Fortunately, autoscaling can help.

Systems like Kubernetes can automatically scale applications and infrastructure up and down to meet demand. This means you should always be running the right amount of infrastructure and not paying for infrastructure you don’t need.

For example. Most cloud-based Kubernetes clusters allow you to define *node pools* with upper and lower limits. An example might be a node pool with a low limit of 2 nodes and high limit of 10 nodes. This allows Kubernetes to scale the number of pool nodes anywhere between 2 and 10 based on demand. Kubernetes can also dynamically provision volumes on external storage systems and then release them when they’re no longer needed. All of this helps ensure you have the right balance of cost and performance – running and paying for the right amount of infrastructure to provide a good user experience.

Two important autoscaling configuration options include.

- Configuring the right upper and lower limits
- Watching the right metrics

We’ve already explained upper and lower limits for node pools. They provide protection against compromised or runaway services. For example, a node pool configured with an upper limit of 10 will never provision more than 10 nodes. This helps cap potential costs.

Watching the right metrics is also important. For example, queue size and request response time might be better metrics than CPU utilisation for a message queue system.

Also related to rightsizing and cost management is identifying unused resources and turning them off or deleting them. Autoscaling can help here. However, if you don’t have autoscaling configured for all resources you should regularly check that you’re not paying for infrastructure and services you’re no longer using.

In summary, autoscaling is a major tool in cloud native cost management. If configured correctly, it can ensure you’re only using and paying for the right amount of infrastructure to meet requirements. Choosing the right instance types is also a huge part of getting cloud costs right. Spot instances offer the biggest discounts but can be terminated without warning. Reserved instances offer discount in exchange for a long-term commitment to the resource, usually 1-3 years. On demand instances are more expensive than spot instances and reserved instances, however, they won’t be deleted by your cloud platform and you’re not bound to a long-term usage contract.

Chapter summary

In this chapter, you learned the three main types of telemetry are logs, metrics, and traces. Logs are application events and are a great tool for determining why an application or microservice isn't working. Metrics provide historical trends and help identify changes over time. Traces let you track requests as they traverse today's complex microservices apps. You also learned that the OpenTelemetry project offers specifications and instrumentation to standardise and simplify the way we generate and store telemetry.

You learned that Prometheus is the most popular monitoring platform used with Kubernetes and is often coupled with Grafana for dashboards and visualisations. It expects services to expose telemetry on the `/metrics` endpoint and operates a pull model where it "scrapes" the data at periodic intervals. It has a push gateway for getting telemetry from short-lived processes, and an alert manager for sending out alerts.

You finished the chapter learning how things like reserved instances, spot instances, and autoscaling have an impact on costs and cost management.

Exam essentials

Telemetry

Telemetry is jargon for the monitoring-related outputs generated by a system. They include logs, metrics, and traces. High-quality telemetry is vital for troubleshooting, tweaking, and understanding the behaviour of cloud native apps.

OpenTelemetry

The OpenTelemetry project is an incubating CNCF project that defines specifications and instrumentation aimed at making high-quality telemetry part of all cloud native apps.

Logs

Logs are a form of telemetry relating to application events and are one of the best places to look when things break. For example, application restarts, failed logins, dropped connections, and more will all be logged as events. Log data is often structured and formatted as JSON to make it easy to read and search. It's also commonly categorised according to severity, with common severity levels including *info, warning, error,* and *critical.*

Metrics

Metrics are usually performance related data captured over long periods of time and used to identify performance trends. Autoscalers can use metrics to forecast potential issues and perform proactive scaling operations.

Traces

Tracing is the art of tracking the end-to-end journey of requests as they traverse complex distributed systems such as microservices applications. They're an extremely important element of optimising and troubleshooting distributed applications. Jaeger is a popular tool for analysing traces.

Observability

A highly observable application is one that outputs high-quality telemetry that is stored in a central location for analysis. Observing telemetry is often the best way of figuring out how distributed applications work and troubleshooting them.

Prometheus

Prometheus is a cloud native monitoring platform. It's a graduated CNCF project and the most popular monitoring platform for Kubernetes environments. It operates a pull-based model where it scrapes metrics and log data from remote systems and stores them in a timeseries database. It also provides basic dashboards but is commonly paired with a more powerful visualisation platform such as Grafana.

On-demand instances

On-demand instances are the most common type of cloud instance. They're the most flexible and come at the highest cost. You can create and destroy them any time you want, and the cloud platform guarantees their availability.

Reserved instances

Many cloud platforms offer reserved instances at discounted pricing in exchange for a long-term commitment.

Spot instances

Many cloud platforms offer spot instances from the spare capacity they carry. They're the cheapest type but have no availability guarantees. For example, the cloud will take spot instances away from you if the capacity is needed elsewhere.

Recap questions

See Appendix A for answers.

1. Which of the following best describes telemetry?

- A. Remote shell access to a system
- B. A map diagram of a microservices app
- C. Pay-as-you-use cloud infrastructure
- D. Health and performance related system outputs

2. Which of the following are the three main types of telemetry data? Choose three.

- A. Logs
- B. Traces
- C. Metrics
- D. Audits

3. What format does Prometheus require metrics in?

- A. Hexadecimal

- B. YAML
- C. SOAP
- D. Timeseries

4. Which of the following describes the main aim of telemetry and observability?

- A. Troubleshooting and auditing systems
- B. Troubleshooting and improving systems
- C. Auditing and improving systems
- D. Troubleshooting and proving system snapshots

5. Which of the following is a CNCF project defining specifications and instrumentation for cloud native observability?

- A. CloudWatch
- B. The OpenTelemetry project
- C. Kubernetes
- D. The OpenMonitoring project

6. Which of the following telemetry classes outputs application events in text or JSON format?

- A. Metrics
- B. Logs
- C. Traces
- D. Audits

7. Which of the following shows log data ranked from lowest severity to highest?

- A. Info, warning, critical, error
- B. Warning, critical, error, info
- C. Info, warning, error, critical
- D. Info, critical, warning, error

8. Which of the following generates more output?

- A. High verbosity
- B. Low verbosity

9. You're troubleshooting an application and need as much info as possible from logs. Which logging level should you choose?

- A. Critical
- B. Info
- C. Debug
- D. Warning

10. Which two types of cluster-level logging does Kubernetes support?

- A. Node-level agent
- B. Sidecar
- C. Aggregated
- D. Persistent logging

11. Which of the following logs can be read with the `kubectl logs` command? Choose two.

- A. Logs written to a container's stdin stream
- B. Logs written to a container's stderr stream
- C. Logs written to a container's stdout stream
- D. Logs written to a bespoke filesystem location

12. You have an application that doesn't generate logs in the format required by your central logging system. Which of the following might help? choose all correct answers.

- A. A node-level logging agent
- B. A sidecar container with logging agent
- C. A Kubernetes DaemonSet

13. Why is application-level logging sometimes considered an anti-pattern in cloud native apps?

- A. Applications shouldn't contain environment-specific data
- B. Applications shouldn't generate log data
- C. Application logging is too verbose
- D. Application logging generates too much east-west network traffic

14. Which of the following commands will show the logs of a terminated Kubernetes pod called "kcna-pod"?

- A. kubectl logs kcna-pod -p
- B. kubectl logs kcna-pod -c
- D. kubectl get logs kcna-pod

15. Which of the following telemetry classes is the best for seeing historical trends?

- A. Logs
- B. Timeseries
- C. Traces
- D. Metrics

16. Which of the following metrics are usually the best at determining application health and performance?

- A. CPU utilisation
- B. Memory utilisation
- C. Custom metrics
- D. Network latency

17. Which of the following telemetry classes is the best for tracking requests as they traverse the components of a distributed microservices app?

- A. Tracing
- B. Logging
- C. Metrics

18. Which of the following is considered by many as the industry standard monitoring tool for Kubernetes?

- A. Jaeger
- B. Grafana
- C. Prometheus
- D. GraphQL

19. Which of the following tools is commonly paired with Prometheus to provide visual dashboards?

- A. Kubernetes
- B. Jenkins
- C. GraphQL
- D. Grafana

20. Which of the following endpoints does Prometheus expect metrics to be exposed on?

- A. /healthz
- B. /metrics
- C. /metricz
- D. /health

21. What is happening when Prometheus is “scraping”?

- A. It’s discarding metrics older than a certain date
- B. It’s pushing an alert to a remote device
- C. It’s gathering metrics data from remote systems
- D. It’s updating visual dashboards

22. Does Prometheus use a push or pull model for gathering remote metrics?

- A. Push
- B. Pull

23. What format does Prometheus store metrics data in?

- A. Timeseries
- B. Timeslice
- C. XML
- D. SOAP

24. You run lots of short-lived applications and need a way to gather their telemetry into Prometheus. What should you do?

- A. Keep your apps alive until Prometheus scrapes their telemetry
- B. Nothing, Prometheus handles this automatically
- C. Have your apps push metrics to a Prometheus pushgateway
- D. Prometheus doesn’t have a solution for this

25. You have several in-house apps that you need to integrate with your Prometheus monitoring environment. Which of the following is a good solution?

- A. Prometheus pushgaetway
- B. Prometheus client libraries
- C. A Prometheus exporter
- D. Prometheus alert manager

26. You have several 3rd-party applications that you need to integrate with your Prometheus monitoring environment. Which of the following solutions will help?

- A. Prometheus pushgaetway
- B. Prometheus client libraries
- C. A Prometheus exporter
- D. Prometheus alert manager

27. Which of the following are three pillars of cloud native cost management? Choose three.

- A. Choosing the right infrastructure
- B. Rightsizing
- C. Annual billing
- D. Turning unused things off

28. You’re deploying a new CI/CD pipeline that’s **not** business critical. Part of it is build servers and you need to keep costs as low as possible. Which of the following instance types is most cost effective?

- A. On-demand instances
- B. Spot instances
- C. Reserved instances
- D. Serverless instances

29. Which of the following generic cloud instance types gets you discount in exchange for a long-term commitment?

- A. On-demand instances
- B. Spot instances
- C. Reserved instances
- D. Serverless instances

30. Which of the following technologies can help with application and infrastructure rightsizing?

- A. CI/CD
- B. Autoscaling

- C. Self-healing
- D. Reserved instances

31. Which of the following are important when configuring autoscaling for infrastructure? Choose two.

- A. Request response times
- B. Setting upper and lower limits
- C. Watching the right metrics
- D. Using reserved cloud instances

The exam

This chapter prepares you to take and pass the exam. It’s divided as follows:

- Exam domains and competencies
- About the exam
- Booking the exam
- Taking the exam
- Getting your result
- Staying connected

Exam domains and competencies

The overall aim of the KCNA is to test and prove your understanding of the following broad topics:

- Cloud native and microservices architectures
- Kubernetes architecture
- The wider Kubernetes and cloud native landscape (CI/CD, GitOps, observability, security etc.)
- Basics of using kubectl

The exam tests the following topics that it calls *domains and competencies,* and you’ll notice they map directly to the chapters of the book. However, the book organises them so each chapter builds on the previous one and prepares you for the next one. As the name of the certification suggests, Kubernetes accounts for nearly half of the exam and the Kubernetes questions feel slightly more in-depth than other topics.

- Kubernetes fundamentals: 46%
  - Kubernetes resources
  - Kubernetes architecture
  - Kubernetes API
  - Containers
  - Scheduling
- Container orchestration: 22%
  - Container orchestration fundamentals
  - Runtime
  - Security
  - Networking
  - Service mesh
  - Storage
- Cloud native architecture: 16%
  - Autoscaling
  - Serverless
  - Community & governance
  - Roles and personas
  - Open standards
- Cloud native observability: 8%
  - Telemetry and observability
  - Prometheus
  - Cost management
- Cloud native application delivery: 8%
  - Application delivery fundamentals
  - GitOps
  - CI/CD

About the exam

Here are the important details about the exam:

- **Delivery method.** Remote via secured browser with remote proctor
- **Style.** Multiple-choice
- **Duration.** 90 minutes
- **Passing score.** 75%
- **Valid for.** 3 years
- **Cost.** $250 USD with one free resit

The exam is an online, proctored, multiple-choice exam. This means you take it remotely over the internet and have a person remotely monitoring your desktop, video, and audio so you don't cheat. You'll be required to download a secure browser and there's no option to take the exam at authorised testing centers.

All questions are multiple choice and almost all of them have 4 possible answers. There are no questions where you have to type your answers and there are no hands-on/simulator style questions.

Once you start your exam you have 90 minutes to complete all 60 questions. The platform allows you to mark questions for review that you can revisit later. You have to review and submit your answers to **all questions** before the 90 minutes expires.

You need to score a mark of 75% or more to pass the exam and you'll receive your score within 24 hours of completing the exam. This is frustrating for an online exam where it would be simple to give you an instant result. It usually takes nearly 24 hours to receive your result, so don't sit there all day clicking refresh in your inbox or in the exam portal.

Booking the exam

You can book your exam at the following link and the process is as follows:

1. Purchase the exam
2. Book your exam date
3. Prepare and complete the pre-exam checklist
4. Take your exam

https://training.linuxfoundation.org/certification/kubernetes-cloud-native-associate/

The exam costs $250 and I recommend you book it ASAP and use the date as motivation and a target. However, if it's getting close and you don't think you're ready, you can re-schedule as close as 24 hours in advance for no extra charge. You also get one free resit if you don't pass first time.

When you purchase the exam, you'll receive a confirmation email from *The Linux Foundation Training* at `<noreply@mail.linuxfoundation.org>`. The email is titled "Order Confirmation" and it has a button that takes you to your Linux Foundation portal where can complete all of the following.

- Agree to the Global Candidate Agreement
- Verify your name
- Schedule your exam date
- Check your system requirements
- Read the Candidate Handbook and other important instructions

I highly recommend you complete all pre-exam requirements as soon as possible so you're prepared for the exam and don't have lots of admin to complete on the day of your exam.

When scheduling the exam, you'll be shown a calendar view where you choose an available date and time as shown in Figure 7.1.

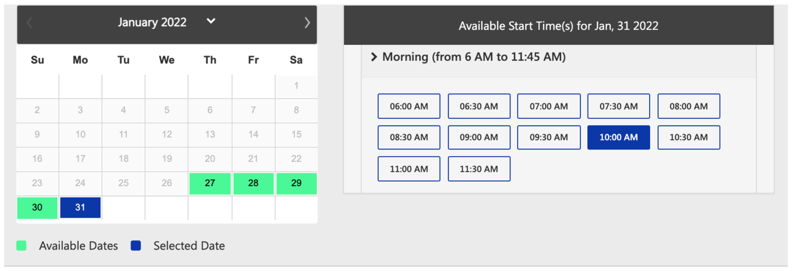

Figure 7.1

Once booked, the exam will show up in the dashboard of your Linux Foundation portal where you can reschedule if required.

You should practice logging in to your Linux Foundation portal on the machine you plan to use for the exam, and you should run the built-in system checker to ensure your machine, including video and audio, meet requirements. When checking your system requirements, you may need to speak during the audio test to prove your microphone works.

The exam will be delivered through a proprietary secure browser that does not work on mobile devices. This may include devices such as the iPad Pro, so you should be sure to check this and have another machine available in case the secure browser doesn't work.

Taking the exam

The most important thing about your exam day is to have everything ready in advance and have as much fun as possible. Don't stress, relax and enjoy the occasion.

With this in mind, you should complete all exam pre-requisites before the day of your exam so you can chill on the day and not have to worry about completing long lists of checks.

The high-level process for taking your exam is as follows. I'm including recommended steps to make your exam day experience as smooth as possible:

- Make sure you have your authorised form of ID available
- Get a good night sleep the night before
- Clear your schedule at least one hour before the exam
- Prepare the room according to exam requirements
- Choose where you're going to sit and make sure it meets exam requirements
- Make sure you eat and take any required comfort breaks before you start so you're comfortable during it
- Prepare a clear drink in a clear container in case you need to drink during the exam
- Reboot the machine you'll be using to take the exam
- Close any unnecessary background processes that might interfere or consume network bandwidth
- Make sure nobody in the house is streaming or gaming over the network
- Log on to your Linux Foundation portal and click the `Take Exam` button (you can start 15 minutes before your official start time)

When you click the `Take Exam` button you'll download the exam's secure browser. You then need to use your webcam to take several still photos and videos of your surroundings and upload them to the system. This can take around 10 minutes, and you should relax while you do this as you need to be calm and relaxed for the exam.

You'll then be assigned a proctor (person who will monitor your exam) who will communicate with you through a live chat feature in the browser. This is text-based chat and not audio. The proctor will check the photos and videos you uploaded before authorizing you to start the exam. If any of your photos or videos don't meet the requirements, you'll be able to take new ones and upload them.

Once the proctor is happy with your ID and surroundings, they'll authorise you to start the exam.

Beware of your screen resolution during the exam, especially if using a small monitor. I took my exam on a 12-inch MacBook and didn't realise some of the answer options were off the bottom of my screen and I had to scroll to see them. This won't be a problem on most screens but be aware that most questions have 4 possible answers, so be sure to check you don't have answers off the bottom of your screen that you can't see.

If you're unsure about the answer to any question you can mark it for review. You can then proceed with the other questions and revisit those marked for review at the end. Obviously, you have to submit all answers, including those marked for review, before your time is up.

Once you complete the exam, your proctor will come back on the live chat and authorise you to go. You'll then get a short survey to complete.

Getting your result

Despite being an online multiple-choice exam, it's normal for the grading process to take the full 24 hours. As soon as your result is available, you'll get an email and be able to view it in your Linux Foundation portal. Hopefully you'll receive a passing grade and be able to celebrate.

Staying connected

I'm really excited for you to take your KCNA exam and would love to hear about your exam experience and your result.

I’ll also update the book with any tips and recommendations you may have.

Email me at **kcnabook@nigelpoulton.com**

8: Sample test

This chapter contains a sample test. The questions are written in a similar format to the real exam questions, but they are not real exam questions. Posting real exam questions online or in a book is a breach of exam rules and is not something I would ever do. I’m interested in helping you gain real skills and real qualifications – I have no interest in helping anyone cheat to pass exams.

I recommend the following to get the most out of this sample exam.

- Get a pen and paper to record your answers
- Write the numbers 1-60 on the paper before you start
- Write each answer against the question number on the paper
- Time yourself (90 minutes)

The number of questions per topic matches the weight of each topic in the exam. For example, the *Kubernetes fundamentals* domain accounts for 46% of the exam, so 46% of the questions in this test are about *Kubernetes fundamentals.*

The answers to the questions start on page 175, in the paperback, and include explanations.

Good luck and have fun practicing for the exam.

1. Which of the following are important concepts when designing and building resilient systems that self-heal? Choose all correct answers.

- A. Desired state
- B. Reconciliation
- C. Iteration
- D. Observed state

2. You're deploying a new application to an environment running the containerd container runtime and you've been told you need to secure containers as much as possible. Which of the following technologies might help? Choose the two correct answers.

- A. Seccomp
- B. SSL
- C. Capability dropping
- D. Secure Shell

3. What does Kubernetes use as its default and preferred method of service discovery?

- A. Environment variables
- B. Key-value pairs
- C. DNS
- D. IPv6

4. You're about to deploy a new Kubernetes cluster. Which of the following is important for the control plane? Choose two answers.

- A. It runs on-premises
- B. It's highly available
- C. You run at least two control plane nodes
- D. It's high performance

5. You need to ensure your Kubernetes cluster can dynamically add nodes to deal with spikes in performance. You're running on a cloud and you know it supports Kubernetes node pools. Which Kubernetes technology do you need to configure so that node pools can grow and shrink?

- A. The horizontal node autoscaler
- B. The horizontal pod autoscaler
- C. The cluster autoscaler
- D. The vertical autoscaler

6. You need to deploy a tool to store and analyse trace data. You've just been told the technology should also have graduated the CNCF. Which should you deploy?

- A. Rook
- B. Linkerd
- C. Jaeger
- D. Envoy

7. You're a security engineer being asked to give an example of a threat vector that exists for cloud native workloads but not traditional monolithic applications. Which could you use?

- A. Shared kernels
- B. Firewall breaches
- C. Software vulnerabilities
- D. Unpatched servers and applications

8. You're looking to hire a hands-on team player who understands entire applications and application lifecycles and will bring a security-first approach. Which of the following job roles should you advertise?

- A. Cloud Architect
- B. DevSecOps Engineer

- C. Data Engineer
- D. Sysadmin

9. You're building a business case to re-factor an existing application to make it cloud native. What are some of the benefits this will bring? Choose all correct answers.

- A. Application resiliency
- B. Automatic scaling
- C. Less human interaction with changes
- D. Better visibility into what's going on inside the app

10. Which of the following are OCI specifications?

- A. image-spec
- B. distribution-spec
- C. pod-spec
- D. runtime-spec

11. How are service meshes able to augment Kubernetes workloads?

- A. By enforcing KubeBench best practices on all Kubernetes clusters
- B. By injecting a sidecar container into every application pod
- C. By automatically scanning the image of every container
- D. By implementing an automated testing pipeline

12. Which of the following are examples of container runtimes? Choose all correct answers.

- A. ESXi
- B. Docker
- C. Kubernetes
- D. containerd

13. You're heading up a new project building a microservices application on containers. However, you've just learned of strict security requirements and concerns over shared kernels. What can you do?

- A. Ensure the project uses namespaced containers
- B. Ensure the project uses virtualised containers
- C. Ensure the project runs on Linux nodes
- D. Ensure the project runs on FreeBSD nodes

14. Which of the following container runtimes creates every container in its own virtual machine?

- A. Docker
- B. RunC
- C. Kata containers
- D. CRI-O

15. You're about to deploy a new Kubernetes cluster running version 1.23. Which of the following versions of `kubectl` will you be able to safely use to manage it? Choose all correct answers.

- A. 1.19
- B. 1.24
- C. 1.25
- D. 1.22

16. You're designing infrastructure to host microservices apps and you've been told that all container-to-container traffic has to be encrypted. Which of the following will accomplish this?

- A. Force every application to encrypt its own traffic
- B. Use the CRI-O container runtime
- C. Deploy a service mesh
- D. Use Kubernetes network policies

17. Put the following four Cs of cloud native security in order starting with the one you typically have the most control over and finishing with the one you typically have the least control over.

- A. Code, containers, clusters, cloud
- B. Code, clusters, containers, cloud
- C. Cloud, containers, clusters, code
- D. Clusters, containers, cloud, code

18. You're deploying a Kubernetes cluster for some new applications. Which of the following network services does Kubernetes natively support? Choose two.

- A. IP address management (IPAM)
- B. Traffic encryption
- C. Service discovery
- D. Traffic shaping

19. You're deploying a new production Kubernetes environment and management want to know why you're deploying a service mesh. Which of the following would you include in your response? Choose all correct answers.

- A. Service meshes can automatically encrypt traffic
- B. Service meshes can automatically make network traffic more observable
- C. Service meshes can be used to make networks more reliable
- D. Service meshes enable IPv6 traffic

20. You're adding instrumentation to an in-house application to output high-quality telemetry. Where should you expose it so that Prometheus can scrape it?

- A. `/status`
- B. `/prometheus`

- C. `/healthz`
- D. `/metrics`

21. You’re deciding between containers and virtual machines for a new app. Which of the following are true? Choose all correct answers.

- A. You can run more virtual machines on a single server
- B. You can run more containers on a single server
- C. Virtual machines start faster than containers
- D. Containers start faster than virtual machines

22. How many Kubernetes releases happen in calendar year?

- A. 1
- B. 2
- C. 3
- D. 4

23. Which of the following container runtimes implements a proxy kernel layer between a container and the host’s kernel?

- A. Docker
- B. containerd
- C. Kata containers
- D. gVisor

24. You’re deploying a new Kubernetes cluster and need to ensure the control plane is highly available. How many control plane nodes should you deploy?

- A. 1
- B. 3
- C. 4
- D. 6

25. You’re managing a Kubernetes cluster with all nodes running the containerd container runtime. Which of the following might be a concern? Choose all correct answers.

- A. Unable to run OCI containers
- B. Root containers
- C. Doesn’t work with the CRI
- D. Shared kernel

26. You’re responsible for a large, shared environment with many competing customers that need their workloads isolating from each other. You’re about to containerise some of their applications and run them on Kubernetes and you already have a single Kubernetes cluster. What can you do to ensure competing customers workloads are securely isolated from each other?

- A. Create and Namespace for each customer on the existing Kubernetes cluster
- B. Use affinity scheduling rules to target each customer onto their own labelled nodes
- C. Create a new Kubernetes cluster for each competing customer
- D. Use the gVisor container runtime on all nodes

27. You manage a production Kubernetes environment and need to ensure pods are deployed across user-defined failure domains. Which technology will help?

- A. Pod topology spread constraints
- B. Pod HA policies
- C. Pod affinity rules
- D. Network policies

28. You want to divide a Kubernetes cluster so that different departments have their own resource quotas and users. Which technology will help?

- A. KubeVirt
- B. The gVisor container runtime
- C. A quota policy
- D. Namespaces

29. Which of the following is the process for accepting new objects into the Kubernetes API?

- A. sandbox > incubating > graduated
- B. alpha > beta > gamma
- C. alpha > beta > stable
- D. incubating > sandbox > graduated

30. You're deploying a new application. Which of the following types of check will ensure the application is up and running as expected?

- A. Health checks
- B. Prometheus checks
- C. Reconciliation
- D. Metric aggregation

31. Which of the following commands returns the logs of a recently terminated pod called "kcna-pod"?

- A. kubectl logs kcna-pod
- B. kubectl logs kcna-pod -p
- C. kubectl get logs kcna-pod
- D. kubectl get logs kcna-pod -p

32. Which of the following kubectl commands will return detailed information about a pod called "kcna-pod"?

- A. kubectl get pod kcna-pod
- B. kubectl get pod kcna-pod –detail
- C. kubectl describe pod kcna-pod

- D. kubectl get pod kcna-pod –verbose

33. You need to deploy a service mesh and the leadership team at your organisation has decided to only deploy technologies that have graduated the CNCF. Which of the following can you deploy?

- A. Istio
- B. Linkerd
- C. prometheus
- D. Consul

34. You're managing multiple Kubernetes clusters from a single admin machine with kubectl installed and need to tell kubectl about a new cluster you want to manage. How will you do this?

- A. Add the cluster and user credentials to the kubeconfig file
- B. Add the cluster and user credentials to the clusters.yml file
- C. Add the cluster and user credentials to the api.yml file

35. You manage a Kubernetes environment and need to make provisions for a new application that requires each container to run in its own lightweight virtual machine. Which cluster component might you need to change or update to allow this to happen?

- A. Kubelet on some worker nodes
- B. Scheduler on the control plane
- C. Container runtime on some worker nodes
- D. Kube-proxy on some worker nodes

36. You're deploying a new Kubernetes cluster and are deciding how many control plane nodes to implement for high availability. Which of the following is the best choice?

- A. 1
- B. 2
- C. 3

- D. 4

37. What are some of the advantages of hosted Kubernetes? Choose all correct answers.

- A. Hosted Kubernetes is always cheaper than the alternative
- B. You can focus more on running applications
- C. The cloud platform handles upgrades
- D. The cloud platform is responsible for performance and availability

38. You manage a Kubernetes cluster and one of the GA objects your applications are using has just been deprecated. How long do you have until it's no longer supported?

- A. 24 hours
- B. 12 months
- C. 12 weeks
- D. 24 months

39. You need to deploy a stateless application to an existing Kubernetes cluster and ensure it's resilient and can perform rolling updates. Which Kubernetes controller should you use to deploy and manage the application?

- A. Workload controller
- B. StatefulSet controller
- C. StatelessSet controller
- D. Deployment controller

40. You're operating on a public cloud and want Kubernetes to be able to dynamically add and remove nodes from your node pool. Which of the following technologies do you need to configure?

- A. The Kubernetes Cluster Autoscaler
- B. The Kubernetes Cloudscaler

- C. The Kubernetes Multi-dimensional Autoscaler
- D. The Kubernetes Node Autoscaler

41. You need a way to ensure a single instance of a logging agent is running on every worker node in an existing Kubernetes cluster. How can you accomplish this?

- A. Deploy it via a DaemonSet controller
- B. Deploy it via a node controller
- C. Deploy it via a Deployment controller
- D. Deploy it via a logging sidecar

42. Which container runtime is based on the core functionality from the Docker engine?

- A. CRI-O
- B. Kata containers
- C. Swarm containers
- D. containerd

43. You've described a new application in a declarative Kubernetes manifest file called "kcna-app.yml" and want to deploy it with `kubectl`. Which command should you run?

- A. kubectl run -f kcna-app.yml
- B. kubectl deploy -f kcna-app.yml
- C. kubectl post -f kcna-app.yml
- D. kubectl apply -f kcna-app.yml

44. You're concerned about the networking requirements of some applications you're in the process of containerising and deploying to Kubernetes. Which of the following network-related technologies does Kubernetes provide out-of-the-box?

- A. IPAM
- B. Service registration and discovery

- C. Traffic encryption
- D. Deep observability

45. What is the atomic unit of scheduling on a Kubernetes cluster?

- A. Container
- B. Pod
- C. Deployment
- D. Function

46. An application microservice in your live production environment is down and you need to troubleshoot the root cause. Which class of telemetry data is the most likely place you'll find the root cause?

- A. Metrics
- B. Application logs
- C. Audit logs
- D. Trace data

47. One of your development teams is writing an in-house application that creates and uses persistent data and they want to run it on Kubernetes. Which of the following Kubernetes controllers might be appropriate for the app?

- A. Workload controller
- B. StatefulSet controller
- C. StatelessSet controller
- D. Deployment controller

48. Why does Kubernetes implement higher-level controllers instead of deploying static pods?

- A. They provide advanced network telemetry
- B. They add features such as resiliency, scaling, and updates
- C. They allow pods to be migrated between different cloud platforms

- D. They run pods faster

49. How do Kubernetes controllers operate?

- A. As key-value stores
- B. Client-server architecture
- C. As RESTful objects exposed over HTTPS
- D. As background reconciliation loops

50. Why is serverless called serverless if it uses servers?

- A. It uses “less” servers
- B. The servers are so well hidden
- C. All work runs on a single large Mainframe
- D. It only works in the public cloud

51. You’re deploying a new GitOps workflow that deploys to production upon successful merge. Which of the following common practices will be an anti-pattern when your GitOps workflow goes live?

- A. Leaving manual comments in the PR comments forum
- B. Making manual changes to the live environment
- C. Pushing builds with the deploy label
- D. Deploying Kubernetes objects via declarative manifests

52. You have a small Kubernetes environment and need a GitOps tool that’s easy to deploy and learn how to use. Which of the following is a good tool to choose?

- A. Jenkins
- B. ArgoCD
- C. CircleCI
- D. FluxCD

53. You’re about to deploy a new application and the developers want to be able to push reliable frequent changes to production. How can you help?

- A. By using Kubernetes pods
- B. By building a CI/CD pipeline
- C. By enabling IPv6 on the pod network
- D. By building a service mesh

54. You need to make your developers more productive, but you’re not allowed to automate software deployment to live environments. Which tools or processes can help?

- A. CD
- B. GitOps
- C. CI
- D. Jaeger

55. You’ve been tasked with choosing a new CI/CD tool for your environment. Which tools should you consider? Choose all correct answers.

- A. Istio
- B. Jenkins
- C. GitLab
- D. Linkerd

56. You have an application and a helper service that need to be able to share data. Which Kubernetes object provides shared volumes and co-locates workloads?

- A. A container
- B. The pod network
- C. A service mesh
- D. A pod

57. You're configuring a new app and want to feed it the best type of data to make autoscaling decisions. Which class of telemetry is best?

- A. Metrics
- B. Application logs
- C. Audit logs
- D. Trace data

58. You're deploying some new applications that write logging data to non-standard locations. You've been tasked with implementing a logging solution that ships all log data to a central location for aggregation and makes logs available to the `kubectl logs` command. Which solution will do this?

- A. A node-level logging agent that redirects logs to its own stdout stream
- B. A sidecar with logging agent that pushes directly to the central repository
- C. Get the application to ship all logs directly to the central repository
- D. Run kube-proxy on every node

59. Which of the following is a common format for writing application log data?

- A. OpenLogging
- B. YAML
- C. CSV
- D. JSON

60. You're planning a new serverless application and want it to run on Kubernetes and be open-source. Which of the following projects are suitable? Choose two answers.

- A. KubeVirt

- B. OpenFaaS
- C. Knative
- D. OpenServerless

Congratulations. You’ve finished the practice exam. If you’ve completed the book and score more than 75% on this practice exam you should be ready to take your KCNA exam.

To score 75% or more you need to have answered 45 or more questions correctly.

Appendix A: Chapter quiz answers

This appendix lists the answers to all the in-chapter quizzes. See Appendix B for the answers to the Sample Test.

Chapter 1: Setting the scene

Question 1, answers A, B, D

Answer C is **not** correct because containers are **less secure** than virtual machines. This is due to the shared kernel model.

Question 2, answer B

Every virtual machine has its own kernel and is therefore more secure than a container. However, this makes them bigger, slower to start, and not a good fit for microservices architectures.

Question 3, answer D

Each container is a virtual operating system with its own root filesystem, its own process tree, and its own eth0 network interface.

Question 4, answers A, D

Containers virtualise operating system constructs such as process trees and filesystems. They do not virtualise hardware constructs such as CPUs and hard drives.

Question 5, answer A

Containers all share the OS and kernel of the host they're running on. This makes them smaller than virtual machines and means you can run more containers on a host than virtual machines.

Question 6, answer D

Containers share the hosts kernel meaning a single compromised container can potentially compromise all other containers. A compromised host kernel also compromises all containers on the host.

Question 7, answers A, B

A container image comprises application source code and all dependencies.

Question 8, answer B

Monolithic applications code all features into a single program.

Question 9, answers B, C

Monolithic applications code all features into a single program. This means updating one feature requires the entire application to be taken down and updated. It also means every feature of the application scales with every other feature – everything is very tightly coupled.

Question 10, answers A, C, D

Microservices applications have every application feature coded as its own small, independent, single-purpose microservice. They're all loosely coupled. This means every feature can have its own small dev team and can be scaled and patched without impacting other features.

Question 11, answers C, D

The distributed nature of lots of small loosely-coupled microservices means there's a lot more network traffic and that traffic is more dynamic.

Chapter 2: Cloud native architecture

Question 1, answer B

Resiliency is the ability for applications and infrastructure to self-heal.

Question 2, answer C

Applications and infrastructure can both be resilient. Kubernetes implements controllers, such as the deployment controller, to make applications resilient. Many cloud platforms allow you to build node pools that are resilient.

Question 3, answer A

Things fail all the time. Designing with failure in mind allows applications to continue working when things fail.

Question 4, answer C

Loose coupling means different components aren't reliant on each other and can be less impacted when other components fail.

Question 5, answer D

Reconciliation is the process of synchronising observed state with desired state.

Question 6, answer A

`/healthz` is a common endpoint for application health checks and probes.

Question 7, answer D

Health checks are a good way to test whether an application is functioning as expected.

Question 8, answer A

Desired state is a record if intent – what you want.

Question 9, answers B, C, D

Autoscaling is the ability for applications and infrastructure to scale up and down without human intervention.

Question 10, answer A

Cloud infrastructure costs money to run. If you run infrastructure you're not using you'll still be paying for it.

Question 11, answer C

Individual microservices of an application can be scaled up and down without impacting other microservices. "A" is not an advantage of microservices because monolithic applications can also scale up and down.

Question 12, answer A

Horizontal scaling adds and removes instances of something you already have. For example, the Kubernetes horizontal pod autoscaler adds or removes more instances of a pod. The Kubernetes cluster autoscaler adds or removes more instances of a particular node type.

Question 13, answer B

Cloud native apps prefer horizontal scaling, sometimes referred to as *scaling out.*

Question 14, answers A, D

Kubernetes implements a horizontal pod autoscaler (HPA) for scaling applications, and a cluster autoscaler (CA) for scaling cluster nodes.

Question 15, answers B, C

Serverless is an event-driven computing model that still uses servers. The servers are so abstracted and hidden from the developer that we call it "serverless".

Question 16, answer D

Many serverless platforms let you upload functions packaged as container images.

Question 17, answer A

Serverless functions are small, single-purpose workloads that follow microservices design patterns.

Question 18, answers A, C

Containers are small and ideal for microservices applications, making them ideal for serverless workloads.

Question 19, answer A

Many serverless platforms charge you each time a serverless function executes.

Question 20, answer B

CloudEvents is attempting to bring standards to the serverless ecosystem.

Question 21, answer C

Serverless functions automatically start when an associated event occurs, and they exit when they complete. This is a native form of scaling.

Question 22, answer B

Serverless workloads run small functions when an event triggers them. It's commonly called Function as a Service (FaaS).

Question 23, answer D

The CNCF was founded under the umbrella of the Linux Foundation in 2015 to bring governance and direction to a new breed of open-source container-related technologies. The Linux Foundation was founded in 2000.

Question 24, answers A, B, D

Projects enter the CNCF as "sandbox" projects, progress through the "incubating" stage, and eventually "graduate".

Question 25, answers A, B, C

CNCF projects don't have to have financial backing from at least two vendors.

Question 26, answers A, C

Most cloud platforms operate a pay-as-you-use model that most organisations classify as an operational expenditure (opex).

Question 27, answers C, D

DevOps engineers are expected to understand the entire applications and their entire lifecycles. They're also expected to have some experience as a developer and operator. Typically, the developer/operator split is somewhere along the lines of 80/20.

Question 28, answer D

CloudOps is very similar to DevOps but focusses almost entirely on cloud-only technologies and skills. For example, a DevOps engineer may have skills with one or two open-source key-value store technologies (NoSQL) that can be used on premises or in the cloud, whereas a CloudOps engineer might have skills with a particular cloud's proprietary key-value store technology.

Question 29, answer A

Continuous integration, continuous delivery (CI/CD) tools are core DevOps tools.

Question 30, answer C

Full stack developers are go-getters with a lot of skills that just get stuff done.

Question 31, answer A

It's commonly said that SREs should automate themselves out of a job.

Question 32, answers B, C

Open standards in the cloud native world are like standardised rail tracks in the manufacturing and engineering world – they enable interoperable solutions that help avoid lock-in and give developer's confidence the solutions they build will work well and integrate well.

Question 33, answers A, B, C

The Open Containers Initiative (OCI) is responsible for low-level container-related standards and currently maintains the image-spec, runtime-spec, and distribution-spec.

Question 34, answers A, B, C, D

Cloud native applications should be resilient, provide high-quality telemetry for high observability, and be able to autoscale. Things like scaling and self-healing should happen without a human having to get involved.

Chapter 3: Container orchestration

Question 1, answers A, D

Containers are similar to virtual machines. However, they share the host's kernel making them more lightweight than virtual machines. You normally run one small, single-purpose application per container.

Question 2, answer A

Containers are smaller and more lightweight than virtual and physical machines. This means you can run more apps if you use containers.

Question 3, answer A

Containers are smaller and more lightweight than virtual machines, making them easier to package and share. Containers also package application source code and all dependencies meaning they will work the same in production as they do on a developer's laptop – this also makes them better for sharing.

Question 4, answer B

Kubernetes and containers don't care what languages your applications are written in.

Question 5, answer D

A container image holds application code and all dependencies.

Question 6, answer C

As a container includes application code and all dependencies, it means whatever works on a developer's laptop should also work in production. This is because the application doesn't rely on external dependences, such as shared library files, that might be different versions on the developer's laptop and production.

Question 7, answer B

Container runtimes are a class of low-level software that starts and stops containers.

Question 8, answers A, C, D

Container runtimes download container images from registries and start and stop containers. Higher level decisions, such as choosing which node a container should run on, are made by higher level tools such as Kubernetes.

Question 9, answer B

The OCI defines low-level container standards such as the image-spec, runtime-spec, and distribution-spec.

Question 10, answer B

An OCI image is a container image built according to the OCI image-spec.

Question 11, answer D

Docker builds images that comply with the OCI image-spec.

Question 12, answers A, B, D

Namespaced containers are the most popular and operate the shared kernel model. Sandboxed containers insert a proxy layer between the container and the host kernel. Virtualised containers run each container in its own lightweight virtual machine.

Question 13, answers B, C, D

Docker, CRI-O, and containerd all create namespaced containers. Kata containers creates virtualised containers.

Question 14, answer D

Kubernetes cannot create containers. It has to use a container runtime to do this. Kubernetes handles higher-level operations such as scheduling and scaling.

Question 15, answer A

Most modern Kubernetes clusters default to using containerd as their container runtime.

Question 16, answer B

The Container Runtime Interface (CRI) is the low-level technology implemented by Kubernetes to make the container runtime layer pluggable.

Question 17, answer A

The OCI is responsible for low-level container-related standards. The CRI makes the Kubernetes container runtime layer pluggable.

Question 18, answers A, B, D

Container orchestrators, such as Kubernetes, can schedule, scale, and self-heal containers. Build tools are responsible for creating container images.

Question 19, answer C

Kubernetes is the industry-standard container orchestrator and by far the most popular container orchestrator in use today.

Question 20, answer C

A Kubernetes cluster comprises one or more control plane nodes and worker nodes.

Question 21, answers A, B, C, D

Root containers can potentially cause damage to the shared container host. Unsecured networks make container-to-container traffic easy to snoop. Untrusted images can contain malicious code. Shared kernels increase the risk of a compromised container affecting other container or even the host it’s running on.

Question 22, answer B

User namespaces is a Linux kernel technology that maps the root user ID inside a container to a different, less powerful, user on the shared host. This means a compromised root process in a container will not have root access on the shared host.

Question 23, answer D

The four Cs of cloud native security are code > containers > clusters > clouds

Question 24, answers A, B, C, D

Containers are typically microservices applications that send internal application traffic over the network. This means network security is vital. Each container (microservice) needs to know how to discover other services. Self-healing, scaling, and update operations all add and remove containers from the network, this includes IP addresses and causes a lot of IP churn. Tracing requests as they are handed off between lots of different microservices is vital to understanding how applications work and how traffic flows.

Question 25, answer B

Container-to-container traffic happens over the same network and is referred to as east-west traffic. There is lots of it in microservices architectures where each application feature is deployed in its own container and needs to share data with other containers.

Question 26, answers A, B, C

Self-healing, autoscaling, and rollouts all cause containers to be added and removed from the network. This means IP addresses are being added and removed frequently.

Question 27, answer A

The Kubernetes pod network is open and flat. This means all containers can communicate by default but is a security risk and should be locked down.

Question 28, answer C

The Container Network Interface (CNI) is a low-level technology implemented by Kubernetes to make the networking layer pluggable.

Question 29, answer D

East-west traffic refers to traffic between containers on the same network.

Question 30, answer B

Service meshes implement features at the *platform layer.* This is another way of saying any container on the service mesh automatically gets the features without having to change the configuration of applications or containers.

Question 31, answers B, C

Istio and Linkerd are popular service meshes. Kubernetes is a container orchestrator and Prometheus is a monitoring tool.

Question 32, answer B

The Container Storage Interface is a low-level technology implemented by Kubernetes to make the storage layer pluggable.

Question 33, answers A, B

Seccomp, SELinux, and AppArmor all allow you to secure containers. Secure shell gets you a secure command line session on a remote system, whereas KubeVirt lets Kubernetes orchestrate virtual machine workloads.

Chapter 4: Kubernetes fundamentals

Question 1, answer D

Kubernetes was released in 2014.

Question 2, answers A, B, D

Kubernetes is a container orchestrator and can add self-healing, autoscaling, and zero-downtime rolling-updates to containerised applications. Image scanning is usually provided by registries or CI/CD pipelines.

Question 3, answers A, B, D

Kubernetes natively orchestrates containers. KubeVirt extends it to be able to orchestrate virtual machines, whereas OpenFaaS and Knative extend it to be able to orchestrate serverless functions.

Question 4, answer B

KubeVirt enables Kubernetes to manage virtual machine workloads inside of pods. Knative and OpenFaaS are for serverless, whereas Cilium is a network plugin.

Question 5, answers A, D

OpenFaaS and Knative extend Kubernetes for serverless function workload orchestration. Cilium is a network plug, and KubeVirt enables virtual machine workloads on Kubernetes.

Question 6, answer B

Linkerd provides service mesh functionality. Prometheus is a monitoring tool, OpenFaaS is serverless, and Harbor is a registry.

Question 7, answer D

A Kubernetes pod is a thin wrapper around one or more containers that are mandatory if a container wants to run on Kubernetes.

Question 8, answer D

Containers can only run on Kubernetes if they're wrapped in a pod.

Question 9, answers A, B, C

Pods allow multiple containers to be co-scheduled to the same node and share the same execution environment (memory and volumes). Containers in the same pod do not share the same container image as they should be different, but complimentary, containers. For example, a web container running the NGINX image and a helper container running a Git synchroniser image.

Question 10, answers A, B, D

This is similar to the previous question. All containers in the same pod will run on the same node and share access to the same volumes and network stack. They will not share the same container image.

Question 11, answers B, C

A Kubernetes cluster is made up of control plane nodes and worker nodes.

Question 12, answers B, D

The API server and scheduler run on the Kubernetes control plane. Kubectl is the Kubernetes command line utility and you normally install it on your laptop or an admin workstation. Kube-proxy handles node-level networking and while it technically runs on control plane nodes as well as worker nodes, it's not considered part of the control plane.

Question 13, answer C

All control plane services run on all control plane nodes for high availability.

Question 14, answer C

You should always deploy an odd number of control plane nodes. 3 is the most common, but 5 is also common.

Question 15, answer D

Kubectl is the Kubernetes command line utility.

Question 16, answer B

The cluster store is the only stateful control plane service and is usually based on the `etcd` distributed database.

Question 17, answer A

Existing applications will keep running if the control plane goes down. However, you won’t be able to manage them or deploy more apps until the control plane comes back up.

Question 18, answer D

Kubectl sends all requests to the API server.

Question 19, answer B

Kubernetes controllers run as reconciliation loops that ensure observed state is always in sync with desired state.

Question 20, answer D

The deployment controller manages stateless workloads on Kubernetes.

Question 21, answer C

The statefulset controller manages stateful workloads on Kubernetes.

Question 22, answer A

The daemonset controller ensures an instance of a pod runs on every worker node in the cluster.

Question 23, answers C, D

Current state and *actual state* are both terms that mean the same thing as *observed state.*

Question 24, answer C

Reconciliation is the process of bringing observed state back into sync with desired state.

Question 25, answer B

Kubernetes controllers operate as background control loops that bring observed state into sync with desired state.

Question 26, answer A

Kube-proxy is responsible for implementing local networking rules on cluster nodes.

Question 27, answer D

The Container Runtime Interface (CRI) is the tool that makes the Kubernetes container runtime layer pluggable.

Question 28, answer A

Containerd is the most popular container runtime on modern Kubernetes clusters. Docker is more popular on older clusters but was deprecated in Kubernetes 1.20 and removed in 1.24.

Question 29, answer C

Docker was deprecated in Kubernetes 1.20 and removed in 1.24.

Question 30, answer C

The kubeconfig file is normally called "config" and stored in the hidden ".kube" folder of your profile. It configures the local copy of the `kubectl` command line tool.

Question 31, answer D

`kubectl describe pod kcna` will return detailed info on a pod called "kcna".

Question 32, answers B, C

Kubectl should be no more than one major version higher or lower than the version of Kubernetes on your API server.

Question 33, answer A

A pod will remain in the pending state until the scheduler can find a node capable of running it.

Question 34, answer C

Pod topology spread constraints let you use labels to define failure zones and spread pods across them for high availability.

Question 35, answer D

Kubernetes namespaces divide a single Kubernetes cluster into multiple virtual clusters called namespaces. Each namespace can have its own resource quota and user accounts.

Question 36, answer B

All Kubernetes objects are defined in the API.

Question 37, answer C

Kubernetes releases three new major version per year (every four months).

Question 38, answer A

Kubernetes deploys all pods to the "pod network".

Question 39, answer D

Kubernetes implements the Container Network Interface (CNI) to make the networking layer pluggable.

Question 40, answer A

Cilium is the most popular Kubernetes CNI network plugin and uses eBPF for advanced networking.

Chapter 5: Cloud native application delivery

Question 1, answers A, C

“Release” and “rollout” both refer to the process up replacing an existing version of an app with an updated version.

Question 2, answer C

Continuous Integration (CI) allows developers to automate the building and testing of application code.

Question 3, answer A

Continuous Integration (CI) is a great way or identifying bugs and other issues before the code gets to production.

Question 4, answer A

Continuous Deployment (CD) is the process of automating the deployment of updates to live environments.

Question 5, answer C

CI/CD enables frequent releases and is a cornerstone of DevOps workflows.

Question 6, answer D

Jenkins is a popular CI/CD tool. Kubernetes and Swarm are both container orchestrators and Docker is a company, container orchestrator, and container runtime.

Question 7, answer B

Using CI/CD tools to automate building and testing allows developers to spend more time building applications.

Question 8, answers A, C, D

GitOps workflows integrate infrastructure as code, pull requests, and continuous delivery.

Question 9, answers A, B, D

Infrastructure as code lets you define policies, virtual networks, Kubernetes objects, and other cloud-based or virtual infrastructure components. It cannot be used to perform physical tasks such as racking servers and pugging in cables.

Question 10, answer B

Infrastructure as code works in on-premises and public cloud environments. However, as public cloud environments tend to be highly virtualised, IaC works better on them.

Question 11, answer B

Kubernetes objects are typically defined in YAML files.

Question 12, answer A

Pull requests and merges are used to trigger the CD steps of a GitOps workflow.

Question 13, answers A, B

ArgoCD and FluxCD are both popular GitOps tools.

Question 14, answer D

You should not make manual changes once you've started using GitOps processes as they'll be overwritten each time a PR is merged and the CD pipeline kicks in.

Question 15, answer D

Linus Torvalds created the Git version control system that is used by GitHub and other platforms.

Question 16, answer B

GitOps tools only care about deployment to live environments and aren't concerned with build stages.

Question 17, answer A

In a GitOps workflow all manual changes can be overwritten with every PR and subsequent merge.

Question 18, answer C

An anti-pattern is something that seems like a good idea when you do it but turns out to be a bad idea later. A simple example is making a manual fix in a live environment to fix an issue. However, the fix is overwritten later when the automated process kicks in.

Question 19, answer A

FluxCD is known for being simple to configure and use.

Chapter 6: Cloud native observability

Question 1, answer D

Telemetry is log data, metrics, and traces that are output from a system.

Question 2, answers A, B, C

The three main classes of telemetry data are logs, metrics, and traces.

Question 3, answer D

Prometheus stores data in a timeseries format.

Question 4, answer B

Telemetry is used to troubleshoot and tweak performance of a system.

Question 5, answers B

The OpenTelemetry projects provides instrumentation and specifications for cloud native observability.

Question 6, answer B

Log data is application output that is usually output as text or JSON.

Question 7, answer C

Log events are often categorised from lowest to highest severity in the following order: `info` > `warning` > `error` > `critical`.

Question 8, answer A

High verbosity generates more output than low verbosity.

Question 9, answer C

Debug level is the most verbose and intended to dump out as much data as possible.

Question 10, answers A, B

Kubernetes supports cluster-level logging via an agent on each node, or an agent in a sidecar container.

Question 11, answers B, C

The local kubelet on each worker node monitors a container's stdout and stderr streams for logs and makes them available to the `kubectl logs` command.

Question 12, answers A, B

Logging agents can convert log data into the format required by your central logging system.

Question 13, answer A

Cloud native applications should be decoupled from their environment configuration. This allows apps to run in multiple environments without having to be changed. You can inject environment configuration data at runtime.

Question 14, answer A

The `-p` flag returns the logs from the previous instance of a pod/container.

Question 15, answer D

Metrics data tracks system performance over long periods of time, making it a great way to see historical trends and predict future trends.

Question 16, answer C

Custom metrics are metrics tailored to applications and therefore the best way to see the health and performance of an application.

Question 17, answer A

Tracing is the art of tracking requests as they bounce around a distributed system.

Question 18, answer C

Prometheus is considered the industry-standard monitoring system for Kubernetes.

Question 19, answer D

Grafana is commonly paired with Prometheus to provide best-in-class dashboards and visualisations.

Question 20, answer B

Prometheus expects metrics to be exposed over HTTPS on the `/metrics` endpoint.

Question 21, answer C

Scraping is the term Prometheus uses to refer to the process of gathering logs from remote systems.

Question 22, answer B

Prometheus uses a pull model for scraping logs from remotely monitored systems.

Question 23, answer A

Prometheus stores data in timeseries format.

Question 24, answer C

Prometheus provides a pushgateway for short-lived apps that might have terminated before Prometheus scrapes its metrics. The app pushes metrics to the pushgateway and Prometheus scrapes the metrics from there.

Question 25, answer B

Prometheus provides client libraries for most programming languages that you can use to instrument in-house apps for Prometheus integration.

Question 26, answer C

If you don’t have access to the source code of 3rd-party apps, you won’t be able to directly instrument them via client libraries. In these situations, you can run a Prometheus exporter in a sidecar container that will reformat data for Prometheus.

Question 27, answers A, B, D

It’s vital in cloud native environments that you choose the right infrastructure and platforms, rightsize things, and turn things off or delete them when you’re not using them.

Question 28, answer B

Spot instances are the cheapest type of cloud instance and are ideal for services that aren’t business critical and can easily be rebuilt or re-deployed.

Question 29, answer C

Reserved instances get you discounted pricing on cloud infrastructure in exchange for a long-term commitment.

Question 30, answer B

Autoscaling ensures you always have the right amount of infrastructure to run an app against current demand. As such it is constantly right-sizing your infrastructure.

Question 31, answers B, C

When configuring autoscaling it’s vital you watch the right metrics as these are what the system uses to make scaling decisions. If you’re monitoring the wrong metrics the system will make the wrong decisions. It’s also important you set upper and lower bounds to keep costs and usage under control.

Appendix B: Sample Test answers

This appendix contains the answers to the Sample Test.

Question 1, answers A, B, D

Self-healing in Kubernetes is based on controllers that operate in reconciliation loops ensuring observed state is always in sync with desired state.

Question 2, answers A, C

Seccomp and Capability dropping are Linux kernel-related technologies that can be used to secure the kinds of access namespaced containers have to the host's kernel.

Question 3, answer C

Kubernetes has an internal DNS service it uses for service registration and service discovery.

Question 4, answers B, D

It's always important the Kubernetes control plane is high performance and highly available.

Question 5, answer C

The Kubernetes cluster autoscaler (CA) integrates with underlying cloud platforms to dynamically add and remove worker nodes to deal with demand.

Question 6, answer C

Jaeger is a popular tool for storing and analysing trace data. It's also a graduated CNCF project.

Question 7, answer A

Containers normally share the kernel of the host they're running on. This means compromised containers can impact the host and all other containers running on the host. It also means a compromised host can impact all containers.

Question 8, answer B

DevSecOps Engineers bring a security-first approach to DevOps engineering.

Question 9, answers A, B, C, D

Cloud native applications and infrastructure are resilient, can automatically scale, and provide extensive observability. Operations like self-healing and autoscaling happen without human intervention.

Question 10, answers A, B, D

The OCI is responsible for low-level container-related specifications. These include the image-spec, the runtime-spec, and the distribution-spec.

Question 11, answer B

Service mesh technologies inject a sidecar container into every application pod that intercepts network traffic and provides advanced services such as encryption and telemetry.

Question 12, answers B, D

Docker and containerd are examples of container runtimes. ESXi is a lightweight hypervisor and Kubernetes is a cloud native orchestrator.

Question 13, answer B

Virtualised containers avoid the security concerns of shared kernels by running each container in its own small virtual machine.

Question 14, answer C

Kata containers is a container runtime that creates every container in a small, lightweight virtual machine.

Question 15, answers B, D

Kubectl should be no more than one version above or below the version of Kubernetes you're running.

Question 16, answer C

Service meshes are a common way to encrypt traffic in Kubernetes environments.

Question 17, answer A

Generally speaking, you have full control over your own application code, relatively less over your containers, even less over your clusters, and hardly any over your cloud.

Question 18, answers A, C

Kubernetes provides out-of-the-box IPAM and service registration/discovery.

Question 19, answers A, B, C

Service meshes provide traffic encryption, network observability, and network reliability, at the platform layer.

Question 20, answer D

Prometheus expects telemetry to be available on the `/metrics` endpoint.

Question 21, answers B, D

Containers are smaller and start faster than virtual machines. As they’re smaller you can run more containers on a server than virtual machines.

Question 22, answer C

A new version of Kubernetes is released every 4 months. This means there are three new releases per year.

Question 23, answer D

gVisor is a container runtime that implements a proxy kernel between containers and the kernel of the host they’re running on.

Question 24, answer B

You should always deploy an odd number of control plane nodes for high availability. 3 is the most common number, but 5 is also quite common.

Question 25, answers B, D

containerd runs namespaced containers meaning all containers on the same host share the same kernel. This is a security concern for a lot of reasons, but also means compromised root containers can potentially impact the host and all other containers running on the same host.

Question 26, answer C

The only way to secure competing workloads on Kubernetes is to run each in its own Kubernetes cluster. Namespaces are not a secure workload boundary.

Question 27, answer A

Pod topology spread constraints let you spread user workloads across failure zones.

Question 28, answer D

Kubernetes namespaces are a good way to divide a cluster among non-hostile workloads. Each namespace gets its own resource quota and set of user accounts.

Question 29, answer C

New Kubernetes objects come into the API as alpha resources, progress through beta, and are eventually promoted to stable (aka "GA").

Question 30, answer A

Kubernetes provides a mechanism for health checks to probe if an application is up and running as expected.

Question 31, answer B

The `kubectl logs` command returns container logs. Adding the `-p` flag returns logs from the most recently terminated pod.

Question 32, answer C

The `kubectl describe` command is used to return detailed information about objects.

Question 33, answer B

Linkerd is a graduated CNCF service mesh. Istio is also a service mesh but is not a CNCF project.

Question 34, answer A

kubeconfig is the kubectl configuration file.

Question 35, answers C

Container runtimes are in charge of running containers. Some container runtimes, such as Kata containers, run each container in its own virtual machine.

Question 36, answer C

You should always deploy an odd number of control plane nodes for high availability. Three is the most common, and one is better than two.

Question 37, answers B, C, D

Hosted Kubernetes is a consumption model where your cloud provider builds and manages the control plane of you Kubernetes cluster. This includes managing performance, availability and upgrades. As you don’t have to worry about any of this, you can focus more on your applications.

Question 38, answer B

Kubernetes guarantees GA objects will be supported for at least 12 months after deprecation.

Question 39, answer D

The Kubernetes deployment controller manages stateless applications and adds self-healing, scaling, and rolling updates.

Question 40, answer A

The Kubernetes cluster autoscaler (CA) integrates with underlying cloud platforms to grow and shrink node pools based on demand.

Question 41, answer A

The daemonset controller ensures an instance of a particular pod is always running on each cluster node.

Question 42, answer D

containerd was created by Docker, Inc. and donated to the CNCF as a lightweight container runtime suited to environments such as Kubernetes.

Question 43, answer D

The `kubectl apply` command is commonly used to deploy objects from YAML manifest files.

Question 44, answers A, B

Kubernetes provides native IPAM and service discovery. For observability and traffic encryption you need a service mesh.

Question 45, answer B

The pod is the smallest unit of work you can deploy to a Kubernetes cluster.

Question 46, answer B

Application logs are a great place to look when things go wrong. This is because they list detailed application-related events that often include things like service restarts, crashes, failed authentication events, network timeouts, and other things that cause problems.

Question 47, answer B

The statefulset controller manages applications that read and write persistent data. It provides self-healing, sclaing, persistent naming, and ordered startups.

Question 48, answer B

Kubernetes implements higher level controllers to add cloud native features such as self-healing, autoscaling, and rolling updates.

Question 49, answer D

Kubernetes controllers operate as background reconciliation loops keeping observed state in sync with desired state.

Question 50, answer B

Serverless architectures hide the servers so well it’s as if there are no servers.

Question 51, answer B

Making manual changes to environments you’re managing with GitOps is an anti-pattern as manual changes are overwritten with each PR and resulting merge.

Question 52, answer D

FluxCD is well-known for being simple to deploy and use.

Question 53, answer B

CI/CD pipelines can automate all tasks from building and testing, all the way through to deployment in live production environments.

Question 54, answer C

Continuous integration (CI) tools automate building and testing but don’t deploy to live environments.

Question 55, answers B, C

Jenkins and GitLab are popular CI/CD tools. Istio and Linkerd are service meshes.

Question 56, answer D

A Kubernetes pod is a shared execution environment. This means all containers running in the same pod have access to the same environment, including volumes, memory and routing table.

Question 57, answer A

Metrics are usually the best form of telemetry for making scaling decisions. This is because metrics are performance related and measured over time and can display and even predict some future performance issues.

Question 58, answer A

A node-level logging agent that ships logs to its own stdout makes them available to `kubectl logs`. They can then also be pushed to a central repository by the agent or another agent.

Question 59, answer D

JSON is a common format for writing application log data.

Question 60, answer B, C

OpenFaaS and Knative are open-source serverless platforms for Kubernetes.

What next

If you’re about to take your KCNA exam, good luck and I’m excited for you. I’d love to hear about your experience, and I’d love any feedback that could make the book better.

If you’ve already taken it, I hope you passed and are ready to take your next steps.

Other exams

If you work in a hands-on technical role, the following Kubernetes-related exams are obvious next steps.

- Certified Kubernetes Application Developer (CKAD)
- Certified Kubernetes Administrator (CKA)
- Certified Kubernetes Security Specialist (CKS)

All three are hands-on exams and quite a bit harder than the KCNA. This is because they’re specialist certifications and go into a lot more detail. They’re also hands-on and not multiple-choice.

The CKAD is a hands-on exam focussed on deploying and managing applications on Kubernetes. *The CKA* is similar but focusses more on things like building and managing Kubernetes clusters. *The CKS* is obviously security focussed.

Outside of CNCF Kubernetes-related exams, there are several cloud exam tracks that specialise in cloud technologies such as AWS, Azure, and Google Cloud.

Books

I have two other best-selling Kubernetes-related books that you'll love:

- Quick Start Kubernetes
- The Kubernetes Book

Both are available on Amazon, Leanpub, and many other reputable publishers.

Quick Start Kubernetes is aimed at people that are new to Kubernetes. It's only ~100 long and explains Kubernetes from the ground-up with really easy hands-on examples. In 100 pages you'll build a Kubernetes cluster, deploy a simple app, break it and watch Kubernetes self-heal it, scale it, and perform a rolling update. When you're done, you'll be all-good with the basics of Kubernetes.

The Kubernetes Book is an entirely different beast. It covers a lot more and goes into a lot more detail. It's regularly listed as a best-seller.

Both books are updated annually to make sure they're up-to-date with the latest versions of Kubernetes and the latest trends in the cloud native ecosystem.

Video courses

I've created a lot of containers and Kubernetes video courses available at Pluralsight.com. They're a lot of fun and have rave reviews. If you like video courses, you'll love these. Some of them include:

- Docker & Kubernetes: The Big Picture
- Getting Started with Docker

- Getting Started with Kubernetes
- Docker Deep Dive
- Kubernetes Deep Dive

Let’s connect

I’m always happy to connect and I’d love to hear your opinions on the book and your experience with the exam. You can reach me at all of the following places.

- **LinkedIn:** https://www.linkedin.com/in/nigelpoulton/
- **Twitter:** @nigelpoulton
- **Website:** nigelpoulton.com

Feedback on the book and your exam experience is also welcome at `kcnabook@nigelpoulton.com`.

And one last time… good luck with your exam!