**16. Containerization and Virtualization**

Shivakumar R Goniwada[**1**](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Aff2)

(1)

Bangalore, India

So far, you have learned to design and develop cloud native services from the functional and nonfunctional perspectives. In this chapter, you will learn more about deploying and running your developed binaries.

Infrastructure is all about the software and hardware that supports your applications. This includes data centers, operating systems, networks, automation, security, and the system needed to support the lifecycle of an application.

In this chapter, I will explain the details of running your services in a cloud environment with virtualization, containerization, and orchestration.

IaaS is driven by virtualization; it enables multiple operating systems with different configurations to run on a physical machine. The software layer in a VM is a hypervisor, which is required to run the VM on a system. This hypervisor controls all the hardware resources and can move resources from one VM to another depending on the needs. In this chapter, I will explain how some of the systems need to run on VMs and how they are useful in cloud native elements.

Containerization has become a de facto companion to virtualization for cloud native application services. It involves encapsulating software code and all its runtime dependencies so that the software can run uniformly and consistently on any infrastructure. It allows you to develop and deploy your services quickly and securely. In this chapter, I will cover how you can adopt containerization to run your services.

Containers require operational best practices; however, Kubernetes works as an orchestrator for your containerized applications to manage, scale, and schedule. It helps you fully implement container-based infrastructure in a production environment for your cloud native services. In this chapter, I will cover Kubernetes features, secrets, and configuration with monitoring and deployment.

In addition, I will cover the details of containers and how cloud native applications are deployed in containers and in the Kubernetes environment.

* What applications and services are commonly virtualized?
* Cloud native and virtualization
* Container principles and patterns
* Best practices for adopting containers
* Container as a service (CaaS)
* Kubernetes principles and patterns
* How does Kubernetes solve common cloud native problems?
* Scaling your cloud native application
* Kubernetes as a service (KaaS)
* Observability and metrics on Kubernetes
* The stateful workload on Kubernetes

**Introduction**

Cloud native infrastructure is a requirement to effectively run cloud native applications. Without the right design and practices to manage infrastructure, even the best-designed cloud native services can create issues; therefore, you need to provide equal importance in designing your infrastructure.

Before providing more insight on how to build infrastructure for cloud native, you need to understand how you got where you are.

To execute your cloud native application on the cloud, you can produce value faster and focus on your business objectives. Developing only what you need to create your system and consuming services from cloud providers, keep your lead time small and agility high.

The ephemeral nature of cloud services demands automated development workflows that can be deployed as needed. The services must be designed with infrastructure ambiguity in mind. This has led engineers to rely on infrastructures like VMs, containers, and Kubernetes without having to worry about the underlying resources.

Containerization is a mature technology and adopts rapid changes in the way the services test and run application instances on the cloud. Containerization provides a less resource-intensive alternative to running an application on VMs because containers can share computational resources and memory without requiring a full operating system to underpin each application. Containers house all the runtime components that are necessary to execute an application in an isolated environment including configuration, libraries, etc.

All the major cloud providers offer a container as a service (CaaS) model that manages containers on a large scale, including starting, stopping, scaling, and organizing container workloads. CaaS offers both individual containers without orchestration capabilities and full-featured orchestration like Kubernetes. AWS offers the Amazon Elastic Container Service and Kubernetes services, Azure offers the Azure container and Kubernetes service, and Google offers the Kubernetes engine.

According to a Cloud Native Computing Foundation (CNCF) survey, in 2020, 92 percent of organizations surveyed used Docker containers, and 83 percent used Kubernetes for orchestration. This survey shows the overwhelming adoption of containers and Kubernetes for cloud native architecture.

Kubernetes is an open source orchestration engine developed by Google for managing cloud native services on containers across distributed cluster nodes. It provides a highly resilient infrastructure, provides automatic rollback, is highly scalable, and offers the self-healing of containers. The main objective of Kubernetes is to hide the complexity of managing a cluster of containers by providing APIs for configuration.

Bare-bones Kubernetes is not enough for production applications, because you need key services such as cluster monitoring and logging, reserved compute resources, heartbeats, election timeout, regular etcd backups, etc.

Kubernetes is not only for containers; you can use it for VMs too. In 2019, VMware started supporting Kubernetes as part of vSphere, which includes an ESXi hypervisor. Now it is possible to run containers on ESXi.

Kubernetes as a service (KaaS) is offered by various cloud providers as a managed service. The KaaS services are Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), Red Hat OpenShift, VMware Tanzu, and Docker EE. These services manage Kubernetes for deploying, managing, and maintaining clusters. Each managed service offers customized benefits.

Cloud native integrates cloud computing technologies and enterprise management methods, enabling enterprises to migrate services to cloud platforms more efficiently and quickly.

Cloud native infrastructure is not a solution for every problem; it is your responsibility to know if it is the right solution for your system environment.

**What Is Cloud Native Infrastructure?**

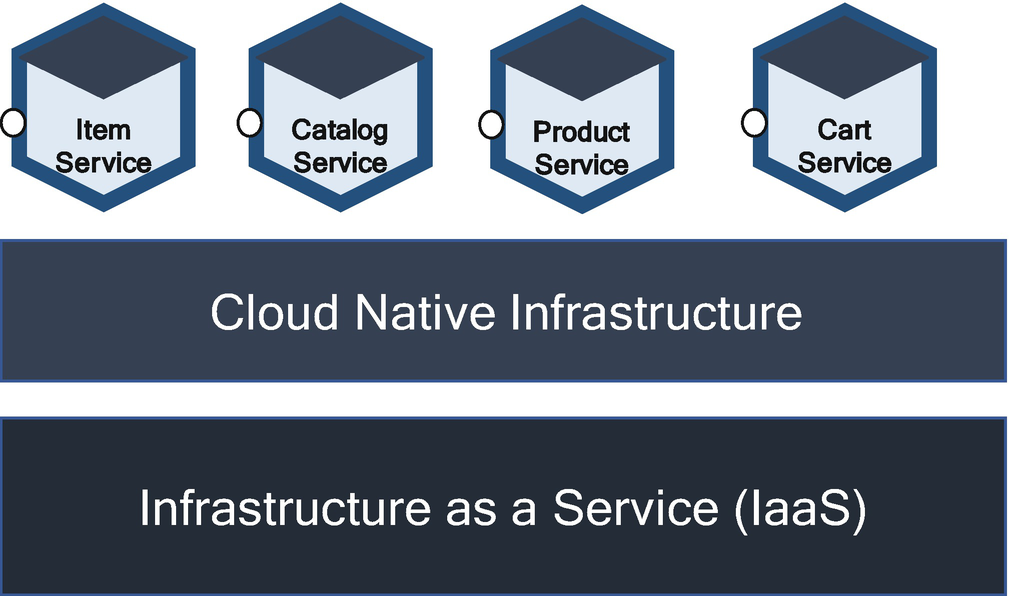
Cloud native infrastructure not only runs your applications in cloud infrastructure but does much more than that. The procedure to use IaaS is no different than running virtual servers on your data center.

You may think that because you have developed your services with microservices principles, used DevSecOps, and deployed in containers and orchestrator, this is cloud native. However, that is not correct. This is not the entire cloud native story. It is the first step, but still, there is a lot of work to be done to adhere to cloud native principles.

Cloud native is not just about running your services in containers and implementing Kubernetes orchestration. For example, Netflix runs all its services in VMs, not containers. You can’t achieve the “-ilities” by packaging your services into microservices by just using the DevSecOps pipeline and infrastructure as code, which defines the automation for your infrastructure in a domain-specific language (DSL). Again, cloud native is not just about automation, services, and container in an infrastructure.

Cloud native is about the combination of all the mentioned technologies with well-designed infrastructure to solve technical and business problems. Cloud native applications do not directly benefit from IaaS; they run in a cloud environment with mostly autonomous systems.

As shown in Figure [16-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig1), the cloud native infrastructure creates a platform on top of IaaS that provides autonomous application lifecycle management. The platform is created on top of dynamic infrastructure to abstract away from individual servers, storage, etc., and it promotes dynamic resource allocation and configuration.



***Figure 16-1***

Cloud native infrastructure

I will explore how cloud native infrastructure is different by looking at the processes to deploy, manage, test, and operate infrastructure in subsequent sections.

**Cloud Native Environment Characteristics**

As I mentioned, simply having a virtualized environment does not equate to being fully cloud native. According to the National Institute of Standards Technology (NIST), a cloud native environment should have all the following characteristics. You should embrace all these characteristics to be truly cloud native.

* On-demand service
* Broad network access
* Elasticity
* Virtualized environment
* Pay-per-use model
* Policy as code
* Resource pooling

**Cloud Virtualization**

The main enabling technology for cloud computing is virtualization. Virtualization separates a physical computing device into one or more virtual devices, each of which can be used to perform separate computing tasks.

Virtualization is a technique that allows the sharing single physical instances through multiple copies of instances. It is the creation of virtual servers, desktops, storage, networks, etc. Cloud virtualization mainly deals with server virtualization.

Cloud infrastructure can contain a variety of bare-metal, virtualization, or container software that can be used to scale and share resources across a network to create a cloud. At the base, cloud computing runs on a stable operating system like Linux or Windows.

Virtualization software called a *hypervisor* is required to run virtual machines on a system. The hypervisor controls all the hardware resources and can take resources from one VM to another VM depending on the needs. The hypervisor always manages the states of all the VMs.

The cloud providers add management and automation layers for administrative control over infrastructure, platform, applications, and data, and they reduce human interaction for a repeatable process. Virtualization in a cloud provides agility and reduces the cost by increasing infrastructure utilization.

The cloud provides the added benefits of autoscaling, self-service access, and dynamic resource pooling, which is distinguished from normal virtualization.

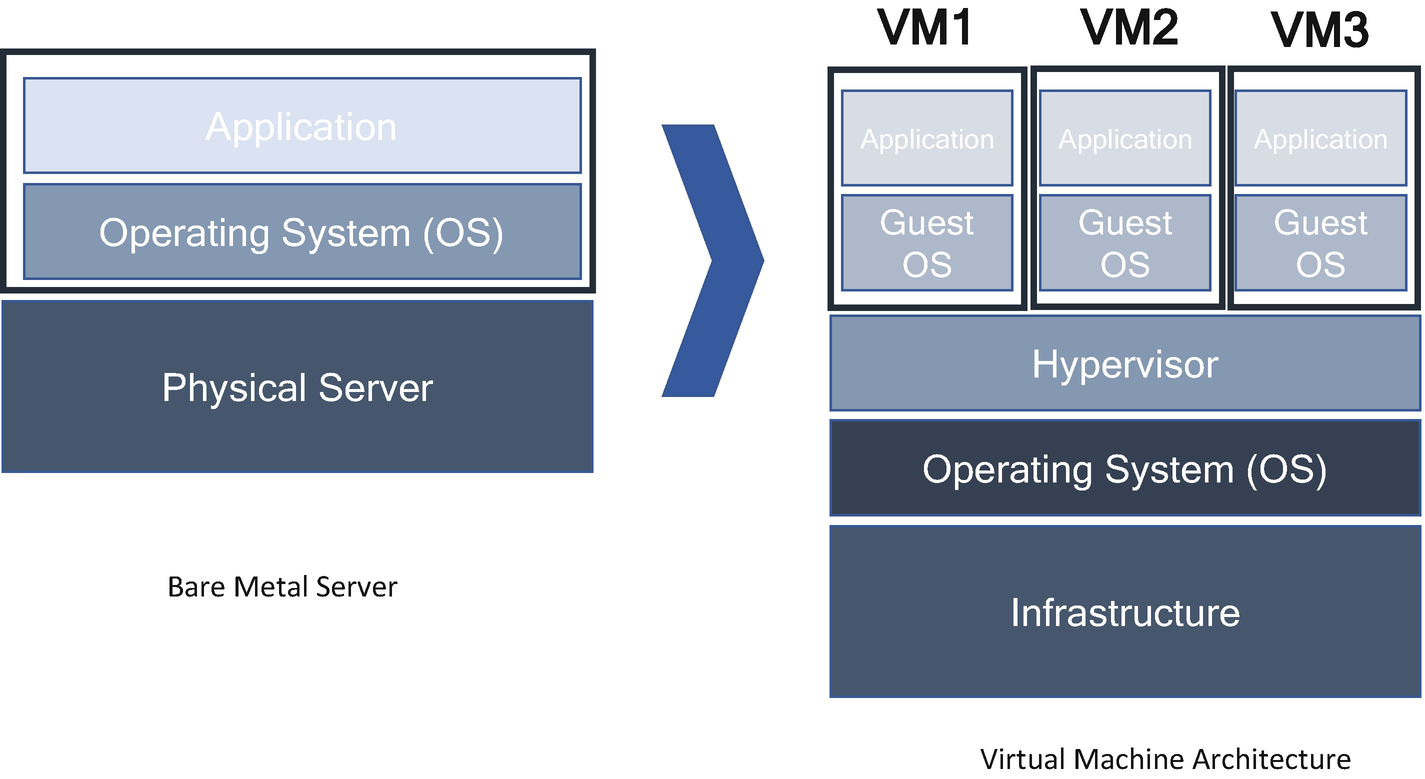
**How Does Virtualization Work?**

Virtualization plays an important role in the cloud. The virtual machines are required to share the infrastructure across users. There are two types of hypervisors, and these hypervisors run the virtual machines as guests.

* Hypervisors run directly on the system hardware. This is a bare-metal, embedded hypervisor.
* Hypervisors run on a host operating system that provides virtualization services, such as I/O support and memory management.

Each vendor provides its own hypervisors like VMware ESX and ESXi, Microsoft Hyper-V, Citrix Xen Server, Oracle VM Virtual Box, Red Hat Enterprise Virtualization, KVM, etc.

As shown in Figure [16-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig2), a hypervisor is the software that creates VMs and then manages the allocation of resources to them. VMs are infrastructure resources set up to use the resources of the host hardware. You can divide these resources to accommodate the necessary virtual machines as guests such as a server with 100GB RAM available and a Linux OS. If you want to virtualize hardware to run your application, you can create VMs and use a hypervisor to manage the resources of the server, like one VM is allocated 25 GB of RAM, etc. A hypervisor virtualizes the server and manages all of them in one physical server, so each VMs operates efficiently.



***Figure 16-2***

Cloud virtualization

**Types of Virtualization in the Cloud**

The cloud computing model depends on virtualization. By virtualizing a server, storage, network, and other physical data center resource, cloud providers can offer a range of services to users including IaaS, PaaS, and SaaS.

Virtualization is widely applied to several concepts including the following:

* *Server virtualization*: With server virtualization, one physical machine is divided into many virtual servers.
* *Desktop virtualization*: This creates multiple desktop operating systems, each in its VM on the same computer. One VM can be Windows, and the other can be Linux.
* *Network virtualization*: With this physical resource of a network, create different virtual networks that work independently of each other.
* *Storage virtualization*: This enables all storage devices on the network like server storage and stand-alone storage. It clusters all block storage into a single shared pool from which they can be assigned to any VMs on the network.

There are many virtualization techniques are available for application, data, data center, CPU, and GPU. Virtualization provides numerous benefits other than just resource isolation. This makes up the technology for IaaS. The following are some of the benefits of virtualization:

* It treats disks of VMs as files that can be snapshotted for quick backup and restore.
* It can be easily migrated and relocated if the machine requires any maintenance.
* It’s easy to expand resources such as CPU, memory, etc.

**What Applications and Services Are Commonly Virtualized?**

Virtualization is a foundational component in the cloud and serves as the underlying infrastructure for cloud native applications. According to IDC, more than 80 percent of workloads are virtualized today. With virtualization, you can improve efficiency, free up resources, and enhance security. Cloud vendors provide agile, fast, and cost-effective virtualization solutions.

Cloud vendors provide various types of customizable VMs that let you create and run virtual machines on a cloud infrastructure. Organization will want to run VMs in the cloud such as Google’s Compute Engine, AWS’s EC2, and Azure’s Virtual Machine. They offer multiple machine families to choose from, each suited for specific use cases. Table [16-1](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Tab1) describes the major virtual machine solutions from the major cloud providers, as of this writing.

***Table 16-1***

Virtual Machine Comparison Across Cloud Providers

| VIRTUAL MACHINES | GCP | AWS | AZURE |
| --- | --- | --- | --- |
| **GENERAL PURPOSE** | This configuration has a lower price and lower performance and is suitable for most workloads including database, nonproduction environment, web application, etc. | This configuration is general-purpose and provides a balance of computing, memory, and networking resources that can be used for general common workloads. | This configuration is a general-purpose ideal for the nonproduction environments and low and medium levels of traffic. |
| **COMPUTER OPTIMIZED** | This configuration is for most compute-intensive workloads and suitable for game servers, IoT use cases, etc. | This configuration is for more compute-bound applications with high-performance processors and is suitable for batch processing, gaming servers, scientific modeling, etc. | This configuration is designed for a high CPU-to-memory ratio and is suitable for batch processes, network appliances, etc. |
| **MEMORY-OPTIMIZED** | This configuration is required for memory-intensive operations such as real-time analytics etc. | This configuration is designed for fast performance for workloads that process large data sets in memory and is suitable for IoT, high-performance DBs, etc. | This configuration is designed for high memory to CPU ratio and is suitable for databases, distributed caching, and in-memory analytics. |
| **ACCELERATOR OPTIMIZED** | This is for complex configurations like 16 GPUs in a single VM and is suitable for machine learning training and interfaces. | NA |  |
| **STORAGE OPTIMIZE** | NA | This configuration is designed for workloads that require high, sequential read and write access to very large data sets on local storage. | This configuration is designed for high disk throughput and IoT and is ideal for databases, etc. |
| **GPU** | NA | NA | This family of VMs is specialized and suitable for graphic rendering and video editing, ML processing, etc. |
| **HIGH-PERFORMANCE COMPUTE** | NA | NA | This family is the fastest and most powerful CPU with optional high throughput network interfaces and suitable for weather modeling, reservoir simulation, digital twin, etc. |

**Cloud Native and Virtual Machines**

Developing and delivering systems keeps your organization more competitive. To do so, many organizations have adopted cloud native services with containers and Kubernetes. In IT, you cannot develop an isolated system; you require access to legacy technologies for any existing transaction. This is reality. Where are these applications run? How do you handle these applications that require VMs without complicating the management of virtualization and containers?

The VMs cannot be easily containerized with cloud native architecture. Some tools like KubeVirt and cloud native VMs (CNVM) reimagine VMs in Kubernetes. You can use your existing Kubernetes tools to natively manage VMs or convert those workloads into a container. This gives a flexible environment for a cloud native application. The cloud native VM is a VM inside a container.

For example, the Red Hat OpenShift virtualization solution supports containerized applications faster by hosting VM-based systems on the same platform as container-based applications. This supports the division of the existing system as well as the continued use of existing virtualized applications by managing virtualized systems and containerized services as part of single application deployment. OpenShift virtualization is enabled for a Red Hat OpenShift cluster; you can create and add virtualized applications to your project in the same way as containerized applications. This enables VMs to run in parallel on the same Red Hat OpenShift nodes as a traditional system container.

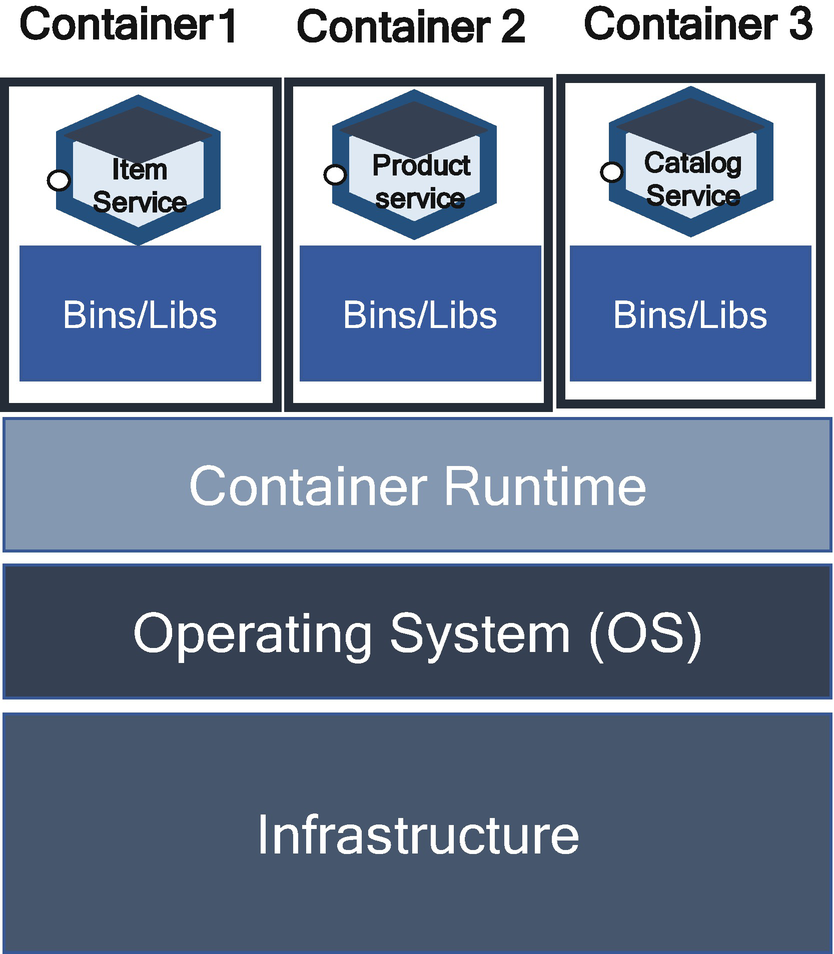
**Containerization**

Cloud native applications are distributed in nature and utilize cloud infrastructure. Numerous techniques and tools are used to implement cloud native applications, but from a computing perspective, cloud native application uses mainly containers. Containerization became a de facto standard for cloud native systems as an alternative to VMs.

As shown in Figure [16-3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig3), the container is a technology that allows you to incorporate and configure your binaries and their dependencies in a package called an *image*. This image can be used to spawn an instance of your services: a container.

The services image in a container is abstracted from the environment in which services are executed. This abstraction allows cloud native-based services to be deployed easily and consistently across all environments, regardless of private, public, or hybrid environments. Container architecture provides a clean separation, as engineers can focus on a service’s business logic and dependencies.

If you compare containers with VMs, as mentioned, the guest OS runs on top of a host OS with virtualized access to the underlying infrastructure. Like the VM, the container allows you to package your services together with binaries and other dependencies, providing an isolated environment for running your cluster of services. Containers provide a more lightweight architecture to work with and with more benefits.



***Figure 16-3***

Container architecture

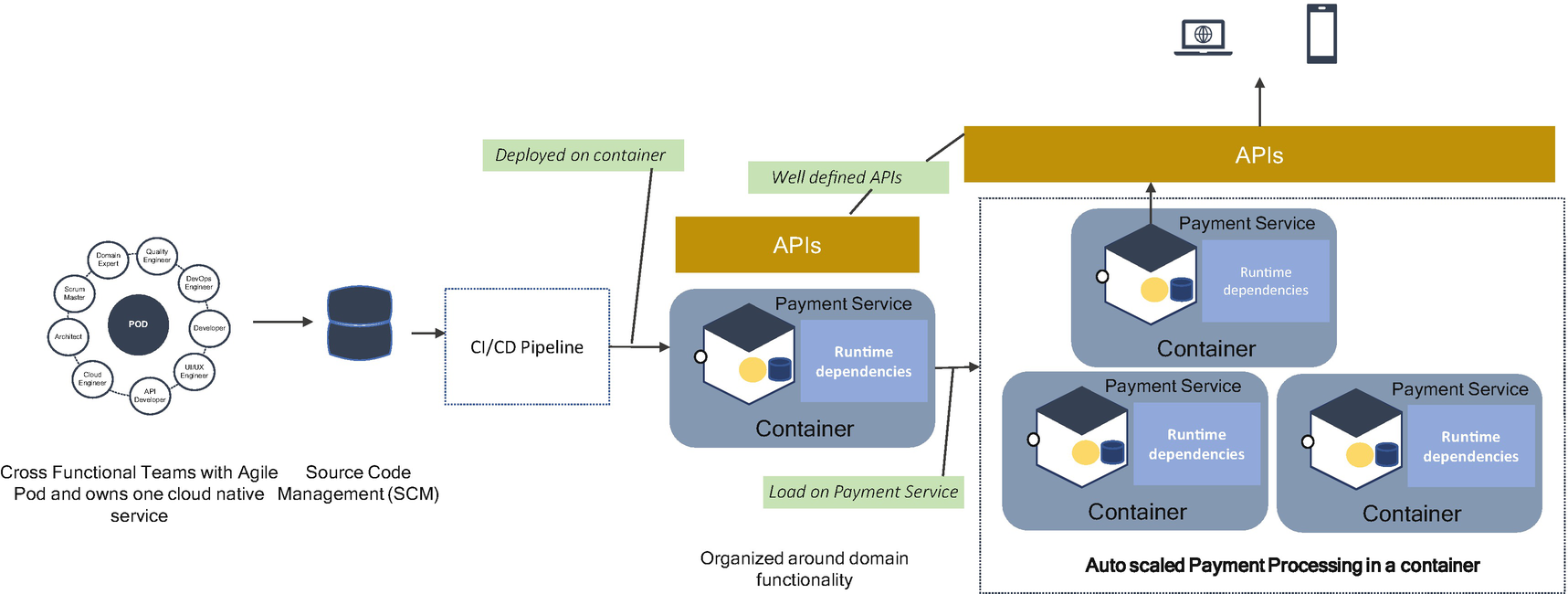
The containers are virtualized at the OS level, with multiple containers running atop the OS kernel directly unlike VMs virtualization at the hardware level. This makes a container more lightweight and allows it to share the OS kernel, start much faster, and use a fraction of the memory compared to booting the entire OS.

Linux Containers (LXC) was created by engineers from IBM around 2008 and is layered with some tooling on top of cgroups and namespaces. LXC works on a single Linux kernel without requiring any patches. LXC 1.0 was released around early 2014 and leveraged longstanding security technologies. In 2013, Docker emerged, and the container exploded in popularity and usage. Initially, Docker was built on top of LXC containers.

After the importance of cloud native services grew, the industry saw containerization become a foundation for modern software infrastructure. Research firm Gartner predicts that by 2022 more than 75 percent of global organizations will be running containerized applications in production.

**What Is a Container Image?**

A container image provides packaging and isolation of your services, as shown in Figure [16-4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig4). The following are the few characteristics of a container image.



***Figure 16-4***

Container image characteristics

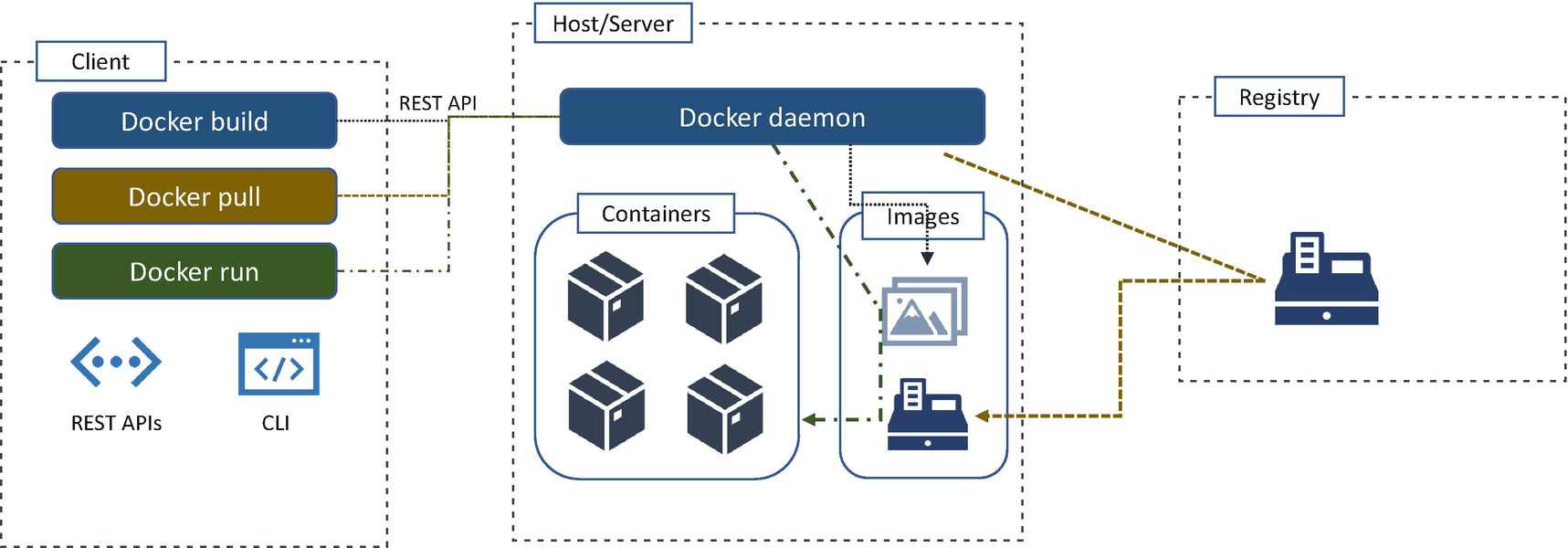
* A container image is immutable, and once it is built, it does not change; it is configured.
* A container image is a unit of domain functionality that addresses a single concern.
* A container image is owned by one agile pod team and has its release cycle.
* A container image is self-contained and defines and carries its runtime dependencies.
* A container image has well-defined structure APIs.
* A container image is disposable and safe to scale in and out.
* A container image is self-healing capability.
* A container image is stateless and is modular.

A container image provides a single unit of functionality, belongs to a single team, has an independent release cycle, and provides deployment and runtime isolation. Most of the time, one cloud native service corresponds to one image.

**Container Architecture**

Docker is open source and the most popular container technology; it’s a containerization engine that works with most of the popular products.

Docker uses the client-server architecture paradigm, as depicted in Figure [16-5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig5). The client talks to the server, which does most of the work like the application client-server architecture. The client is the Docker client, and the server is the Docker daemon. The daemon builds, runs, and distributes Docker containers.



***Figure 16-5***

Container architecture

The Docker client and host/server communicate using REST APIs over a socket or a network interface. The Docker daemon provides a list of services through REST APIs. It listens to APIs and manages objects such as images, containers and networks, and volumes. One daemon can also communicate with another daemon to manage cloud native services.

The client is like a user interface. It is the primary interaction point for external users for configuration, using commands. For example, the client sends commands to the daemon over the APIs of multiple daemons in a cluster. The client can reside on the same host as the daemon or on a remote host.

A registry stores an image. For this, you can configure your private registry or Docker-provided registry called a *hub*. With the docker pull or docker run command, the required images are pulled from your configured registry. The docker push command pushes the images into your registry.

The image is a read-only template with instructions for creating Docker containers. For example, an image is your service with a Tomcat server and additional configurations.

The container is a runnable instance of your image. You can create, start, stop, move, or delete by using the REST APIs or command-line interface (CLI). The containers are isolated each other in the host server.

**Container Principles**

Today the container ecosystem has matured and has diverse and rich tooling that solves new and large-scale problems such as container orchestration, scalability, failure, high availability, cloud native service lifecycle management, and observability. It is not easy to achieve a production-ready large-scale deployment with thousands of cloud native services. The following principles and best practices help you to manage the container cloud native infrastructure effectively. Many of the practices are inspired by the 12-factor methodology, which is a standard way to develop a cloud native service.

The containerized application requires some principles to execute in a runtime container environment. With these principles, you will ensure that the container architecture is well designed to run services. The following are the principles; you can find details in Chapter [3](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_3_Chapter.xhtml):

* **Single-Container Principle (SCP)**
* In a cloud native architecture, SCP is about having a higher level of abstraction than responsibility. The single concern enables every microservice and container to address a single concern. SCP means every container must address a single concern with the cloud native service architecture style.
* **High-Observability Principle (HOP)**
* Observability is a measure of how well internal states of microservices can be derived from external outputs.
* **Lifecycle Conformance Principle (LCP)**
* LCP means that a container should have a way to read the events coming from the platform and conform by reacting to those events. All kinds of events are available for managing platforms that are intended to help you to manage the lifecycle of the container and cloud native services, based on all types of available events; it is up to you to decide which events to handle and whether to react to those events.
* **Image Immutability Principle (IIP)**
* IIP means an image is unchangeable once it is built and requires creating a new image if changes need to be made. You need to store the configuration and variables external to the container. For each image change, you need to build a new image and reuse it across various environments in your development lifecycle.
* **Process Disposability Principle (PDP)**
* PDP is a container runtime principle and states applications must be ephemeral as possible and ready to be replaced with container instances at any point of time by using infrastructure as code.
* **Self-Containment Principle (SCP)**
* SCP addresses the build-time concern, and the objective of this principle is that the container must contain everything that it needs at build time. The container relies on the presence of the Linux kernel or Windows silos and any additional libraries.
* **Runtime Confinement Principle (RCP)**
* RCP states that every container should declare its resource requirements and pass that information to the hosted platform.

**Container Patterns**

The following are a few best practices for making a container easier to design and operate. These practices cover a wide range of topics including security, monitoring, etc. These best practices are not always applicable in all business scenarios; choose the best one depending on the problem domain and business cases.

**Container Security**

Security must be built along with the DevSecOps pipeline throughout its lifecycle and there are a variety of tools available in the industry for managing container security. These tools can be used to scan a container, access a container cluster or vulnerabilities in images, and more.

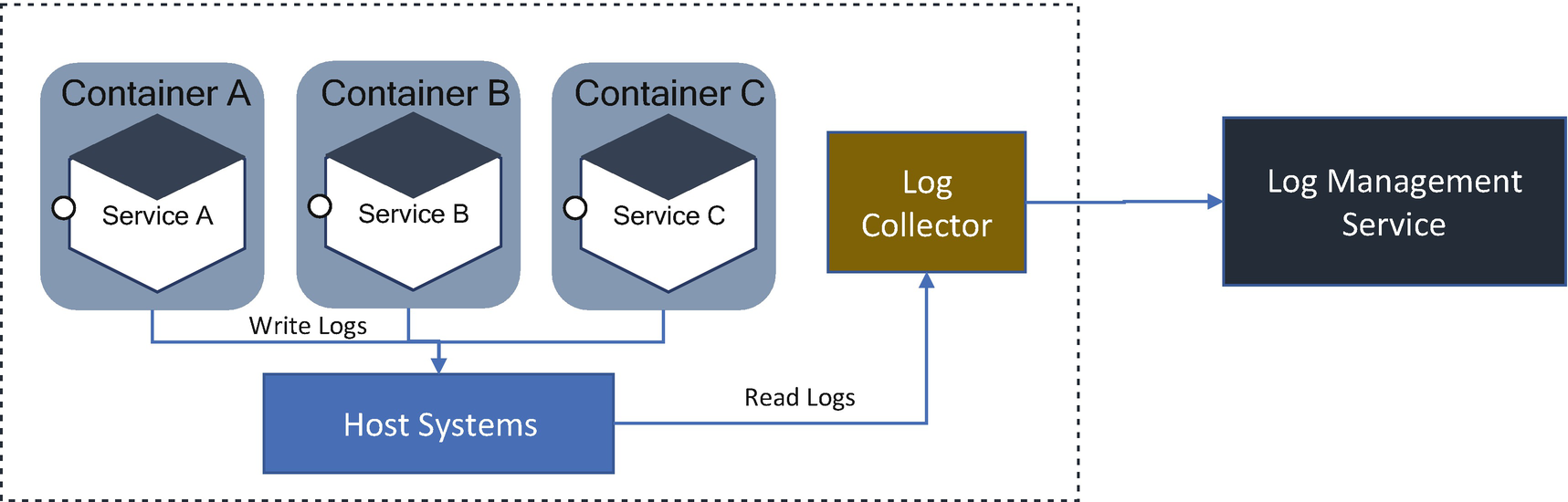
The containers are well placed with process isolation, meaning user namespaces and resource encapsulation with cgroups reduce the attack vector to provide a protection. The container image needs to be well-constructed with security guidelines because these are the components that are eventually running your application. If there are security vulnerabilities packed into the container image as it’s built, you increase the risk and potential severity of issues that will happen in production. The container security is not one concern; it spans pod teams, and there is an array of security layers that apply to containers:

* The container image and the libraries inside
* The interaction between containers and the host OS, both inbound and outbound
* Networking and storage
* Security at runtime, Kubernetes cluster

Alcide, Clair, WhiteSource, and Portshift are a few tools that help you to manage security in a container.

**Logging Mechanism**

Logs are an integral part of a system lifecycle and contain precious information about the events. Containers offer an easy and standardize a way to handle logs by using stdout and stderr. As shown in Figure [16-6](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig6), a container captures these logs and accesses them using Docker logs.



***Figure 16-6***

Log management in a container

You can use a log collector like Fluentd, Fluent Bit, etc., to collect data and send it to log management services like ELK in your application landscape.

**Stateless**

When you are designing your containers, don’t treat them as a normal traditional server. For example, you might follow old practices like updating services on the running container. Don’t do this. The containers are not designed to work this way. Always retire existing containers and create a new one. Follow either rolling deployment or blue-green deployment. Containers are designed to be stateless and immutable. Always store the state outside of your containers like in databases or any other storage event for user session also.

The stateful sessions are not best for cloud native services. When the consumer references a state on the server, the consumer opens a lot of incomplete sessions, and transactions happen. In the stateful system, the state is calculated by the client. This leaves the connection open and is difficult to verify the connections.

The stateless request issues a recent message in any ecommerce applications or social media. The response is independent of any server state.

* *Rolling deployment*: This deployment strategy slowly replaces previous versions of service with new versions of the service by completely replacing the container on which your service is running. For example, if the pod team updates an item service, then the container running the previous item service will be replaced by the new version of the service.
* *Blue/breen deployment*: This uses two identical containers in a separate environment, while the production environment uses one active environment. You can update the other environment without interrupting the active environment. Then, when another is environment ready, you route your request to another environment.

Rolling deployment is faster than blue/green deployment. Unlike blue/green, the rolling deployment does not have any isolation environment. This allows a rolling deployment more quickly, increases risk, and complicates the process of a rollback if the deployment fails. For successful rolling deployment, you need to have well-defined automation with continuous deployment and infrastructure as code.

**Immutable**

Immutable means containers won’t be modified during their lifetime, with no updates, no patches, or even no configuration changes. If you want any modifications, kill the existing container, and create a new one. Immutability makes deployment safer and more repeatable. If you want to remove the existing image, just roll back and redeploy with another image. This helps you to maintain uniformity across environments. To use the same container images across the environment, like from development to production environment, suggest externalizing the configuration.

**Privileged Containers**

If your application uses the root user, then avoid it. If your services are compromised, an attacker would have full access to them. Therefore, avoid using privileged containers. A privileged container has access to all the devices of your environment, bypassing almost all the security features. Suggest giving specific capabilities to the container through security context options like the –cap-add flag of Docker. If you need to modify the host settings, then those details make it separate by using the init or sidecar pattern, as explained in Chapter [4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_4_Chapter.xhtml).

**Monitoring**

Like logging, monitoring is an integral part of your system. In many ways, monitoring containerized applications follow the same process as logging. However, the containers are short-lived; you cannot add monitoring configuration within the containers. While designing and monitoring, separate black box and white box monitoring. Black box monitoring examines your application from outside. It is able to provide details about the audience because it is outside of the infrastructure. White box monitoring examining your services with privileged access and gathers metrics on its behavior that the users cannot view. You can configure various tools like Prometheus to capture these details.

**Running Container as Root**

Containers provide isolation. With container default settings, a process inside a container cannot access information from the host machine or other peer containers. Because containers share the kernel of the host machine, the isolation isn’t complete as it is with VMs. An attacker could find unknown vulnerabilities that would allow the attacker to escape from a container. To avoid this, do not run processes as root inside containers.

**Image Version**

Always tag a version of the image you are using. Suggest using the “recent” tag, which can be moved from image to image.

**Container Networking**

The portability and short lifecycle of containers eliminate manual configuration and externalization configuration and leverage the network automation capabilities of your container orchestration.

**Container Lifecycle Management**

Containers are short-lived, and their lifecycle must be managed carefully by using automation.

**Container Benefits**

The containerization of an application brings many benefits including the following:

* *Agility and productivity*: It brings well-streamlined and accelerated development and improved consistency across the environment with the right best practices.
* *Fine-grained resilience*: It offers isolated deployments of the highly available components with no single point of failure.
* *Portability*: Containers can be deployed on any cloud provider and in their own data centers.
* *Security*: It offers improved security by isolating applications from the host system and each other.
* *Scalability*: Dynamic scaling and orchestration with the use of Kubernetes provides more robust for transaction spikes.
* *Edge level of networks*: Containers are beneficial at the edge level of networks. At the edge levels of networks, low latency, resiliency, and portability requirements are significant.
* *Machine learning models*: Containers benefit ML models where a problem can be separated into a small set of tasks.
* *Cost*: A container doesn’t require a full guest OS and hypervisor and the container has only faster boot times, smaller memory footprints, and generally better performance. This helps trim cost.

**Container Adoption Best Practices**

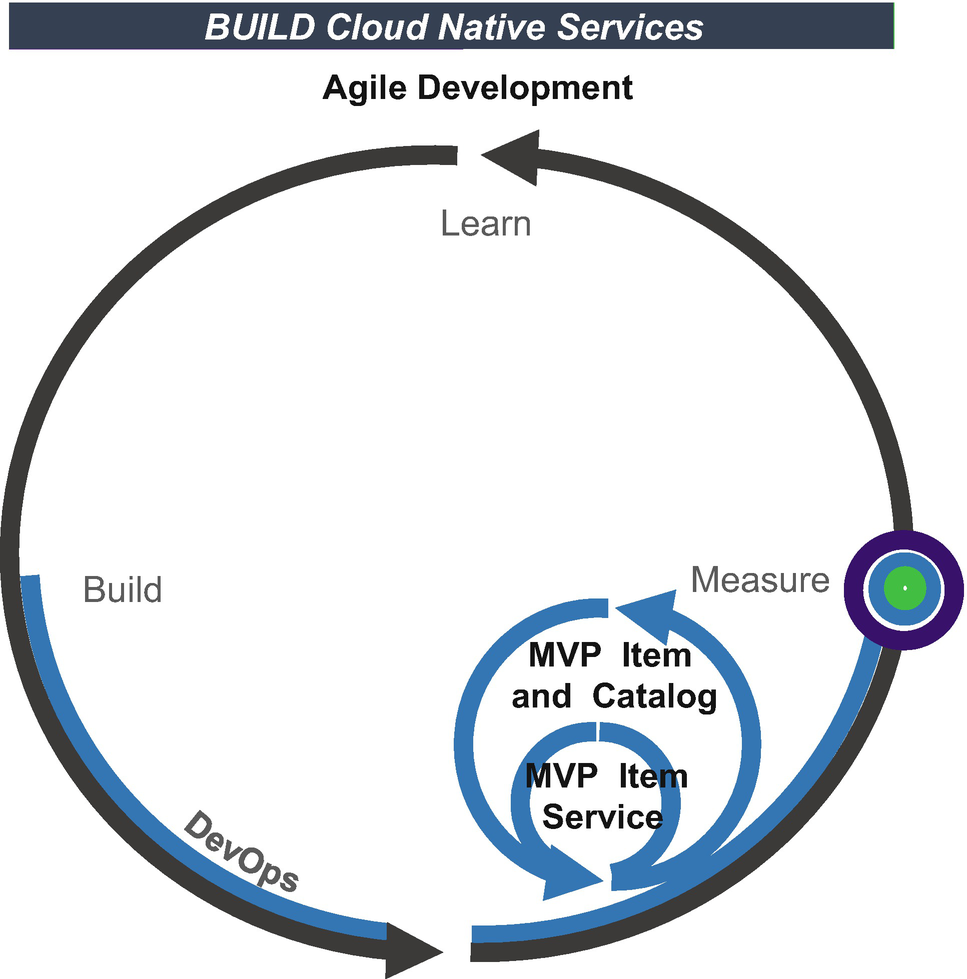
The following are the best practices that are required to adopt for containerization:

* *Use fine-grained components*: The smaller the unit, the easier it is to manage and orchestrate. Break your components into fine-grained single responsibility units.
* *Use disposable components*: Always design and build stateless and lightweight containers. This enables the easy to manage, easy to create, and easy to destroy.
* *Implement container security*: Implement security measures and policies across the entire container environment including container images, hosts, config files, registries, etc.
* *Implement container with orchestration*: For efficient management of a large number of containers, adopt platform orchestration.
* *Automate the pipeline*: Implement automation in every lifecycle of software engineering including the DevSecOps pipeline and infrastructure as code.
* *Agility*: Implement agility to help pod teams improve the development lifecycle and move faster to market.

**Containers in an Enterprise**

The following are the key considerations when you are deploying containers on a large scale in an enterprise:

* **Technology Disruption**
* The container deployment is complex; you need to properly build, configure, deploy, manage, monitor, and update in a production environment. To achieve this, you need a series of tools and culture. However, most tools are third party and are constantly evolving. When tools evolve, you and your team need to keep pace with the disruption of technology, and you are required to continuously upgrade because of disruptions.
* **Culture**
* To use containers in an enterprise, you require strong backing by the leadership team to embrace a culture of cloud native.
* **Most Value Product (MVP** **)**
* It is not viable to jump into creating thousands of services across enterprises. This creates more problems than solves. Start small by running a few services in containers and learn and create a template and scale across enterprises. Figure [16-7](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig7) provides the details of an MVP approach. In this example, the engineering team creates an item microservice as an MVP and deploys it in container. Then it measures and learns from the experiences and mistakes and applies learnings to create another microservice called a *catalog*. Once you have gained experience, then go with @Scale IT for other services in an ecommerce application.



***Figure 16-7***

MVP development

* **Deployment Environment**
* Many options are available for your services deployments, from local on your own data center to public and private cloud and hybrid cloud. The cloud vendors offer a wide range of Docker environments.
* IaaS option, running containers on AWS EC2, Azure VM, and Google Compute Engine
* Fully managed container service (CaaS) solutions designed for hosting containerized services, such as AWS Elastic Container Services (ECS), and Azure Container Services
* Containers with Kubernetes
* Registry
* The containers are built based on images. If there is a vulnerability in the images, the containers inherit the issues and carry them to the production environment. You need to make sure the images are safe to use across the environment. For this, you need to enforce container scanning and a private registry. You adopt the following best practices for container scanning:
* Choose the right version from the artifactory.
* Create an optimized image file.
* Scan an image as part of the DevSecOps pipeline by using tools like Clair.
* Scan an image again in production.
* Ensure your scan images at multiple stages during the development lifecycle.
* **Monitoring and Logging**
* To securely manage the Docker environment across enterprises, you need to gain visibility into every deployed container in an environment. You can achieve this by using well-established integrated monitoring across the environment and automating responses and fault conditions.

**Container Orchestration**

Container orchestration automates the deployment, management, scaling, and networking of containers. If your enterprises want to implement hundreds or thousands of services in a container, then you can’t think about collaborating containers without any orchestration. The container orchestration can be used in any environment like public cloud, private cloud, or hybrid cloud. It can help you to deploy the same services across different environments and automates scheduling, scalability, load balancing, availability, and networking of containers.

Generally, container orchestration tools communicate with a human-created YAML or JSON file that describes the configuration of the application. This configuration file contains rules and directs orchestration on how to retrieve container images, how to create a network between containers, where to store log data, and how to mount storage volumes.

It manages the deployment scheduling of containers into clusters and automatically identifies the right host. The selection of the right host depends on user-defined guidelines, labels, or metadata. Once the host is assigned, the orchestration tool automates and manages the services throughout the lifecycle based on the rules defined.

The orchestration tools automate and manage the cloud native services in containers including the following:

* Configuration and scheduling containers
* Provisioning and container deployment
* Resource allocation
* Scaling containers to balance requests
* Service discovery
* Monitoring container health
* Interaction between containers security
* Traffic routing

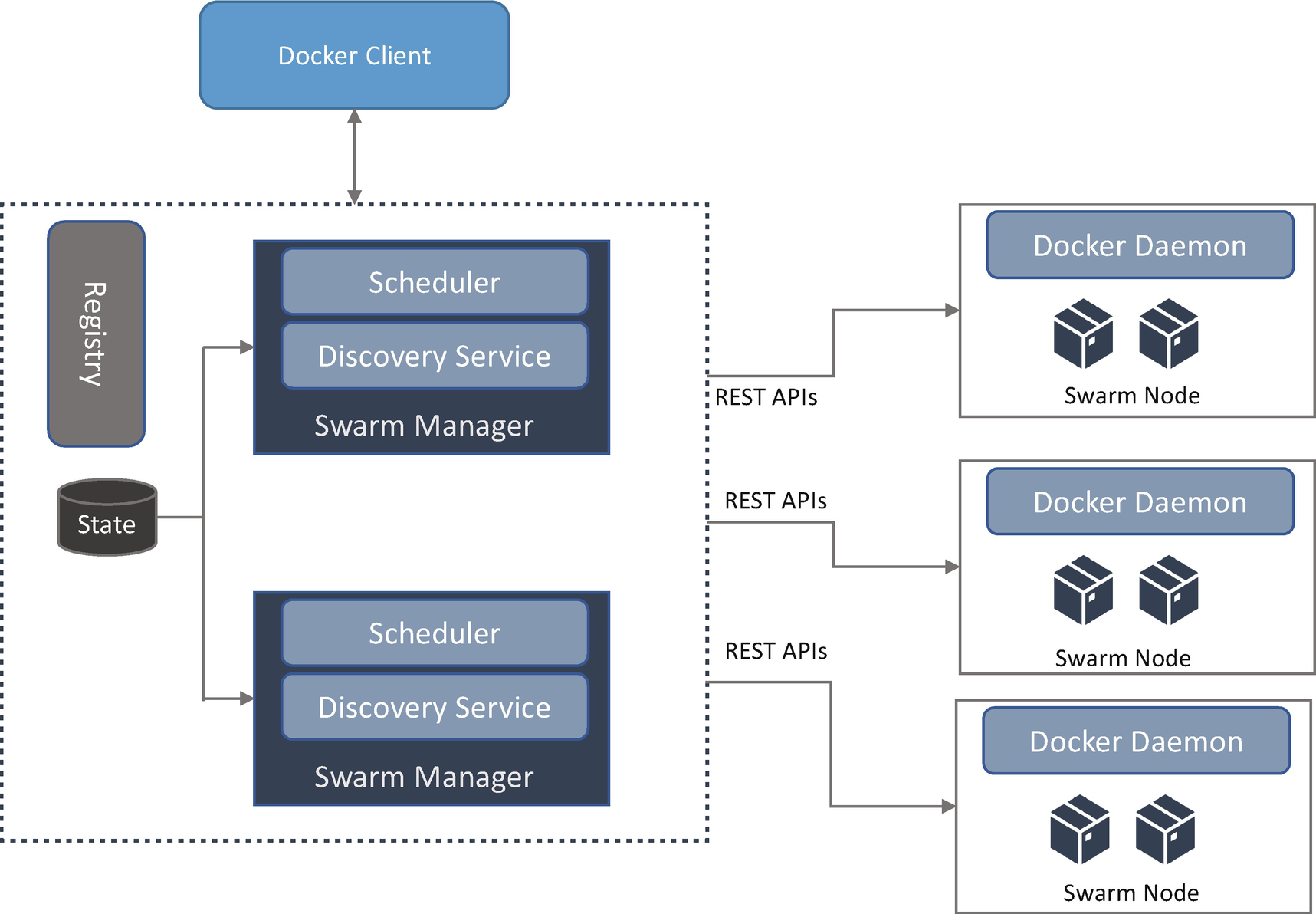
The container orchestration tools provide a framework for managing containers in an enterprise. There are many container orchestration tools available in the market that can be used for managing the container lifecycle. Some popular options are Kubernetes, Docker Swarm, Apache Mesos, etc. Some cloud provides offering PaaS services on top of Kubernetes such as OpenShift, Google Kubernetes Engine, AWS Elastic Kubernetes Service, Azure Kubernetes Service, etc.

**Types of Orchestration Tools**

Choosing the right orchestration tool for your enterprises involves diverse factors such as the number of containers in an environment, technical experience and skill level of your resources, maturity of tools, widely used references, etc. The following are the most popular ones.

**Docker Swarm**

Docker Swarm is a container orchestration tool that is built into Docker engines. As shown in Figure [16-8](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig8), a Swarm is based on the client-server architecture style. The client is the Swarm manager, and the host servers are containers hosted in multiple nodes.



***Figure 16-8***

Docker Swarm architecture

Docker Swarm consists of a node, manager, and services. The manager node uses the Raft consensus algorithm to internally manage the cluster state. This is to ensure all manager nodes that scheduling and controlling the containers in the cluster maintain state. Docker Swarm can have more than one manager node lead by a single manager node elected using a Raft algorithm, and a manager node can act as both a Swarm node and a manager node. The manager nodes act as orchestration and cluster management functions required to maintain the desired state of the swarm. The Manager node elects a single leader to conduct an orchestration task.

The Raft algorithm achieves a consensus via an elected leader. A server is a Raft cluster that is either a leader or a follower and can be a candidate in the precise cause of an election. Consensus involves multiple servers agreeing on values. Once they decide on a value, that is final.

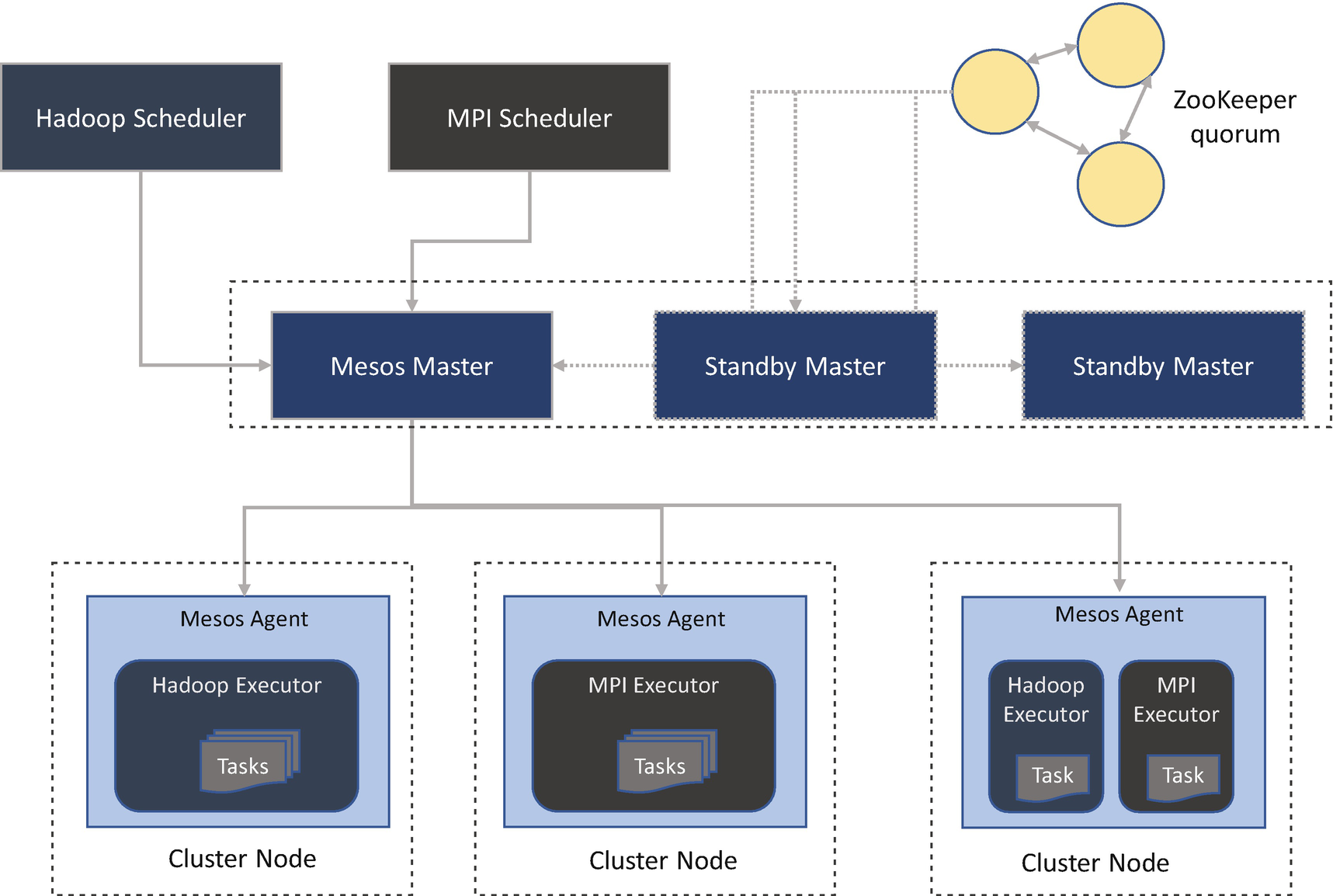
The Swarm node is a cluster of Docker containers running on a cloud server. These nodes receive and execute tasks dispatched from manager nodes. Each swarm of worker node reports back to the manager on tasks. The swarm node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

**Apache Mesos**

Mesos is a cluster management tool that handles the workload of both containers and noncontainers in a distributed environment through dynamic resource sharing and isolation. Mesos works differently than Kubernetes and Docker Swarm, which are both container management tools with the node-to-node relationship. Mesos is more of a resource allocation manager that allows you to manage jobs and provides a framework that allows you to launch both containers and noncontainers on the same cluster. This means you can run any distributed application like Spark, Hadoop, etc., that requires clustered resources.

Mesos works between the application layer and OS and makes it easier to deploy and manage large-scale clustered environments more efficiently. It works exactly the opposite of virtualization. In virtualization, one physical resource is divided into multiple virtual resources, while Mesos unites multiple distributed resources into one.

As shown in Figure [16-9](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig9), the Mesos master is the main component of a tool. It makes sure the framework is highly available and provides user interfaces that provide information about the resources available in a cluster. All the tasks in a master are stored in memory.



***Figure 16-9***

Apache Mesos architecture

The Mesos agent manages the containers that host the services. It manages the communication between the Mesos master and an executor.

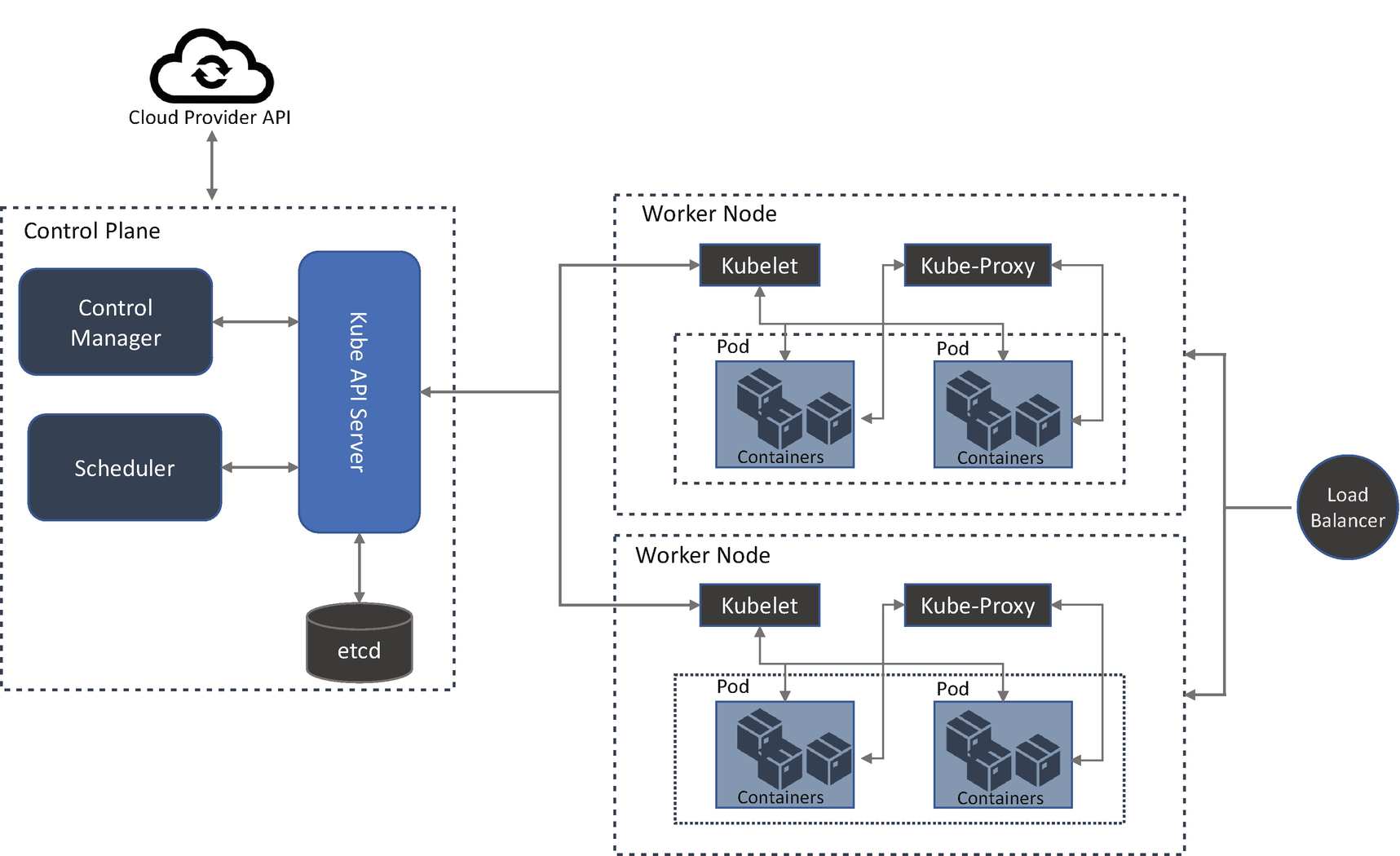
Mesos consists of a master daemon that manages agent daemons on each cluster node, and Mesos runs tasks on these agents. The master enables fine-grained sharing resources across frameworks by making them resource offers. The Mesos master decides how many resources to offer to each framework. The framework consists of the scheduler and an executor. The scheduler registers with the Mesos master for resources, and the executor process launches an agent node to run the framework tasks. The Mesos master determines how many resources are offered to each framework, and the framework scheduler selects the resources to use.

In subsequent sections, I will consider only the Kubernetes framework to explain further cloud native use cases.

**Kubernetes**

Kubernetes is a container orchestration platform. The origin of the platform is from Google data centers, where Google’s internal orchestration platform is named Borg. Google used Borg for many years to run in its data centers before it transfers into a new open source project called Kubernetes in 2014, and later it became part of Cloud Native Computing Foundation in 2015. Currently Kubernetes is one of the most active projects in GitHub.

As shown in Figure [16-10](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig10), Kubernetes is designed with a client-server architecture style. It consists of a control plane (master node) and several worker nodes. The control plane consists of an API server, control manager, scheduler, and etcd storage. Initially the setup will contain one control plane, but you can have a multiple control planes for high availability.



***Figure 16-10***

Kubernetes architecture

A control plane maintains the details of all the Kubernetes objects and continuously manages object states, responding to changes in the worker nodes. The API server acts as a bridge between the worker node and control plane, and engineers can access the control plane by using this API server. The control manager is a daemon that runs the control loop, watches the state of the nodes, and makes changes appropriately. This manager integrates with the cloud for availability zones, VMs, storage services, etc., and the scheduler schedules the containers across the worker nodes.

The worker nodes are the machine that runs containers and is managed by the control plane, and the Kubelet controls the execution of containers in a node.

**Orchestration Tool Comparison**

Table [16-2](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Tab2) provides a high-level comparison between major orchestration tools.

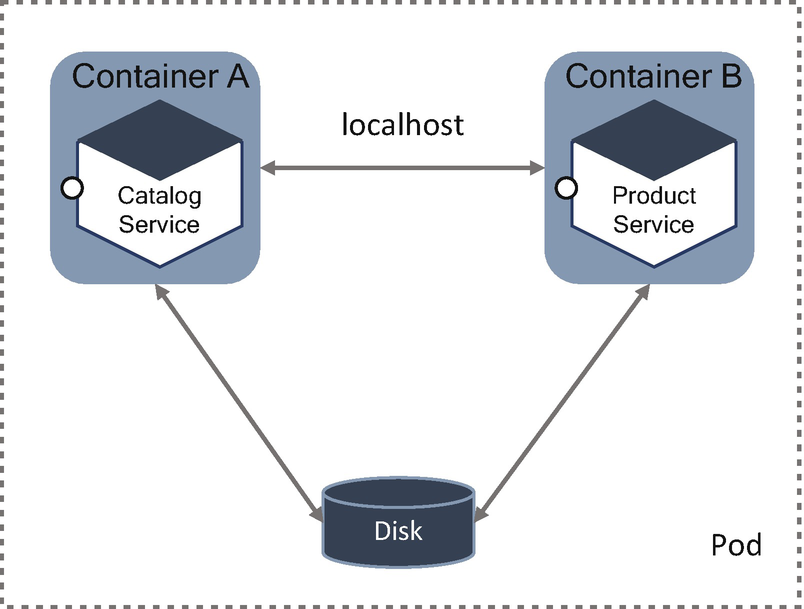
***Table 16-2***

Orchestration Tool Comparison

| Docker Swarm | Apache Mesos | Kubernetes |
| --- | --- | --- |
| **It is a native Docker clustering solution that makes it easy to integrate and set up flexible APIs.** | It is more of a resource allocation manager that allows you to manage jobs and provides a framework that allows you to launch both containers and noncontainers on the same cluster. | The control plane maintains the details of Kubernetes, and the API server acts as a bridge between the worker node and the control plane. |
| **YAML-based configuration deployment.** | Unique format-based configuration deployment. | YAML-based configuration deployment. |
| **Mature and has good stability.** | Mature. | Mature and has good stability with continuous update and large community. |
| **Defined using a Docker Compose file and same compose file to maintain a cluster of containers on a single machine.** | It is based on an N-ary tree with groups as branches and applications as leaves. | It is a combination of ReplicaSets, controllers, and pods. |
| **A Swarm is based on the client-server architecture style. The client is the Swarm manager, and the host servers are containers hosted in multiple nodes.** | Handles the workload of both containers and noncontainers. | Container orchestration platform and works with the client-server architecture. |
| **It supports only Docker.** | It supports both container and noncontainer workloads. It runs on Swarm and Kubernetes. | It supports both Docker and rkt. |
| **It includes a DNS server out of the box that allows for service discovery by name.** | Does not support any service discovery. | It includes optional DNS for discovery by name; here services are exposed through HTTP ingress or mapped to an external load balancer. |

**Kubernetes Features**

In a cloud native platform, you cannot just run on the siloed container; you need to have a group of containers for your application. Kubernetes provides a management layer for the lifecycle of a group of containers called a *pod*, as illustrated in Figure [16-11](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig11).



***Figure 16-11***

Pod deployment and management

A pod is an atomic unit of scheduling, deployment, and runtime isolation for a cluster of containers. All containers in a pod are always scheduled to the same host, deployed together whether for scaling or host migration and sharing a namespace, filesystem, and networking. Containers in a pod interact with each other over the filesystem or networking. The features of the pod are as follows:

* A pod is an atomic unit of scheduling, which means the scheduler identifies a host that satisfies all the containers in a pod.
* A pod ensures the colocation of containers, provides various means of communication patterns like networking, provides a filesystem, etc.
* A pod has an IP address, name, and port range that are shared by all containers within it. This means the containers in a pod are carefully configured to avoid port clashes.
* Pods are ephemeral. They are disposable. A pod can be rescheduled at a different node at any time if the existing node is unhealthy.

**Kubernetes Principles and Patterns**

Containers are the building blocks of Kubernetes-based cloud native applications, and containers play a fundamental role in Kubernetes. Creating modularized, reusable, single responsibility container images is fundamental to your cloud native architecture.

Containers and pods and their unique characteristic offer a new set of principles and patterns for designing a cloud native service. Adhering to these principles and patterns will help ensure your applications are suitable for automation in cloud native platforms. Here, I am covering a few, but you can find more patterns in a book called *Kubernetes Patterns*.

**Predictable Demands**

The successful application deployment, management, and coexistence on a multitenant environment is dependent on application resource requirement and runtime dependencies. This pattern is about how you declare requirements.

Kubernetes can manage polyglot containerized services, and each service has different resource requirements. For example, some services execute faster than other services, or some programming languages require different things from other languages. It is difficult to identify the number of resources required for a container. Some services require more memory and more CPU, and some may require less. Some may require polyglot storage, and some may be stored in memory, etc. Defining all these characteristics in a cloud native application is a must.

Knowing about these requirements is helpful because Kubernetes can make intelligent decisions to place a container on a cluster and helps you with proper capacity planning.

**Declarative Deployment**

This pattern is about managing the rollback and upgrading a newer version of the container in a pod. The cloud native maturity in your organization leads to more adoption of services. The number of services increases because you have to continually update and replace them with newer versions due to business changes. Upgrading these services leads us to starting a new version of a pod, stopping the old version of the pod, etc. Performing manually leads to human errors and takes time.

Kubernetes has automated these activities without involving any humans. Using this concept, you need to describe how your application should be updated using different strategies. Rolling deployments are the declarative way of updating services in Kubernetes through the concept of deployment. The rolling update behavior ensures there is no downtime during the update process.

Use a rolling deployment, fixed deployment, blue-green release, or canary release deployment strategy.

**Health Probe**

This pattern is about how a service can communicate its health state to Kubernetes. The cloud native service must adopt an observable-as-a-service approach, which helps Kubernetes to detect whether the service is healthy. These observations influence the lifecycle management of pods and the way traffic is routed to the application. Kubernetes regularly checks the container process status, but checking just the status does not provide the complete health of a service, for example, if your service hangs or is slow to respond. You can get the health of a service from this.

A *process health check* is the health check process by Kubelet that is done for all the services in a pod. This process identifies the service failure or service shutdown in a container.

A *liveness probe* is the health check process and regularly checks the Kubelet agent to confirm the container’s health. This helps to kill the unhealthy container and replace them with a new one.

A *readiness probe* performs the readiness of a container like liveness probes.

Your cloud native services must be highly observable by providing a means for the managing platform to identify the health of the service. Health checks play a fundamental role in cloud native services such as automating the deployment, self-healing, scaling, etc. Logging is one good practice for health probes. You need to design your containers such that they provide relevant APIs for health checks. These APIs are read-only endpoints the platform is continuously probing to get application insights. You can refer to the box and port style architecture in Chapter [5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_5_Chapter.xhtml). It is recommend that you use the health checks process to manage the pods.

**Automated Placement**

One of the core functions of the Kubernetes scheduler is to assign new pods to nodes, satisfying container resources requests and honoring scheduling policies. This pattern talks about Kubernetes’ scheduling algorithm.

Matured cloud native enterprises might have hundreds or even thousands of isolated processes. Containers and pods do provide a nice abstraction of packaging and deployment but do not support how to place these processes on suitable nodes. With growing cloud native services, assigning and placing them individually to nodes is not a manageable activity.

In Kubernetes, assigning pods to nodes is done by a scheduler, which is highly configurable. The main operation of the Kubernetes scheduler is to retrieve each newly created pod definition from the API server and assign it to a node. It finds suitable nodes for every pod like moving from unhealthier to healthier node, etc. However, for the scheduler to do its job and allow declarative placement, the scheduler needs nodes with available capacity and containers with declarative resource profiles and guiding policies.

**Singleton Service**

This pattern ensures only one instance of a service is active at a time and yet highly available. Pods can scale with the command kubectl scale or declaratively through a definition replica set. Running multiple instances of the same services increases the throughput and availability. In Kubernetes, multiple instances are replicas of a pod. For some use cases, you may need to run only one instance of a service. For example, when polling on specific payment interfaces, you want to ensure only single resources to perform polling and processing. In these kinds of services, you need to have control of several instances of services.

Running multiple replicas of the same pod creates an active-active topology where all instances of services are active. You need an active-passive topology where only one instance is active and all other instances are passive. This can be achieved with out-of-application and in-application locking.

Out-of-application locking can be achieved in Kubernetes by starting a pod with one replica. This alone does not help you to make a singleton; along with this, the replica set turns the singleton pod into a highly available singleton.

In-application locking, in a distributed environment, is one way to control the service instance count through a distributed lock. Whenever a service instance or a component inside the instance is activated, it can try to acquire a lock, and if it succeeds, the service becomes active. Any subsequent service instance that fails to acquire lock waits continuously tries to get the lock.

**Init Container**

This pattern enables the separation of concerns by providing a separate lifecycle for initialization-related tasks distinct from the main application containers. You can find more details of this pattern in Chapter [4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_4_Chapter.xhtml).

**Sidecar**

It extends and enhances the functionality of a preexisting container without changing it. This pattern allows you to add several additional configuration details from a third party without modifying the microservices. It is a single-node pattern made up of two containers. One container for the application container contains the core business logic, and another container is for technical configuration details. You can find more details of this pattern in Chapter [4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_4_Chapter.xhtml).

**Running a Cloud Native Application on the Container and Kubernetes Strategy**

Containers and Kubernetes are mature, but the ecosystem is immature due to the lack of operational best practices in organizations. However, the adoption of container and Kubernetes is increasing every day with the evolution of cloud native elements. Organizations are adopting containers in production, but production deployments are still concerned with operational challenges such as security, observability, data management, infrastructure, and networking, and most important is automation because cloud native services require a high degree of end-to-end automation.

If you want mature and streamlined containers and Kubernetes in production, you need to embrace best practices and strategy, and you need strong leadership.

* You need to have a strong DevSecOps culture to ensure a seamless move to production.
* Develop a Kubernetes platform that uses best practices and patterns across security, governance, observability, lifecycle, and cloud provider selections.
* Create an intelligent single operations team and development team.
* You need the right talent to create a roadmap to upskill your resources on containers and Kubernetes.

The most important point is to select the right Kubernetes platform because there are various platforms available in the industry. You can consider the following factors while selecting the platforms:

* Support for OS and container runtimes
* UI and application lifecycle management
* Hybrid, private, and multicloud cluster management
* Operational capabilities such as governance, security, networking, automation, and observability
* PaaS adoption for other services in your enterprises
* Licensing and pricing model
* Industry maturity of a vendor

For deploying containers in production, you need to create a strategy to operationalize Kubernetes. The following are the elements and best practices:

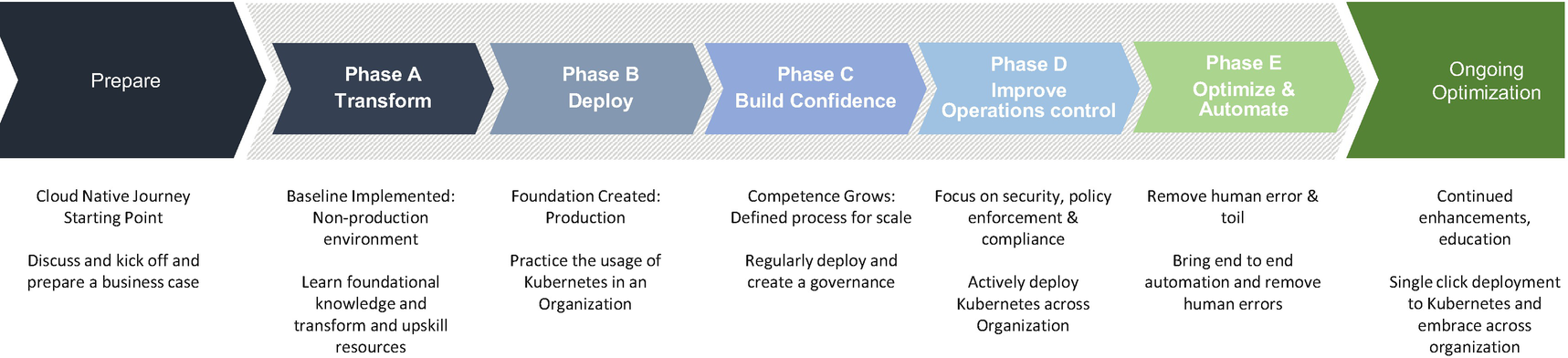
* *Security and governance*: Integrate container scanning and image scanning to prevent vulnerabilities along with the CI/CD pipeline. Use the configuration-as-a-service model to harden the configuration and deploy security products that provide whitelisting, behavioral monitoring, etc. Adopt a shift-left approach for code and security vulnerability tests.
* *Automation including infrastructure as code*: Automate infrastructure provisioning by using an infrastructure-as-code tool, use the container-aware configuration management system to manage the lifecycle of a container image, and integrate containers and Kubernetes with CI/CD toolchains.
* *Observability*: Focus on monitoring at the container level and across services so that you are monitoring your container both internally and externally. Use container commands and the right tools.
* *Networking host* : Check that your Kubernetes distribution or software-defined networking (SDN) solution supports Kubernetes networking. If this is not available, select the Container Networking Interface (CNI), and ensure it provides an ingress controller support for load balancing across hosts in the cluster. If it is not sufficient, then consider other proxies or service meshes or event meshes. Along with this strategy, you need the following service to make Kubernetes production ready.
  + *Cluster monitoring and logging*: When running in production, containers and Kubernetes are required to scale to hundreds or thousands of pods depending on the size of your enterprise. Without the effective implementation of monitoring and logging, downtime can cause serious or irreversible errors that can cause business dissatisfaction. Create integrated monitoring by using various open source or commercial tools.
  + *Reserved compute resources for daemons*: Reserve resources for system daemons, which both Kubernetes and OS require. The system daemon utilizes CPU, memory, and temporary storage resources. You can use Kubelet flags to reserve resources for system daemons.
  + *Heartbeat and election timeout interval for etcd members*: When configuring an etcd cluster, it is important to specify the heartbeat correctly and choose timeout settings.
  + *Regular etcd backups*: Regularly back up etcd data because it stores the state of the cluster.

**Kubernetes Maturity Model**

If your organization is new to Kubernetes or has already been using Kubernetes in multiple applications, Kubernetes has a complexity that you’ll need to overcome. The Kubernetes maturity model shown in Figure [16-12](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig12) provides the end-to-end journey of Kubernetes adoption. One thing you need to remember is that the maturity doesn’t occur in one day. This requires a certain amount of time.

If you start using the maturity model, know that when you do reach a certain phase, you still may go back to the previous phase to check certain things. You need to use this framework to understand where you are and what you need to focus on.

The Kubernetes maturity model helps you to review where you are in your cloud native journey, like whether you are new to Kubernetes or you have deployment experience.



***Figure 16-12***

Kubernetes maturity model

**Prepare**

This is the first phase in the maturity model, and this is the preparation phase of your cloud native Kubernetes journey. In this phase, you do the following:

* You will anticipate how cloud native and Kubernetes can help you to support your business and technical objectives, cost, and end goal.
* You will prepare a strategy for your organization on the importance of cloud native and Kubernetes.
* You will prepare the value proposition and impact of cloud computing, containers, and Kubernetes.

**Transform**

You will start adopting Kubernetes in this phase.

* You will verify foundational knowledge and create an MVP on Kubernetes deployments in a cluster.
* You will prepare an initial implementation, migration, and learning curve roadmap.
* You will start socializing across teams on Kubernetes.

**Deploy**

If you reached this maturity, you will have covered the basics and initial steps of Kubernetes. In this phase, you will do the following:

* One service must be deployed into production. External dependencies are configured properly, and traffic to your services are routed to Kubernetes through a load balancer.
* Logs and metrics are accessible and configured for autoscaling to Kubernetes.
* You will cover the implementation, build, and deployment process, setting up DevOps and introducing basic observability.

**Build Confidence**

As your Kubernetes deployment matures, you have a foundation in place. This phase is to build confidence in your organization and get Kubernetes up and running in production. Building confidence in Kubernetes requires experience, and the business outcome depends on your team’s experience. In this phase, monitoring will be implemented.

**Improve Operations**

You are actively deploying Kubernetes across organizational successfully and on time to improve the “-ilities” and operationalizing your Kubernetes clusters.

**Measure and Control**

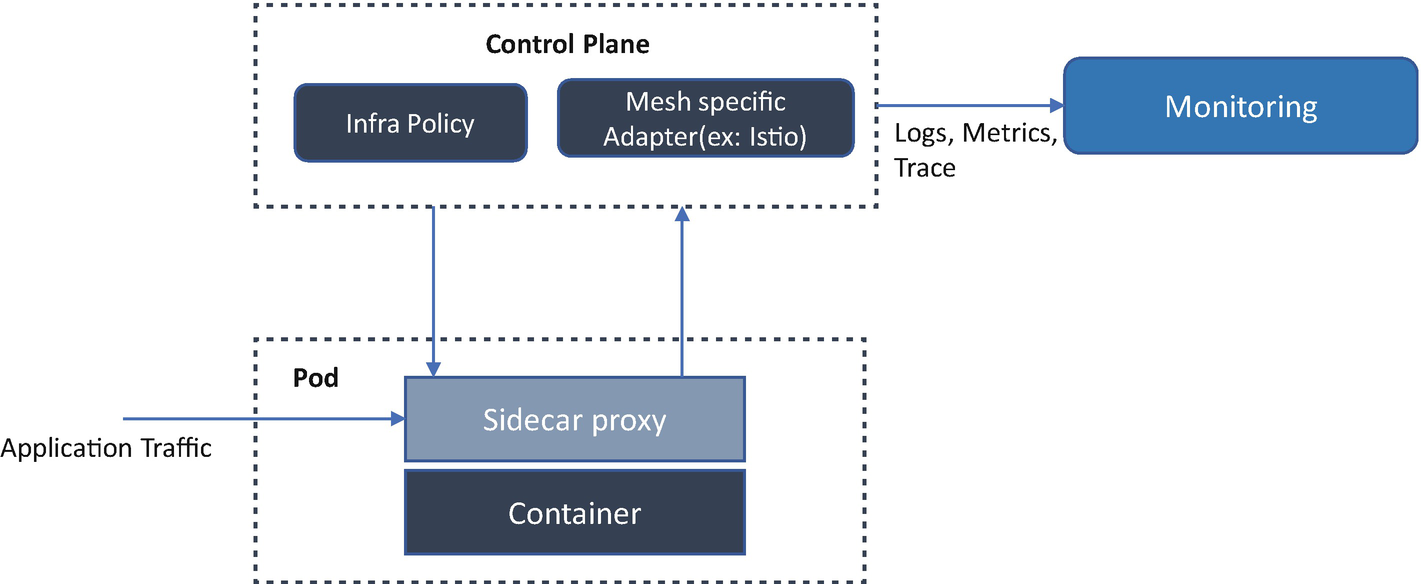
In this phase, you introduce more measurement and control of the Kubernetes environment. You and your teams have an overall understanding, and there is an organization-wide option. You have started understanding the Kubernetes cluster and overall environment on a deeper level. You will gather more data and resolve the technical debt identified in previous phases and will streamline the monitoring and observability.

**Optimize and Automate**

In this phase, you introduce more measurement and control of the environment, achieve business outcomes, and have measurable results to show to various stakeholders. You make further improvements on cost and performance metrics. You are required to revisit your earlier goals and fine-tune them based on the learnings. In this phase, you need to automate as much as possible and adopt best practices and principles.

**Service Meshes and Kubernetes**

A service mesh pattern is a logical extension of the sidecar proxy. As shown in Figure [16-13](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig13), by attaching a sidecar proxy to every pod, a service mesh can control functionality for service-to-service requests, such as advanced routing rules, retries, and timeouts. Along with every request pass through a proxy, service meshes can implement mutual TLS encryption between services. There are several service mesh tools available, such as Istio, Linkerd, Kuma, and Consul. Istio is the most popular implementation of the service mesh pattern. You can find more details about using service meshes in Kubernetes in Chapter [5](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_5_Chapter.xhtml).



***Figure 16-13***

Service meshes and Kubernetes

**Stateful Workloads on Kubernetes**

Usually, services in cloud native are stateless, which is the recommended approach, but some use cases require stateful services. An example of a stateful application is a database or key-value store in which the data is retrieved and stored by other services or applications.

You can deploy the stateful application in Kubernetes with a ReplicaSet or deployment and use StatefulSets. Many distributed stateful applications have their clustering mechanism or consensus algorithm. For this kind of application, the StatefulSets provide static pod naming based on an ordinal system.

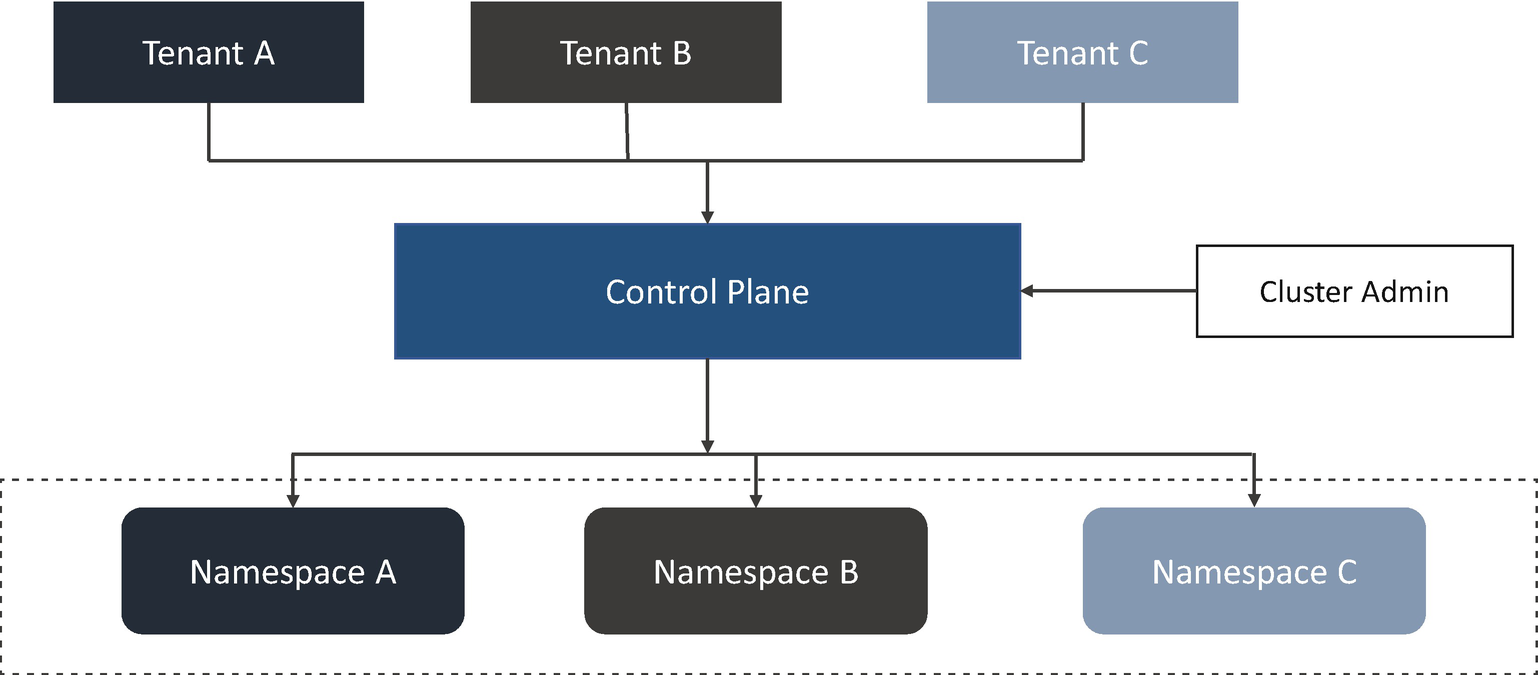
To illustrate how StatefulSets can help run the stateful application on Kubernetes, let’s look at how you might run PostgreSQL on Kubernetes with StatefulSets. Running PostgreSQL on Kubernetes requires a container image and makes sure it has all the necessary configuration and startup commands.

Scaling PostgreSQL is not like running stateless applications; here you can scale your service without creating a new state. Each member of the PostgreSQL cluster knows about the other members, and most importantly, it knows which member of the cluster is the leader. This is how databases like PostgreSQL can offer a consistency guarantee and ACID compliance. Since each member in a PostgreSQL cluster needs to know about the other, you need to run your pods in a way that they have a common way to communicate with each other. The StatefulSets offer is through ordinal pod numbering. This way, the application that needs to self-cluster while running on Kubernetes knows that a common naming scheme will be used. You can find more about the replica sets in Chapter [4](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_4_Chapter.xhtml).

**Kubernetes Multitenancy**

A multitenant cluster is shared by multiple users and services. The operators of multitenant clusters must isolate tenants from each other to minimize the effect. As shown in Figure [16-14](https://learning.oreilly.com/library/view/cloud-native-architecture/9781484272268/html/511610_1_En_16_Chapter.xhtml#Fig14), when you plan a multitenant in a deployment model, you must consider the layers of resource isolation in Kubernetes, cluster, namespace, node, pod, and container. You must consider security implications when sharing among tenants.

Multitenancy capabilities aim to drive the efficient use of infrastructure while providing operators with robust isolation mechanisms between users, services, and teams. Kubernetes allows you to build multitenant platforms leveraging built-in capabilities.



***Figure 16-14***

Multitenancy in Kubernetes

Kubernetes cannot guarantee perfectly secure isolation between tenants; you can separate each tenant and their resources into their namespaces, roles and role bindings, resource quota, network policy, etc. You can use policies to enforce isolation; it can be scoped by namespace and can be used to restrict API access to constrain resource usage.

The control plane is important as you bring in more tenants because it becomes a single point of failure for clusters. The Kubernetes API is a critical service for you because it provides the interface for administrators and tenant teams to manage clusters and services. So, the Kubernetes response of an API is a critical part of the multitenant strategy.

The tenants of a multicluster share extensions, controllers, add-ons, and custom resource definition and cluster control plane.

The multitenant comes with advantages, such as reduced managed overhead, reduced resource fragmentation, improved return on investment (ROI), etc.

**Kubernetes Secrets**

Kubernetes secrets are the native resources for storing and managing sensitive data such as passwords, SSH keys, and OAuth tokens. You need to distribute these secrets across your Kubernetes clusters. When sending these secrets, it’s critical to ensure only authorized entities can access them.

Secrets are native Kubernetes resources, and Kubernetes provides a basic set of protection layers to them. These protections are as follows:

* *Secret resources*: Pods and secrets are separate objects; you can expose secrets during the pod lifecycle. If you are passing sensitive information as an environment variable to pods, you should create separate secret objects.
* *Kubelet*: This is an agent that runs on each Kubernetes node and interacts with the container at runtime. Data in the secrets is used inside containers and available to the node where the containers run. It stores secret data in a temp file instead of disks.
* *Pods*: Numerous pods are running on the node, but only the pods can access secrets specified in the definition. Pods consist of several containers, and secrets are mounted only on required containers.
* *Kubernetes API*: Secrets are created and accessed over the Kubernetes API.
* *etcd*: Secrets also stored in etcd; it’s possible to access secrets when you access etcd on the control plane.

These protection measures ensure that secrets are separated from Kubernetes resources, accessed, and stored securely, but Kubernetes is not highly secured but comes with risks. The risks are as follows:

* *Configuration as code (CaC)* : You can create a secret object using JSON or YAML manifest files. Make sure this file isn’t checked into the repository or shared.
* *Service layer*: When secrets are loaded in your services, be careful about logging or monitoring.
* *Pods*: If a user has sufficient permissions to create a pod that mounts and uses a secret, the secret value will visible to them.
* *Nodes*: Containers run on the nodes, and it is possible to retrieve any secret from the Kubernetes API server if you’re the root on the node.

Various tools and services are available in cloud native to manage secrets. The following are the few services you can adopt to manage secrets:

* *Cloud* *key management system (KMS)* : The cloud providers such as AWS, Azure, and Google have their KMS, a centralized cloud service through which you can create and manage keys to perform cryptographic operations.
* *Helm secret plugins*: This plugin allows you to encrypt values files with a secret key of your choice. It is also possible to edit the encrypted files.

**Kubernetes as a Service**

Kubernetes as a service (KaaS) makes it possible to operate Kubernetes in a cloud environment. These services are commonly provided by cloud vendors. The functionality of the KaaS platform is to deploy, manage, and maintain Kubernetes clusters. Key features of KaaS include self-service deployment, upgrades, scalability, and multicloud portability. Here I will provide a brief description. For more details, you can refer to each cloud provider’s documentation.

KaaS can help you leverage the best practices of Kubernetes without the complexities involved in managing operations. A KaaS can take care of a variety of services including setup, monitoring, and managing the operations and ensuring HA and release updates as needed. The following are a few key capabilities of KaaS:

* Deploy and manage
* Continuous monitoring
* Control plane management
* Security

**Google Kubernetes Engine**

Google Kubernetes Engine (GKE) was the first commercial KaaS offering. It is a fully managed containerized service in the Google Cloud infrastructure. The GKE environment consists of multiple machines grouped to form a cluster. When you run the GKE cluster, you will get advanced cluster management features such as the following:

* Autopilot mode of operation
* Pod and clustering autoscaling
* Workload and network security
* Node pools to designate subsets of nodes within a cluster for additional flexibility
* Logging and monitoring

**Amazon Elastic Kubernetes Service**

Amazon Elastic Kubernetes Service (EKS) is a service used to run managed Kubernetes on AWS. It can deploy clusters across multiple availability zones (AZs) with HA. EKS integrates with other services in an AWS. EKS helps you provide highly available and secure clusters and automates key tasks such as patching, node provisioning, etc. The following are some of the benefits of EKS:

* EKS runs the Kubernetes control plane across multiple AZs, automatically detects and replaces unhealthy control plane nodes, and provides on-demand, zero-downtime upgrades.
* Provision and scale your services efficiently.

**Azure Kubernetes Services**

Azure Kubernetes Services (AKS) is a fully managed service that lets you manage Kubernetes on Azure resources. It allows you to deploy directly on Azure services and also integrate with existing Azure services. These are some of the benefits of AKS:

* Elastic provisioning of capacity without the need to manage the infrastructure
* Faster end-to-end development experience with Azure services
* Most comprehensive authentication and authorization capabilities
* Availability in more regions

**Red Hat OpenShift**

Red Hat OpenShift is a highly customizable managed service you can use to deploy Kubernetes to any infrastructure. It supports multitenancy, has a built-in dedicated image registry, and provides extended support of the DevOps pipeline. It has several preconfigured packages.

**VMware Tanzu**

It is a platform that enables you to build and manage Kubernetes environments, alongside traditional VMware workloads, with central control. It enables integrated Kubernetes with VMware technologies such as vSphere, vSAN, and NSX, to manage VMware Kubernetes clusters within the data center.

**Summary**

Cloud native infrastructure is about practices and how you build and maintain infrastructure. It impacts much more than servers, networks, and storage; it is about how you manage services in a cloud native infrastructure.

In this chapter, I explained the elements of cloud native infrastructure and how you can use containers and Kubernetes to deploy your cloud native services.