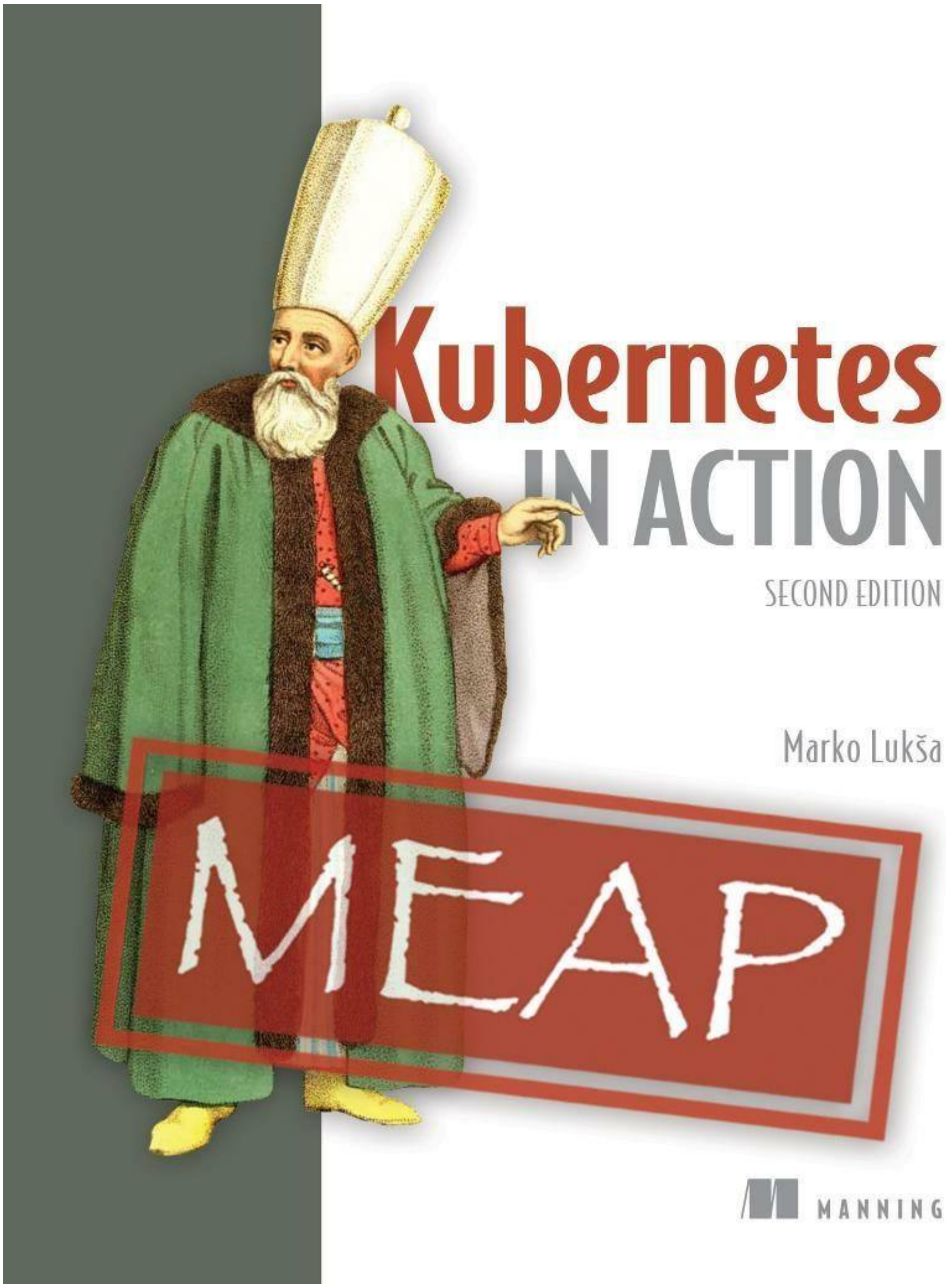
Kubernetes IN ACTION

SECOND EDITION

Marko Lukša

MEAP

MANNING


Kubernetes
IN ACTION
SECOND EDITION
Marko Lukša
MEAP
MANNING

Kubernetes in Action, Second Edition MEAP V15

MEAP Edition

Manning Early Access Program

Kubernetes in Action

Second edition

Version 15

welcome

Thank you for purchasing the MEAP for *Kubernetes in Action, Second Edition*.

As part of my work at Red Hat, I started using Kubernetes in 2014, even before version 1.0 was released. Those were interesting times. Not many people working in the software industry knew about Kubernetes, and there was no real community yet. There were hardly any blog posts about it and the documentation was still very basic. Kubernetes itself was ridden with bugs. When you combine all these facts, you can imagine that working with Kubernetes was extremely difficult.

In 2015 I was asked by Manning to write the first edition of this book. The originally planned 300-page book grew to over 600 pages full of information. The writing forced me to also research those parts of Kubernetes that I wouldn't have looked at more closely otherwise. I put most of what I learned into the book. Judging by their reviews and comments, readers love a detailed book like this.

The plan for the second edition of the book is to add even more information and to rearrange some of the existing content. The exercises in this book will take you from deploying a trivial application that initially uses only the basic features of Kubernetes to a full-fledged application that incorporates additional features as the book introduces them.

The book is divided into five parts. In the first part, after the introduction of Kubernetes and containers, you'll deploy the application in the simplest way. In the second part you'll learn the main concepts used to describe and deploy your application. After that you'll explore the inner workings of Kubernetes components. This will give you a good foundation to learn the difficult part - how to manage Kubernetes in production. In the last part of the book you'll learn about best practices and how to extend Kubernetes.

I hope you all like this second edition even better than the first, and if you're reading the book for the first time, your feedback will be even more valuable. If any part of the book is difficult to understand, please post your questions, comments or suggestions in the liveBook forum.

Thank you for helping me write the best book possible.

—Marko Lukša

In this book

1 Introducing Kubernetes

This chapter covers

- Introductory information about Kubernetes and its origins
- Why Kubernetes has seen such wide adoption
- How Kubernetes transforms your data center
- An overview of its architecture and operation
- How and if you should integrate Kubernetes into your own organization

Before you can learn about the ins and outs of running applications with Kubernetes, you must first gain a basic understanding of the problems Kubernetes is designed to solve, how it came about, and its impact on application development and deployment. This first chapter is intended to give a general overview of these topics.

1.1 Introducing Kubernetes

The word *Kubernetes* is Greek for pilot or helmsman, the person who steers the ship - the person standing at the helm (the ship's wheel). A helmsman is not necessarily the same as a captain. A captain is responsible for the ship, while the helmsman is the one who steers it.

After learning more about what Kubernetes does, you'll find that the name hits the spot perfectly. A helmsman maintains the course of the ship, carries out the orders given by the captain and reports back the ship's heading. Kubernetes steers your applications and reports on their status while you - the captain - decide where you want the system to go.

How to pronounce Kubernetes and what is k8s?

The correct Greek pronunciation of Kubernetes, which is *Kie-ver-nee-tees*, is different from the English pronunciation you normally hear in technical conversations. Most often it's *Koo-ber-netties* or *Koo-ber-nay'-tace*, but you may also hear *Koo-ber-nets*, although rarely.

In both written and oral conversations, it's also referred to as *Kube* or *K8s,* pronounced *Kates*, where the 8 signifies the number of letters omitted between the first and last letter.

1.1.1 Kubernetes in a nutshell

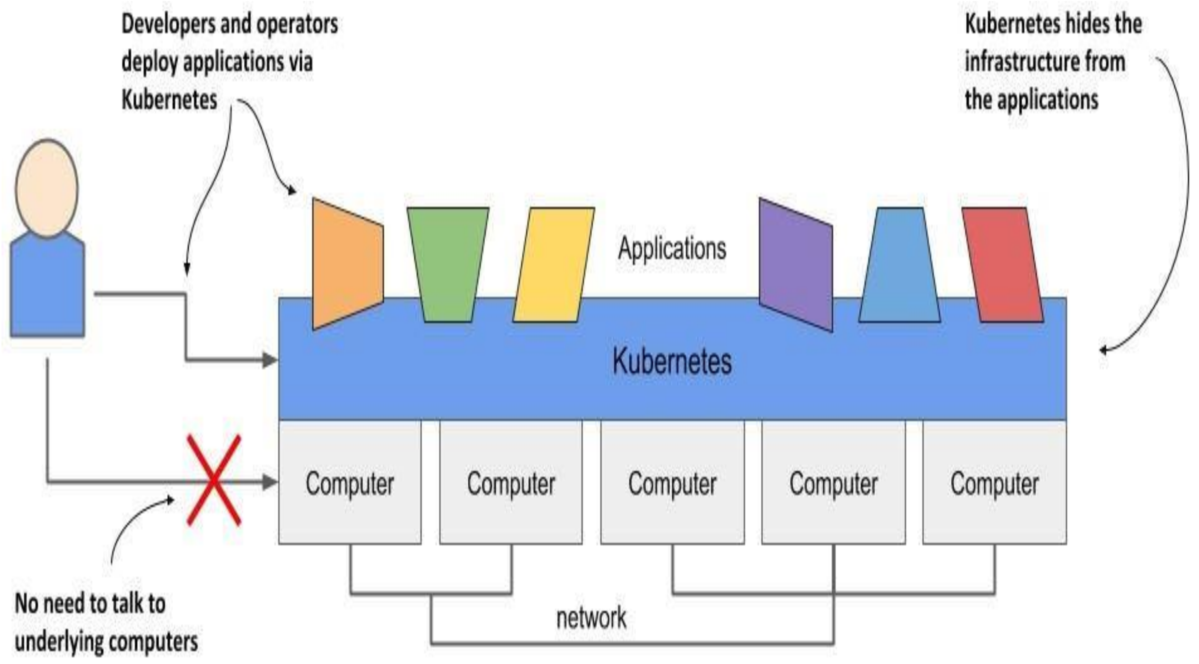
Kubernetes is a software system for automating the deployment and management of complex, large-scale application systems composed of computer processes running in containers. Let's learn what it does and how it does it.

Abstracting away the infrastructure

When software developers or operators decide to deploy an application, they do this through Kubernetes instead of deploying the application to individual computers. Kubernetes provides an abstraction layer over the underlying hardware to both users and applications.

As you can see in the following figure, the underlying infrastructure, meaning the computers, the network and other components, is hidden from the applications, making it easier to develop and configure them.

Figure 1.1 Infrastructure abstraction using Kubernetes


Developers and operators deploy applications via Kubernetes
Kubernetes hides the infrastructure from the applications
Applications
Kubernetes
Computer
Computer
Computer
Computer
Computer
network
No need to talk to underlying computers

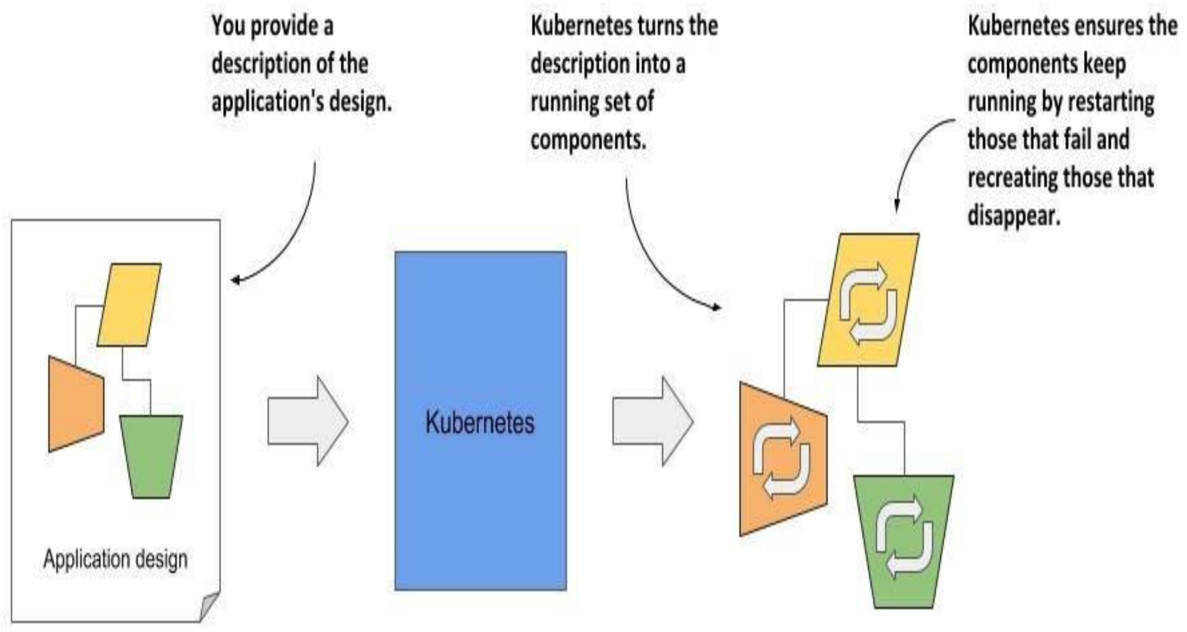
Standardizing how we deploy applications

Because the details of the underlying infrastructure no longer affect the deployment of applications, you deploy applications to your corporate data center in the same way as you do in the cloud. A single manifest that describes the application can be used for local deployment and for deploying on any cloud provider. All differences in the underlying infrastructure are handled by Kubernetes, so you can focus on the application and the business logic it contains.

Deploying applications declaratively

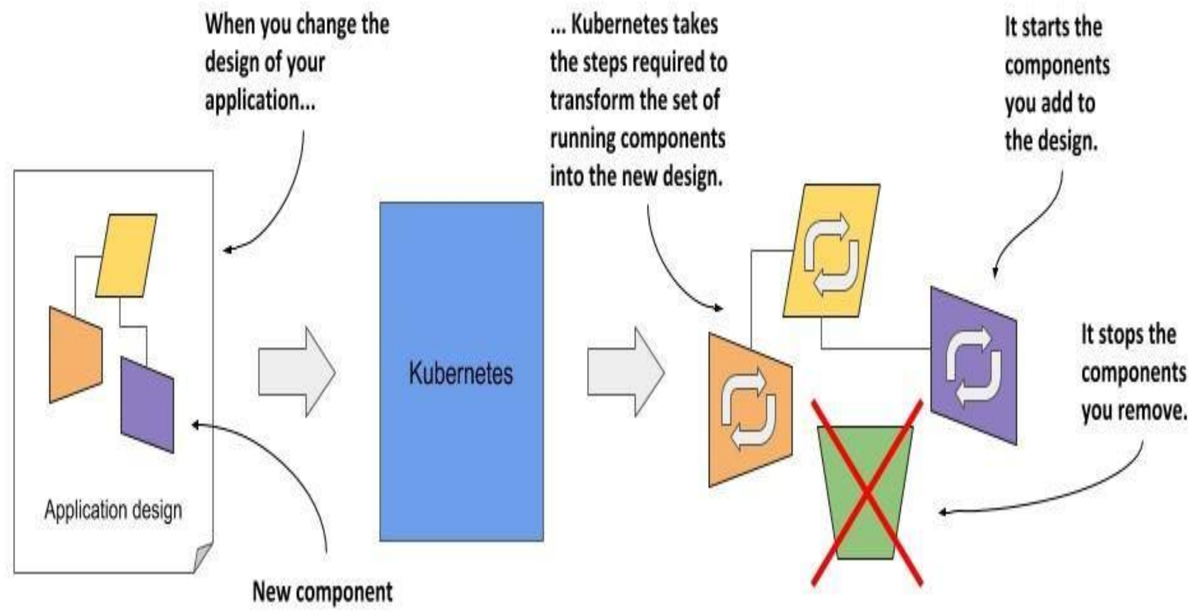
Kubernetes uses a declarative model to define an application, as shown in the next figure. You describe the components that make up your application and Kubernetes turns this description into a running application. It then keeps the application healthy by restarting or recreating parts of it as needed.

Figure 1.2 The declarative model of application deployment


You provide a description of the application's design.
Kubernetes turns the description into a running set of components.
Kubernetes ensures the components keep running by restarting those that fail and recreating those that disappear.
Application design
Kubernetes

Whenever you change the description, Kubernetes will take the necessary steps to reconfigure the running application to match the new description, as shown in the next figure.

Figure 1.3 Changes in the description are reflected in the running application

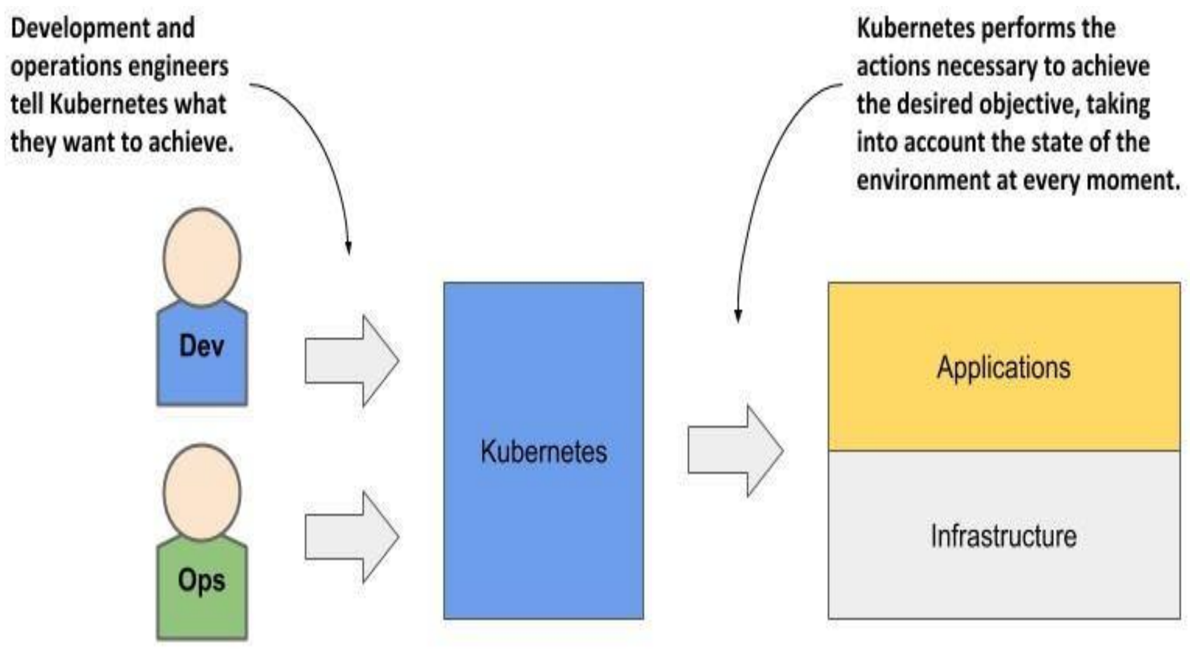

When you change the design of your application...
... Kubernetes takes the steps required to transform the set of running components into the new design.
It starts the components you add to the design.
It stops the components you remove.
Application design
Kubernetes
New component

Taking on the daily management of applications

As soon as you deploy an application to Kubernetes, it takes over the daily management of the application. If the application fails, Kubernetes will automatically restart it. If the hardware fails or the infrastructure topology changes so that the application needs to be moved to other machines, Kubernetes does this all by itself. The engineers responsible for operating the system can focus on the big picture instead of wasting time on the details.

To circle back to the sailing analogy: the development and operations engineers are the ship's officers who make high-level decisions while sitting comfortably in their armchairs, and Kubernetes is the helmsman who takes care of the low-level tasks of steering the system through the rough waters your applications and infrastructure sail through.

Figure 1.4 Kubernetes takes over the management of applications


Development and operations engineers tell Kubernetes what they want to achieve.
Kubernetes performs the actions necessary to achieve the desired objective, taking into account the state of the environment at every moment.
Dev
Ops
Kubernetes
Applications
Infrastructure

Everything that Kubernetes does and all the advantages it brings requires a longer explanation, which we'll discuss later. Before we do that, it might help you to know how it all began and where the Kubernetes project currently stands.

1.1.2 About the Kubernetes project

Kubernetes was originally developed by Google. Google has practically always run applications in containers. As early as 2014, it was reported that they start two billion containers every week. That's over 3,000 containers per second, and the figure is much higher today. They run these containers on thousands of computers distributed across dozens of data centers around the world. Now imagine doing all this manually. It's clear that you need automation, and at this massive scale, it better be perfect.

About Borg and Omega - the predecessors of Kubernetes

The sheer scale of Google's workload has forced them to develop solutions to make the development and management of thousands of software components manageable and cost-effective. Over the years, Google developed an internal system called *Borg* (and later a new system called Omega) that helped both application developers and operators manage these thousands of applications and services.

In addition to simplifying development and management, these systems have also helped them to achieve better utilization of their infrastructure. This is important in any organization, but when you operate hundreds of thousands of machines, even tiny improvements in utilization mean savings in the millions, so the incentives for developing such a system are clear.

Note

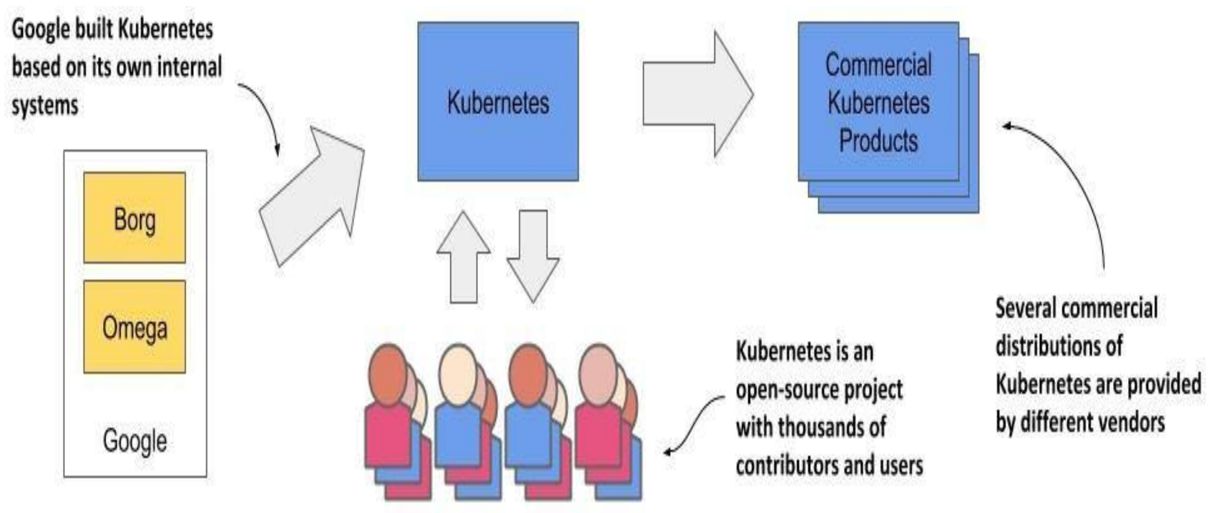
Data on Google's energy use suggests that they run around 900,000 servers.

Over time, your infrastructure grows and evolves. Every new data center is state-of-the-art. Its infrastructure differs from those built in the past. Despite the differences, the deployment of applications in one data center should not differ from deployment in another data center. This is especially important when you deploy your application across multiple zones or regions to reduce the likelihood that a regional failure will cause application downtime. To do this effectively, it's worth having a consistent method for deploying your applications.

About Kubernetes - the open-source project - and commercial products derived from it

Based on the experience they gained while developing Borg, Omega and other internal systems, in 2014 Google introduced Kubernetes, an opensource project that can now be used and further improved by everyone.

Figure 1.5 The origins and state of the Kubernetes open-source project


Google built Kubernetes based on its own internal systems
Borg
Omega
Google
Kubernetes
Commercial Kubernetes Products
Kubernetes is an open-source project with thousands of contributors and users
Several commercial distributions of Kubernetes are provided by different vendors

As soon as Kubernetes was announced, long before version 1.0 was officially released, other companies, such as Red Hat, who has always been at the forefront of open-source software, quickly stepped on board and helped develop the project. It eventually grew far beyond the expectations of its founders, and today is arguably one of the world's leading open-source projects, with dozens of organizations and thousands of individuals contributing to it.

Several companies are now offering enterprise-quality Kubernetes products that are built from the open-source project. These include Red Hat OpenShift, Pivotal Container Service, Rancher and many others.

How Kubernetes grew a whole new cloud-native eco-system

Kubernetes has also spawned many other related open-source projects, most of which are now under the umbrella of the *Cloud Native Computing Foundation* (CNCF), which is part of the *Linux Foundation*.

CNCF organizes several KubeCon - CloudNativeCon conferences per year - in North America, Europe and China. In 2019, the total number of attendees exceeded 23,000, with KubeCon North America reaching an overwhelming number of 12,000 participants. These figures show that Kubernetes has had an incredibly positive impact on the way companies around the world deploy applications today. It wouldn't have been so widely adopted if that wasn't the case.

1.1.3 Understanding why Kubernetes is so popular

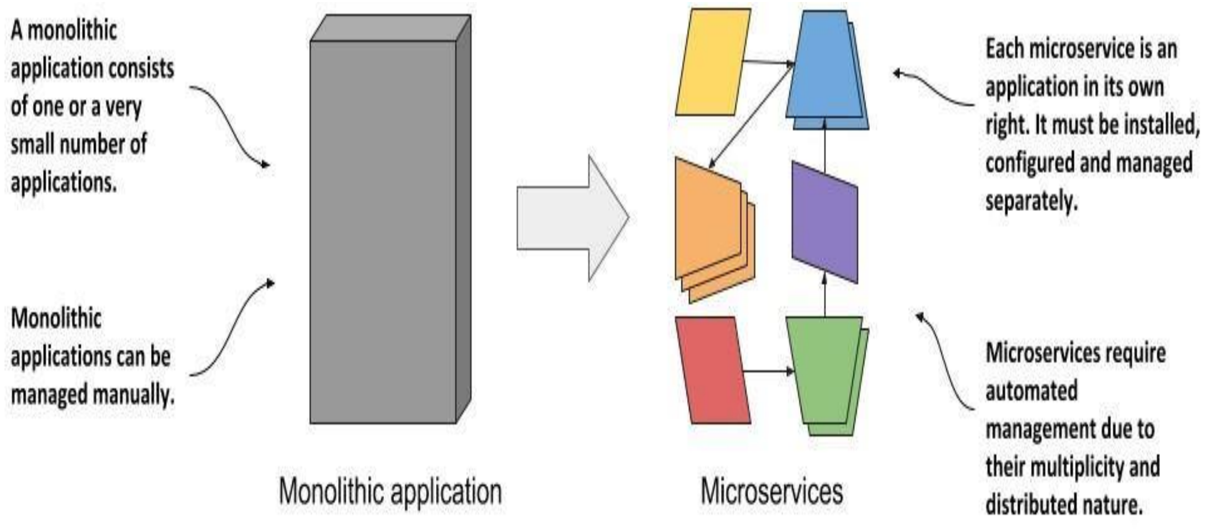
In recent years, the way we develop applications has changed considerably. This has led to the development of new tools like Kubernetes, which in turn have fed back and fuelled further changes in application architecture and the way we develop them. Let's look at concrete examples of this.

Automating the management of microservices

In the past, most applications were large monoliths. The components of the application were tightly coupled, and they all ran in a single computer process. The application was developed as a unit by a large team of developers and the deployment of the application was straightforward. You installed it on a powerful computer and provided the little configuration it required. Scaling the application horizontally was rarely possible, so whenever you needed to increase the capacity of the application, you had to upgrade the hardware - in other words, scale the application vertically.

Then came the microservices paradigm. The monoliths were divided into dozens, sometimes hundreds, of separate processes, as shown in the following figure. This allowed organizations to divide their development departments into smaller teams where each team developed only a part of the entire system - just some of the microservices.

Figure 1.6 Comparing monolithic applications with microservices


A monolithic application consists of one or a very small number of applications.
Monolithic applications can be managed manually.
Monolithic application
Microservices
Each microservice is an application in its own right. It must be installed, configured and managed separately.
Microservices require automated management due to their multiplicity and distributed nature.

Each microservice is now a separate application with its own development and release cycle. The dependencies of different microservices will inevitably diverge over time. One microservice requires one version of a library, while another microservice requires another, possibly incompatible, version of the same library. Running the two applications in the same operating system becomes difficult.

Fortunately, containers alone solve this problem where each microservice requires a different environment, but each microservice is now a separate application that must be managed individually. The increased number of applications makes this much more difficult.

Individual parts of the entire application no longer need to run on the same computer, which makes it easier to scale the entire system, but also means that the applications need to be configured to communicate with each other. For systems with only a handful of components, this can usually be done manually, but it’s now common to see deployments with well over a hundred microservices.

When the system consists of many microservices, automated management is crucial. Kubernetes provides this automation. The features it offers make the task of managing hundreds of microservices almost trivial.

Bridging the dev and ops divide

Along with these changes in application architecture, we’ve also seen changes in the way teams develop and run software. It used to be normal for a development team to build the software in isolation and then throw the finished product over the wall to the operations team, who would then deploy it and manage it from there.

With the advent of the Dev-ops paradigm, the two teams now work much more closely together throughout the entire life of the software product. The development team is now much more involved in the daily management of the deployed software. But that means that they now need to know about the infrastructure on which it’s running.

As a software developer, your primary focus is on implementing the business logic. You don't want to deal with the details of the underlying servers. Fortunately, Kubernetes hides these details.

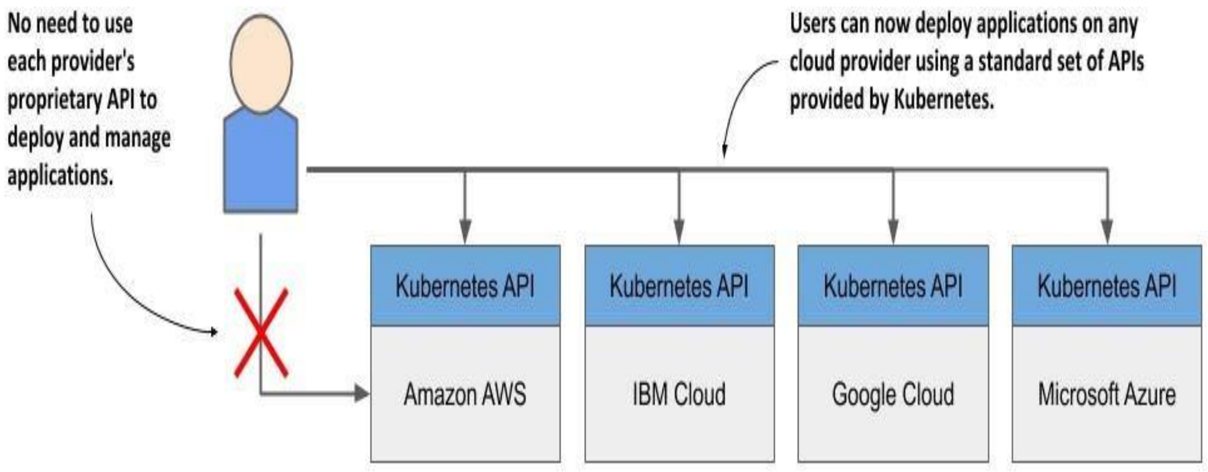
Standardizing the cloud

Over the past decade or two, many organizations have moved their software from local servers to the cloud. The benefits of this seem to have outweighed the fear of being locked-in to a particular cloud provider, which is caused by relying on the provider's proprietary APIs to deploy and manage applications.

Any company that wants to be able to move its applications from one provider to another will have to make additional, initially unnecessary efforts to abstract the infrastructure and APIs of the underlying cloud provider from the applications. This requires resources that could otherwise be focused on building the primary business logic.

Kubernetes has also helped in this respect. The popularity of Kubernetes has forced all major cloud providers to integrate Kubernetes into their offerings. Customers can now deploy applications to any cloud provider through a standard set of APIs provided by Kubernetes.

Figure 1.7 Kubernetes has standardized how you deploy applications on cloud providers


No need to use each provider's proprietary API to deploy and manage applications.
Users can now deploy applications on any cloud provider using a standard set of APIs provided by Kubernetes.
Kubernetes API
Kubernetes API
Kubernetes API
Kubernetes API
Amazon AWS
IBM Cloud
Google Cloud
Microsoft Azure

If the application is built on the APIs of Kubernetes instead of directly on the proprietary APIs of a specific cloud provider, it can be transferred relatively easily to any other provider.

1.2 Understanding Kubernetes

The previous section explained the origins of Kubernetes and the reasons for its wide adoption. In this section we'll take a closer look at what exactly Kubernetes is.

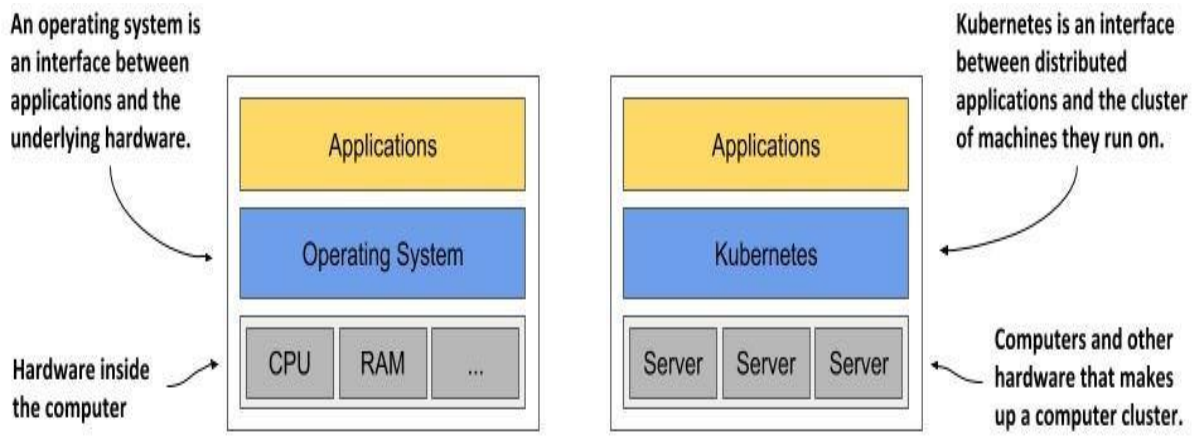
1.2.1 Understanding how Kubernetes transforms a computer cluster

Let's take a closer look at how the perception of the data center changes when you deploy Kubernetes on your servers.

Kubernetes is like an operating system for computer clusters

One can imagine Kubernetes as an operating system for the cluster. The next figure illustrates the analogies between an operating system running on a computer and Kubernetes running on a cluster of computers.

Figure 1.8 Kubernetes is to a computer cluster what an Operating System is to a computer


An operating system is an interface between applications and the underlying hardware.
Applications
Operating System
CPU
RAM
...
Hardware inside the computer
Applications
Kubernetes
Server
Server
Server
Kubernetes is an interface between distributed applications and the cluster of machines they run on.
Computers and other hardware that makes up a computer cluster.

Just as an operating system supports the basic functions of a computer, such as scheduling processes onto its CPUs and acting as an interface between the application and the computer's hardware, Kubernetes schedules the components of a distributed application onto individual computers in the underlying computer cluster and acts as an interface between the application and the cluster.

It frees application developers from the need to implement infrastructurerelated mechanisms in their applications; instead, they rely on Kubernetes to provide them. This includes things like:

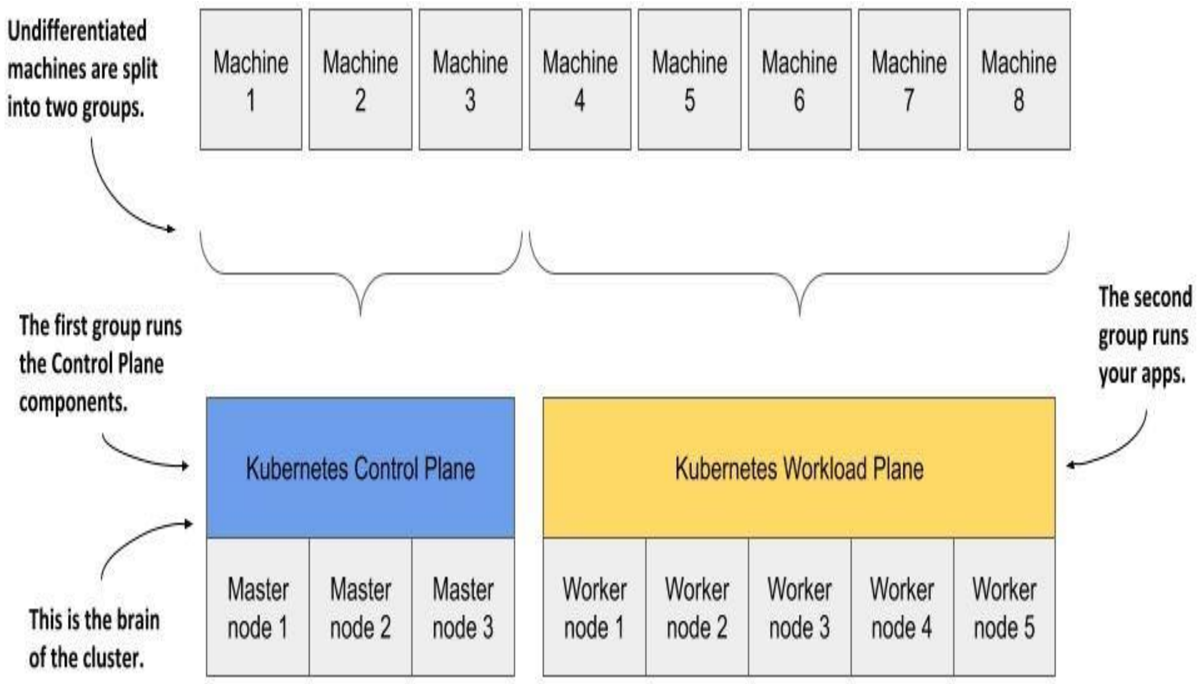
- *service discovery* - a mechanism that allows applications to find other applications and use the services they provide,
- *horizontal scaling* - replicating your application to adjust to fluctuations in load,
- *load-balancing* - distributing load across all the application replicas,
- *self-healing* - keeping the system healthy by automatically restarting failed applications and moving them to healthy nodes after their nodes fail,
- *leader election* - a mechanism that decides which instance of the application should be active while the others remain idle but ready to take over if the active instance fails.

By relying on Kubernetes to provide these features, application developers can focus on implementing the core business logic instead of wasting time integrating applications with the infrastructure.

How Kubernetes fits into a computer cluster

To get a concrete example of how Kubernetes is deployed onto a cluster of computers, look at the following figure.

Figure 1.9 Computers in a Kubernetes cluster are divided into the Control Plane and the Workload Plane



You start with a fleet of machines that you divide into two groups - the master and the worker nodes. The master nodes will run the Kubernetes Control Plane, which represents the brain of your system and controls the cluster, while the rest will run your applications - your workloads - and will therefore represent the Workload Plane.

Note

The Workload Plane is sometimes referred to as the Data Plane, but this term could be confusing because the plane doesn't host data but applications. Don't be confused by the term "plane" either - in this context you can think of it as the "surface" the applications run on.

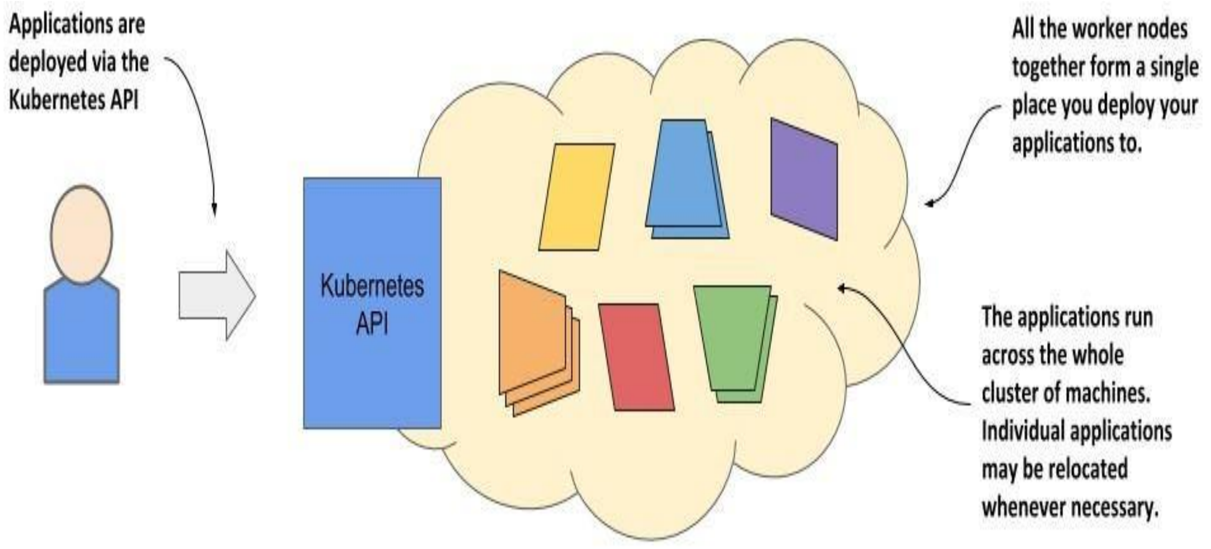
Non-production clusters can use a single master node, but highly available clusters use at least three physical master nodes to host the Control Plane. The number of worker nodes depends on the number of applications you'll deploy.

How all cluster nodes become one large deployment area

After Kubernetes is installed on the computers, you no longer need to think about individual computers when deploying applications. Regardless of the number of worker nodes in your cluster, they all become a single space where you deploy your applications. You do this using the

Kubernetes API, which is provided by the Kubernetes Control Plane.

Figure 1.10 Kubernetes exposes the cluster as a uniform deployment area


Applications are deployed via the Kubernetes API
Kubernetes API
All the worker nodes together form a single place you deploy your applications to.
The applications run across the whole cluster of machines. Individual applications may be relocated whenever necessary.

When I say that all worker nodes become one space, I don't want you to think that you can deploy an extremely large application that is spread across several small machines. Kubernetes doesn't do magic tricks like this. Each application must be small enough to fit on one of the worker nodes.

What I meant was that when deploying applications, it doesn't matter which worker node they end up on. Kubernetes may later even move the application from one node to another. You may not even notice when that happens, and you shouldn't care.

1.2.2 The benefits of using Kubernetes

You've already learned why many organizations across the world have welcomed Kubernetes into their data centers. Now, let's take a closer look at the specific benefits it brings to both development and IT operations teams.

Self-service deployment of applications

Because Kubernetes presents all its worker nodes as a single deployment surface, it no longer matters which node you deploy your application to. This means that developers can now deploy applications on their own, even if they don't know anything about the number of nodes or the characteristics of each node.

In the past, the system administrators were the ones who decided where each application should be placed. This task is now left to Kubernetes. This allows a developer to deploy applications without having to rely on other people to do so. When a developer deploys an application,
Kubernetes chooses the best node on which to run the application based on the resource requirements of the application and the resources available on each node.

Reducing costs via better infrastructure utilization

If you don't care which node your application lands on, it also means that it can be moved to any other node at any time without you having to worry about it. Kubernetes may need to do this to make room for a larger application that someone wants to deploy. This ability to move applications allows the applications to be packed tightly together so that the resources of the nodes can be utilized in the best possible way.

Note

In chapter 17 you'll learn more about how Kubernetes decides where to place each application and how you can influence the decision.

Finding optimal combinations can be challenging and time consuming, especially when the number of all possible options is huge, such as when you have many application components and many server nodes on which they can be deployed. Computers can perform this task much better and faster than humans. Kubernetes does it very well. By combining different applications on the same machines, Kubernetes improves the utilization of your hardware infrastructure so you can run more applications on fewer servers.

Automatically adjusting to changing load

Using Kubernetes to manage your deployed applications also means that the operations team doesn't have to constantly monitor the load of each application to respond to sudden load peaks. Kubernetes takes care of this also. It can monitor the resources consumed by each application and other metrics and adjust the number of running instances of each application to cope with increased load or resource usage.

When you run Kubernetes on cloud infrastructure, it can even increase the size of your cluster by provisioning additional nodes through the cloud provider’s API. This way, you never run out of space to run additional instances of your applications.

Keeping applications running smoothly

Kubernetes also makes every effort to ensure that your applications run smoothly. If your application crashes, Kubernetes will restart it automatically. So even if you have a broken application that runs out of memory after running for more than a few hours, Kubernetes will ensure that your application continues to provide the service to its users by automatically restarting it in this case.

Kubernetes is a self-healing system in that it deals with software errors like the one just described, but it also handles hardware failures. As clusters grow in size, the frequency of node failure also increases. For example, in a cluster with one hundred nodes and a MTBF (mean-time-betweenfailure) of 100 days for each node, you can expect one node to fail every day.

When a node fails, Kubernetes automatically moves applications to the remaining healthy nodes. The operations team no longer needs to manually move the application and can instead focus on repairing the node itself and returning it to the pool of available hardware resources.

If your infrastructure has enough free resources to allow normal system operation without the failed node, the operations team doesn’t even have to react immediately to the failure. If it occurs in the middle of the night, no one from the operations team even has to wake up. They can sleep peacefully and deal with the failed node during regular working hours.

Simplifying application development

The improvements described in the previous section mainly concern application deployment. But what about the process of application development? Does Kubernetes bring anything to their table? It definitely does.

As mentioned previously, Kubernetes offers infrastructure-related services that would otherwise have to be implemented in your applications. This includes the discovery of services and/or peers in a distributed application, leader election, centralized application configuration and others. Kubernetes provides this while keeping the application Kubernetesagnostic, but when required, applications can also query the Kubernetes API to obtain detailed information about their environment. They can also use the API to change the environment.

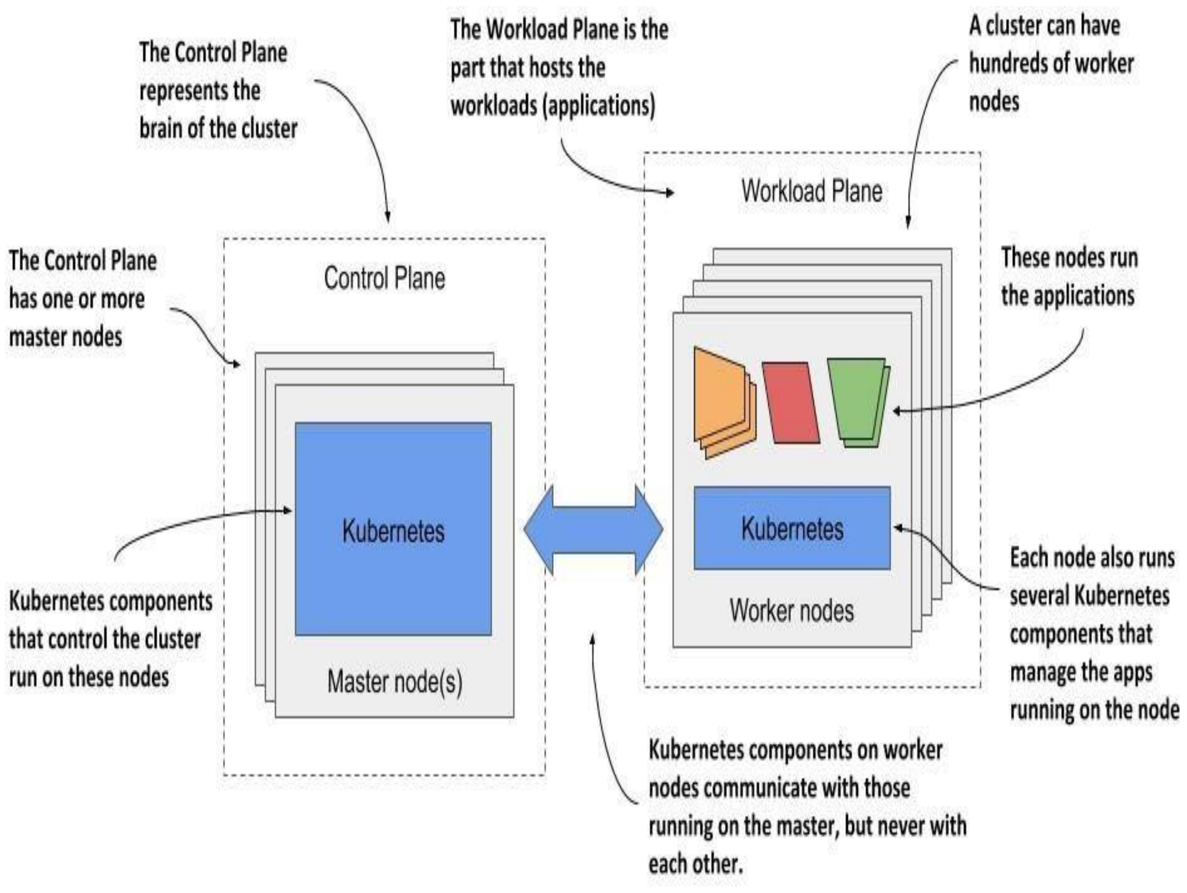
1.2.3 The architecture of a Kubernetes cluster

As you've already learned, a Kubernetes cluster consists of nodes divided into two groups:

- A set of *master nodes* that host the *Control Plane* components, which are the brains of the system, since they control the entire cluster.
- A set of *worker nodes* that form the *Workload Plane*, which is where your workloads (or applications) run.

The following figure shows the two planes and the different nodes they consist of.

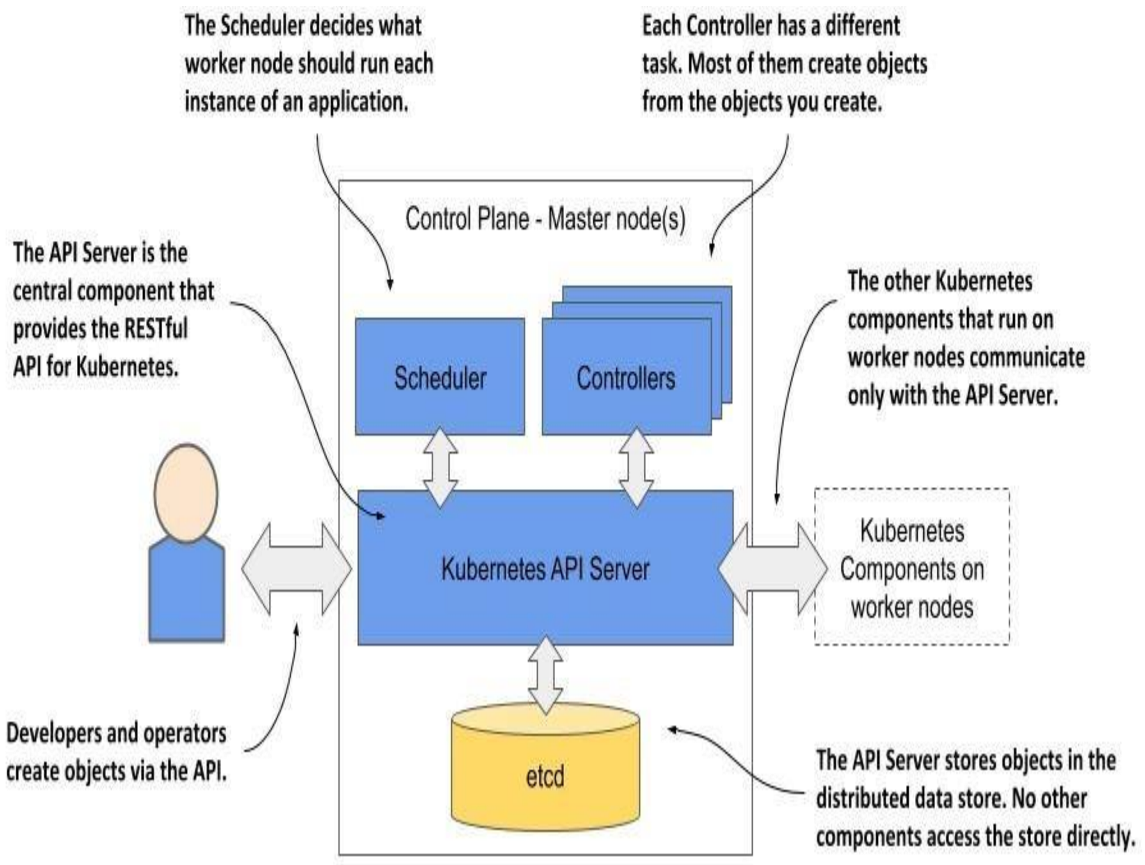
Figure 1.11 The two planes that make up a Kubernetes cluster


The Control Plane represents the brain of the cluster
The Workload Plane is the part that hosts the workloads (applications)
A cluster can have hundreds of worker nodes
Workload Plane
Control Plane
The Control Plane has one or more master nodes
These nodes run the applications
Kubernetes
Kubernetes
Each node also runs several Kubernetes components that manage the apps running on the node
Kubernetes components that control the cluster run on these nodes
Worker nodes
Master node(s)
Kubernetes components on worker nodes communicate with those running on the master, but never with each other.

The two planes, and hence the two types of nodes, run different Kubernetes components. The next two sections of the book introduce them and summarize their functions without going into details. These components will be mentioned several times in the next part of the book where I explain the fundamental concepts of Kubernetes. An in-depth look at the components and their internals follows in the third part of the book.

Control Plane components

The Control Plane is what controls the cluster. It consists of several components that run on a single master node or are replicated across multiple master nodes to ensure high availability. The Control Plane's components are shown in the following figure. **Figure 1.12 The components of the Kubernetes Control Plane**


The Scheduler decides what worker node should run each instance of an application.
Each Controller has a different task. Most of them create objects from the objects you create.
Control Plane - Master node(s)
The API Server is the central component that provides the RESTful API for Kubernetes.
The other Kubernetes components that run on worker nodes communicate only with the API Server.
Scheduler
Controllers
Kubernetes API Server
Kubernetes Components on worker nodes
etcd
Developers and operators create objects via the API.
The API Server stores objects in the distributed data store. No other components access the store directly.

These are the components and their functions:

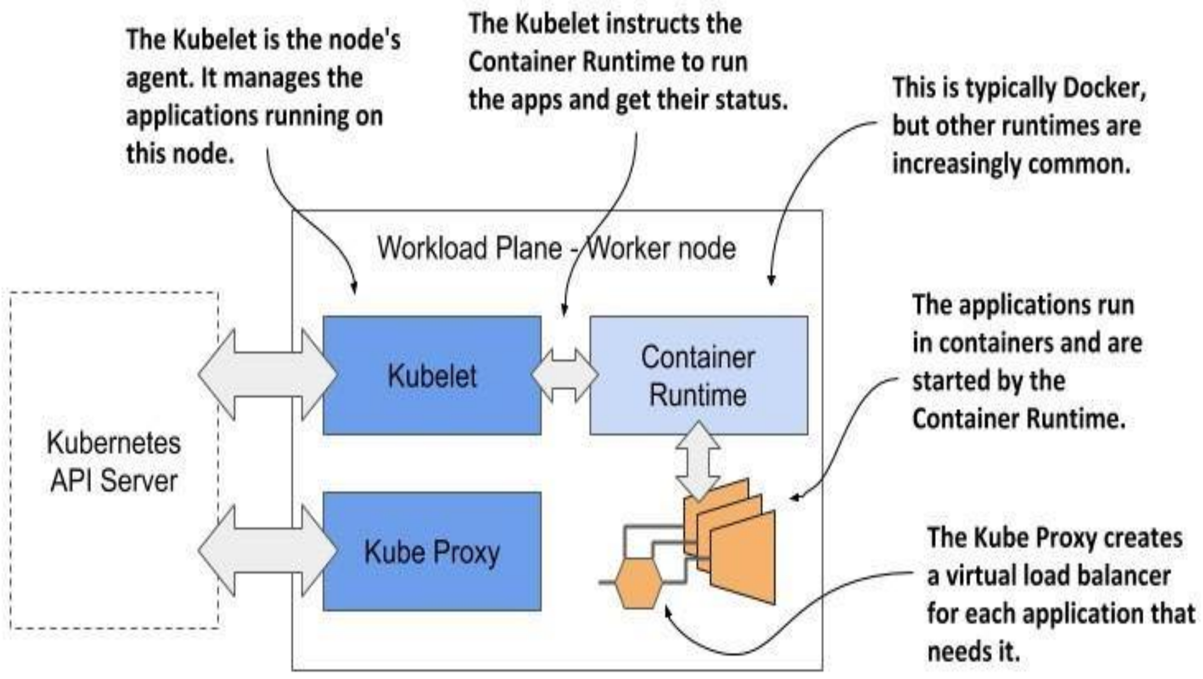
- The *Kubernetes API Server* exposes the RESTful Kubernetes API. Engineers using the cluster and other Kubernetes components create objects via this API.
- The *etcd* distributed datastore persists the objects you create through the API, since the API Server itself is stateless. The Server is the only component that talks to etcd.
- The *Scheduler* decides on which worker node each application instance should run.
- *Controllers* bring to life the objects you create through the API. Most of them simply create other objects, but some also communicate with external systems (for example, the cloud provider via its API).

The components of the Control Plane hold and control the state of the cluster, but they don't run your applications. This is done by the (worker) nodes.

Worker node components

The worker nodes are the computers on which your applications run. They form the cluster’s Workload Plane. In addition to applications, several Kubernetes components also run on these nodes. They perform the task of running, monitoring and providing connectivity between your applications. They are shown in the following figure.

Figure 1.13 The Kubernetes components that run on each node


The Kubelet is the node's agent. It manages the applications running on this node.
The Kubelet instructs the Container Runtime to run the apps and get their status.
This is typically Docker, but other runtimes are increasingly common.
Workload Plane - Worker node
Kubernetes API Server
Kubelet
Container Runtime
Kube Proxy
The applications run in containers and are started by the Container Runtime.
The Kube Proxy creates a virtual load balancer for each application that needs it.

Each node runs the following set of components:

- The *Kubelet*, an agent that talks to the API server and manages the applications running on its node. It reports the status of these applications and the node via the API.
- The *Container Runtime*, which can be Docker or any other runtime compatible with Kubernetes. It runs your applications in containers as instructed by the Kubelet.
- The *Kubernetes Service Proxy (Kube Proxy)* load-balances network traffic between applications. Its name suggests that traffic flows through it, but that’s no longer the case. You’ll learn why in chapter 14.

Add-on components

Most Kubernetes clusters also contain several other components. This includes a DNS server, network plugins, logging agents and many others. They typically run on the worker nodes but can also be configured to run on the master.

Gaining a deeper understanding of the architecture

For now, I only expect you to be vaguely familiar with the names of these components and their function, as I'll mention them many times throughout the following chapters. You'll learn snippets about them in these chapters, but I'll explain them in more detail in chapter 14.

I'm not a fan of explaining how things work until I first explain *what* something does and teach you how to use it. It's like learning to drive. You don't want to know what's under the hood. At first, you just want to learn how to get from point A to B. Only then will you be interested in how the car makes this possible. Knowing what's under the hood may one day help you get your car moving again after it has broken down and you are stranded on the side of the road. I hate to say it, but you'll have many moments like this when dealing with Kubernetes due to its sheer complexity.

1.2.4 How Kubernetes runs an application

With a general overview of the components that make up Kubernetes, I can finally explain how to deploy an application in Kubernetes.

Defining your application

Everything in Kubernetes is represented by an object. You create and retrieve these objects via the Kubernetes API. Your application consists of several types of these objects - one type represents the application deployment as a whole, another represents a running instance of your application, another represents the service provided by a set of these instances and allows reaching them at a single IP address, and there are many others.

All these types are explained in detail in the second part of the book. At the moment, it’s enough to know that you define your application through several types of objects. These objects are usually defined in one or more manifest files in either YAML or JSON format.

Definition

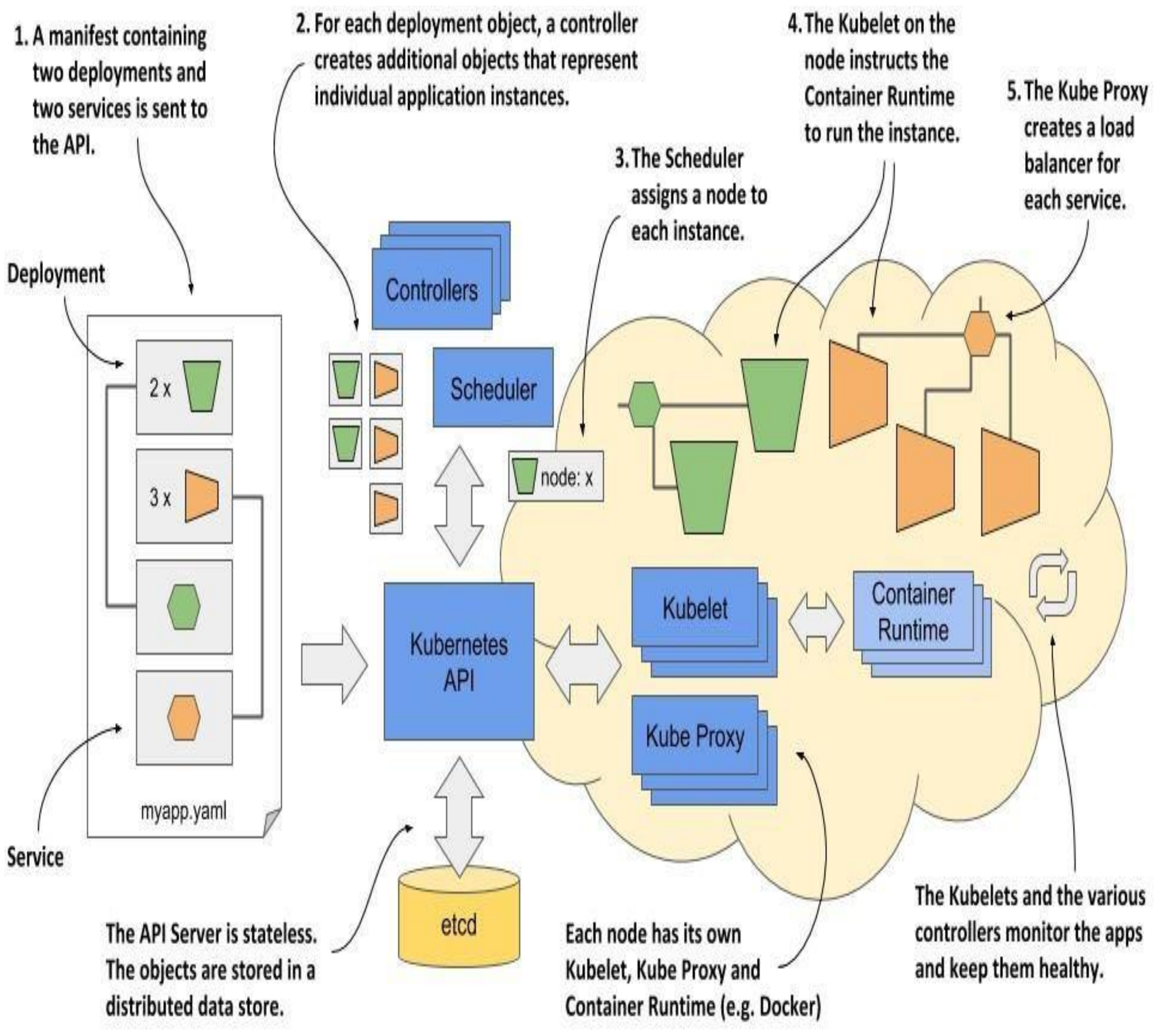
YAML was initially said to mean “Yet Another Markup Language”, but it was latter changed to the recursive acronym “YAML Ain’t Markup Language”. It’s one of the ways to serialize an object into a humanreadable text file.

Definition

JSON is short for JavaScript Object Notation. It’s a different way of serializing an object, but more suitable for exchanging data between applications.

The following figure shows an example of deploying an application by creating a manifest with two deployments exposed using two services.

Figure 1.14 Deploying an application to Kubernetes


1. A manifest containing two deployments and two services is sent to the API.
2. For each deployment object, a controller creates additional objects that represent individual application instances.
3. The Scheduler assigns a node to each instance.
4. The Kubelet on the node instructs the Container Runtime to run the instance.
5. The Kube Proxy creates a load balancer for each service.
Deployment
2 x
3 x
myapp.yaml
Service
Controllers
Scheduler
node: x
Kubernetes API
etcd
Kubelet
Kube Proxy
Container Runtime
The API Server is stateless. The objects are stored in a distributed data store.
Each node has its own Kubelet, Kube Proxy and Container Runtime (e.g. Docker)
The Kubelets and the various controllers monitor the apps and keep them healthy.

These actions take place when you deploy the application:

1. You submit the application manifest to the Kubernetes API. The API Server writes the objects defined in the manifest to etcd.
2. A controller notices the newly created objects and creates several new objects - one for each application instance.
3. The Scheduler assigns a node to each instance.
4. The Kubelet notices that an instance is assigned to the Kubelet's node. It runs the application instance via the Container Runtime.
5. The Kube Proxy notices that the application instances are ready to accept connections from clients and configures a load balancer for them.
6. The Kubelets and the Controllers monitor the system and keep the applications running.

The procedure is explained in more detail in the following sections, but the complete explanation is given in chapter 14, after you have familiarized yourself with all the objects and controllers involved.

Submitting the application to the API

After you’ve created your YAML or JSON file(s), you submit the file to the API, usually via the Kubernetes command-line tool called *kubectl*.

Note

Kubectl is pronounced *kube-control*, but the softer souls in the community prefer to call it *kube-cuddle*. Some refer to it as *kube-C-T-L*.

Kubectl splits the file into individual objects and creates each of them by sending an HTTP PUT or POST request to the API, as is usually the case with RESTful APIs. The API Server validates the objects and stores them in the etcd datastore. In addition, it notifies all interested components that these objects have been created. Controllers, which are explained next, are one of these components.

About the controllers

Most object types have an associated controller. A controller is interested in a particular object type. It waits for the API server to notify it that a new object has been created, and then performs operations to bring that object to life. Typically, the controller just creates other objects via the same Kubernetes API. For example, the controller responsible for application deployments creates one or more objects that represent individual instances of the application. The number of objects created by the controller depends on the number of replicas specified in the application deployment object.

About the Scheduler

The scheduler is a special type of controller, whose only task is to schedule application instances onto worker nodes. It selects the best worker node for each new application instance object and assigns it to the instance - by modifying the object via the API.

About the Kubelet and the Container Runtime

The Kubelet that runs on each worker node is also a type of controller. Its task is to wait for application instances to be assigned to the node on which it is located and run the application. This is done by instructing the Container Runtime to start the application's container.

About the Kube Proxy

Because an application deployment can consist of multiple application instances, a load balancer is required to expose them at a single IP address. The Kube Proxy, another controller running alongside the Kubelet, is responsible for setting up the load balancer.

Keeping the applications healthy

Once the application is up and running, the Kubelet keeps the application healthy by restarting it when it terminates. It also reports the status of the application by updating the object that represents the application instance. The other controllers monitor these objects and ensure that applications are moved to healthy nodes if their nodes fail.

You're now roughly familiar with the architecture and functionality of Kubernetes. You don't need to understand or remember all the details at this moment, because internalizing this information will be easier when you learn about each individual object types and the controllers that bring them to life in the second part of the book.

1.3 Introducing Kubernetes into your organization

To close this chapter, let's see what options are available to you if you decide to introduce Kubernetes in your own IT environment.

1.3.1 Running Kubernetes on-premises and in the cloud

If you want to run your applications on Kubernetes, you have to decide whether you want to run them locally, in your organization's own infrastructure (on-premises) or with one of the major cloud providers, or perhaps both - in a hybrid cloud solution.

Running Kubernetes on-premises

Running Kubernetes on your own infrastructure may be your only option if regulations require you to run applications on site. This usually means that you'll have to manage Kubernetes yourself, but we'll come to that later.

Kubernetes can run directly on your bare-metal machines or in virtual machines running in your data center. In either case, you won't be able to scale your cluster as easily as when you run it in virtual machines provided by a cloud provider.

Deploying Kubernetes in the cloud

If you have no on-premises infrastructure, you have no choice but to run Kubernetes in the cloud. This has the advantage that you can scale your cluster at any time at short notice if required. As mentioned earlier, Kubernetes itself can ask the cloud provider to provision additional virtual machines when the current size of the cluster is no longer sufficient to run all the applications you want to deploy.

When the number of workloads decreases and some worker nodes are left without running workloads, Kubernetes can ask the cloud provider to destroy the virtual machines of these nodes to reduce your operational costs. This elasticity of the cluster is certainly one of the main benefits of running Kubernetes in the cloud.

Using a hybrid cloud solution

A more complex option is to run Kubernetes on-premises, but also allow it to spill over into the cloud. It's possible to configure Kubernetes to provision additional nodes in the cloud if you exceed the capacity of your own data center. This way, you get the best of both worlds. Most of the time, your applications run locally without the cost of virtual machine rental, but in short periods of peak load that may occur only a few times a year, your applications can handle the extra load by using the additional resources in the cloud.

If your use-case requires it, you can also run a Kubernetes cluster across multiple cloud providers or a combination of any of the options mentioned. This can be done using a single control plane or one control plane in each location.

1.3.2 To manage or not to manage Kubernetes yourself

If you are considering introducing Kubernetes in your organization, the most important question you need to answer is whether you'll manage Kubernetes yourself or use a Kubernetes-as-a-Service type offering where someone else manages it for you.

Managing Kubernetes yourself

If you already run applications on-premises and have enough hardware to run a production-ready Kubernetes cluster, your first instinct is probably to deploy and manage it yourself. If you ask anyone in the Kubernetes community if this is a good idea, you'll usually get a very definite "no".

Figure 1.14 was a very simplified representation of what happens in a Kubernetes cluster when you deploy an application. Even that figure should have scared you. Kubernetes brings with it an enormous amount of additional complexity. Anyone who wants to run a Kubernetes cluster must be intimately familiar with its inner workings.

The management of production-ready Kubernetes clusters is a multibillion-dollar industry. Before you decide to manage one yourself, it's essential that you consult with engineers who have already done it to learn about the issues most teams run into. If you don't, you may be setting yourself up for failure. On the other hand, trying out Kubernetes for nonproduction use-cases or using a managed Kubernetes cluster is much less problematic.

Using a managed Kubernetes cluster in the cloud

Using Kubernetes is ten times easier than managing it. Most major cloud providers now offer Kubernetes-as-a-Service. They take care of managing Kubernetes and its components while you simply use the Kubernetes API like any of the other APIs the cloud provider offers.

The top managed Kubernetes offerings include the following:

- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (EKS)
- IBM Cloud Kubernetes Service
- Red Hat OpenShift Online and Dedicated
- VMware Cloud PKS
- Alibaba Cloud Container Service for Kubernetes (ACK)

The first half of this book focuses on just using Kubernetes. You'll run the exercises in a local development cluster and on a managed GKE cluster, as I find it's the easiest to use and offers the best user experience. The second part of the book gives you a solid foundation for managing Kubernetes, but to truly master it, you'll need to gain additional experience.

1.3.3 Using vanilla or extended Kubernetes

The final question is whether to use a vanilla open-source version of Kubernetes or an extended, enterprise-quality Kubernetes product.

Using a vanilla version of Kubernetes

The open-source version of Kubernetes is maintained by the community and represents the cutting edge of Kubernetes development. This also means that it may not be as stable as the other options. It may also lack good security defaults. Deploying the vanilla version requires a lot of fine tuning to set everything up for production use.

Using enterprise-grade Kubernetes distributions

A better option for using Kubernetes in production is to use an enterprisequality Kubernetes distribution such as OpenShift or Rancher. In addition to the increased security and performance provided by better defaults, they offer additional object types in addition to those provided in the upstream Kubernetes API. For example, vanilla Kubernetes does not contain object types that represent cluster users, whereas commercial distributions do. They also provide additional software tools for deploying and managing well-known third-party applications on Kubernetes.

Of course, extending and hardening Kubernetes takes time, so these commercial Kubernetes distributions usually lag one or two versions behind the upstream version of Kubernetes. It’s not as bad as it sounds. The benefits usually outweigh the disadvantages.

1.3.4 Should you even use Kubernetes?

I hope this chapter has made you excited about Kubernetes and you can’t wait to squeeze it into your IT stack. But to close this chapter properly, we need to say a word or two about when introducing Kubernetes is not a good idea.

Do your workloads require automated management?

The first thing you need to be honest about is whether you need to automate the management of your applications at all. If your application is a large monolith, you definitely don’t need Kubernetes.

Even if you deploy microservices, using Kubernetes may not be the best option, especially if the number of your microservices is very small. It’s difficult to provide an exact number when the scales tip over, since other factors also influence the decision. But if your system consists of less than five microservices, throwing Kubernetes into the mix is probably not a good idea. If your system has more than twenty microservices, you will most likely benefit from the integration of Kubernetes. If the number of your microservices falls somewhere in between, other factors, such as the ones described next, should be considered.

Can you afford to invest your engineers’ time into learning Kubernetes?

Kubernetes is designed to allow applications to run without them knowing that they are running in Kubernetes. While the applications themselves don't need to be modified to run in Kubernetes, development engineers will inevitably spend a lot of time learning how to use Kubernetes, even though the operators are the only ones that actually need that knowledge.

It would be hard to tell your teams that you're switching to Kubernetes and expect only the operations team to start exploring it. Developers like shiny new things. At the time of writing, Kubernetes is still a very shiny thing.

Are you prepared for increased costs in the interim?

While Kubernetes reduces long-term operational costs, introducing Kubernetes in your organization initially involves increased costs for training, hiring new engineers, building and purchasing new tools and possibly additional hardware. Kubernetes requires additional computing resources in addition to the resources that the applications use.

Don't believe the hype

Although Kubernetes has been around for several years at the time of writing this book, I can't say that the hype phase is over. The initial excitement has just begun to calm down, but many engineers may still be unable to make rational decisions about whether the integration of Kubernetes is as necessary as it seems.

1.4 Summary

In this introductory chapter, you've learned that:

- Kubernetes is Greek for helmsman. As a ship's captain oversees the ship while the helmsman steers it, you oversee your computer cluster, while Kubernetes performs the day-to-day management tasks.
- Kubernetes is pronounced *koo-ber-netties*. Kubectl, the Kubernetes command-line tool, is pronounced *kube-control*.
- Kubernetes is an open-source project built upon Google's vast experience in running applications on a global scale. Thousands of individuals now contribute to it.

- Kubernetes uses a declarative model to describe application deployments. After you provide a description of your application to Kubernetes, it brings it to life.
- Kubernetes is like an operating system for the cluster. It abstracts the infrastructure and presents all computers in a data center as one large, contiguous deployment area.
- Microservice-based applications are more difficult to manage than monolithic applications. The more microservices you have, the more you need to automate their management with a system like Kubernetes.
- Kubernetes helps both development and operations teams to do what they do best. It frees them from mundane tasks and introduces a standard way of deploying applications both on-premises and in any cloud.
- Using Kubernetes allows developers to deploy applications without the help of system administrators. It reduces operational costs through better utilization of existing hardware, automatically adjusts your system to load fluctuations, and heals itself and the applications running on it.
- A Kubernetes cluster consists of master and worker nodes. The master nodes run the *Control Plane*, which controls the entire cluster, while the worker nodes run the deployed applications or workloads, and therefore represent the *Workload Plane*.
- Using Kubernetes is simple, but managing it is hard. An inexperienced team should use a Kubernetes-as-a-Service offering instead of deploying Kubernetes by itself.

So far, you've only observed the ship from the pier. It's time to come aboard. But before you leave the docks, you should inspect the shipping containers it's carrying. You'll do this next.

2 Understanding containers

This chapter covers

- Understanding what a container is
- Differences between containers and virtual machines
- Creating, running, and sharing a container image with Docker
- Linux kernel features that make containers possible

Kubernetes primarily manages applications that run in containers - so before you start exploring Kubernetes, you need to have a good understanding of what a container is. This chapter explains the basics of Linux containers that a typical Kubernetes user needs to know.

2.1 Introducing containers

In Chapter 1 you learned how different microservices running in the same operating system may require different, potentially conflicting versions of dynamically linked libraries or have different environment requirements.

When a system consists of a small number of applications, it’s okay to assign a dedicated virtual machine to each application and run each in its own operating system. But as the microservices become smaller and their numbers start to grow, you may not be able to afford to give each one its own VM if you want to keep your hardware costs low and not waste resources.

It’s not just a matter of wasting hardware resources - each VM typically needs to be individually configured and managed, which means that running higher numbers of VMs also results in higher staffing requirements and the need for a better, often more complicated automation system. Due to the shift to microservice architectures, where systems consist of hundreds of deployed application instances, an alternative to VMs was needed. Containers are that alternative.

2.1.1 Comparing containers to virtual machines

Instead of using virtual machines to isolate the environments of individual microservices (or software processes in general), most development and operations teams now prefer to use containers. They allow you to run multiple services on the same host computer, while keeping them isolated from each other. Like VMs, but with much less overhead.

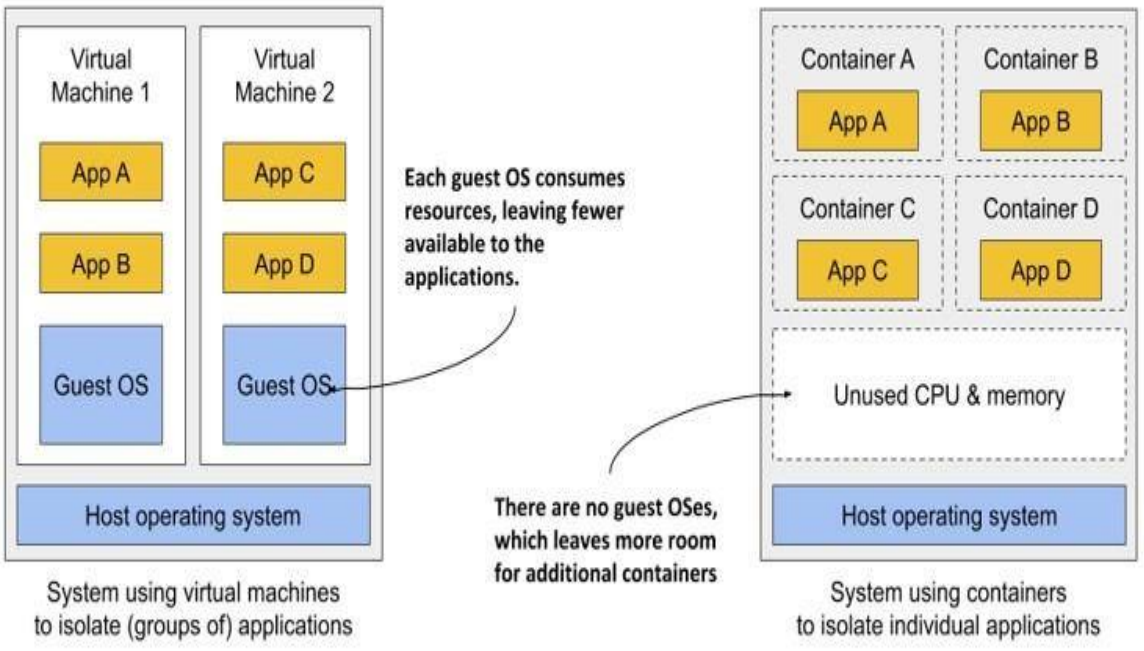
Unlike VMs, which each run a separate operating system with several system processes, a process running in a container runs within the existing host operating system. Because there is only one operating system, no duplicate system processes exist. Although all the application processes run in the same operating system, their environments are isolated, though not as well as when you run them in separate VMs. To the process in the container, this isolation makes it look like no other processes exist on the computer. You'll learn how this is possible in the next few sections, but first let's dive deeper into the differences between containers and virtual machines.

Comparing the overhead of containers and virtual machines

Compared to VMs, containers are much lighter, because they don't require a separate resource pool or any additional OS-level processes. While each VM usually runs its own set of system processes, which requires additional computing resources in addition to those consumed by the user application's own process, a container is nothing more than an isolated process running in the existing host OS that consumes only the resources the app consumes. They have virtually no overhead.

Figure 2.1 shows two bare metal computers, one running two virtual machines, and the other running containers instead. The latter has space for additional containers, as it runs only one operating system, while the first runs three – one host and two guest OSes.

Figure 2.1 Using VMs to isolate groups of applications vs. isolating individual apps with containers


Virtual Machine 1
Virtual Machine 2
App A
App C
App B
App D
Guest OS
Guest OS
Host operating system
System using virtual machines to isolate (groups of) applications
Each guest OS consumes resources, leaving fewer available to the applications.
There are no guest OSes, which leaves more room for additional containers
Container A
Container B
App A
App B
Container C
Container D
App C
App D
Unused CPU & memory
Host operating system
System using containers to isolate individual applications

Because of the resource overhead of VMs, you often group multiple applications into each VM. You may not be able to afford dedicating a whole VM to each app. But containers introduce no overhead, which means you can afford to create a separate container for each application. In fact, you should never run multiple applications in the same container, as this makes managing the processes in the container much more difficult. Moreover, all existing software dealing with containers, including Kubernetes itself, is designed under the premise that there’s only one application in a container. But as you’ll learn in the next chapter, Kubernetes provides a way to run related applications together, yet still keep them in separate containers.

Comparing the start-up time of containers and virtual machines

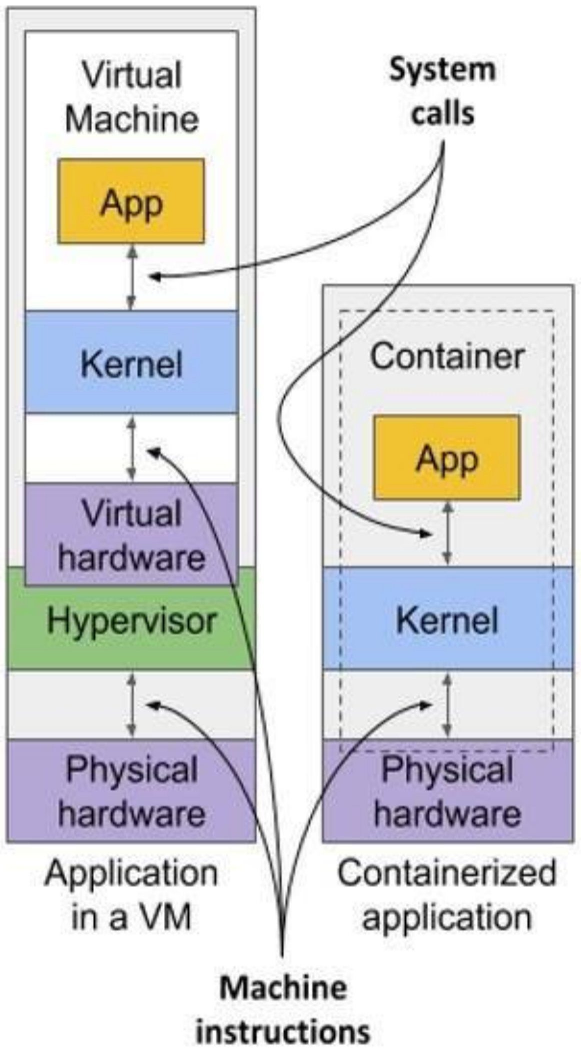
In addition to the lower runtime overhead, containers also start the application faster, because only the application process itself needs to be started. No additional system processes need to be started first, as is the case when booting up a new virtual machine.

Comparing the isolation of containers and virtual machines

You’ll agree that containers are clearly better when it comes to the use of resources, but there’s also a disadvantage. When you run applications in virtual machines, each VM runs its own operating system and kernel.

Underneath those VMs is the hypervisor (and possibly an additional operating system), which splits the physical hardware resources into smaller sets of virtual resources that the operating system in each VM can use. As figure 2.2 shows, applications running in these VMs make system calls (*sys-calls*) to the guest OS kernel in the VM, and the machine instructions that the kernel then executes on the virtual CPUs are then forwarded to the host’s physical CPU via the hypervisor.

Figure 2.2 How apps use the hardware when running in a VM vs. in a container



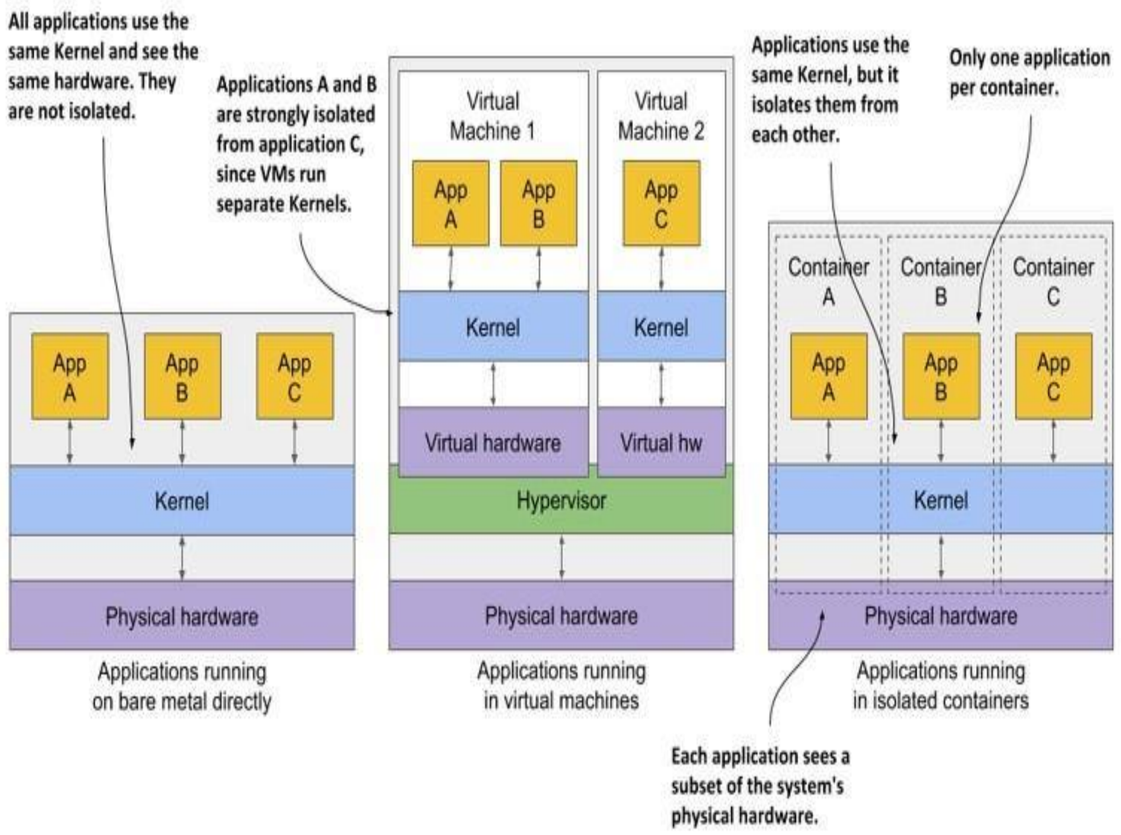
Note

Two types of hypervisors exist. Type 1 hypervisors don’t require running a host OS, while type 2 hypervisors do.

Containers, on the other hand, all make system calls on the single kernel running in the host OS. This single kernel is the only one that executes instructions on the host’s CPU. The CPU doesn’t need to handle any kind of virtualization the way it does with VMs.

Examine the following figure to see the difference between running three applications on bare metal, running them in two separate virtual machines, or running them in three containers.

Figure 2.3 The difference between running applications on bare metal, in virtual machines, and in containers


All applications use the same Kernel and see the same hardware. They are not isolated.
Applications A and B are strongly isolated from application C, since VMs run separate Kernels.
Applications use the same Kernel, but it isolates them from each other.
Only one application per container.
App A
App B
App C
Kernel
Physical hardware
Applications running on bare metal directly
Virtual Machine 1
Virtual Machine 2
Virtual hardware
Virtual hw
Hypervisor
Applications running in virtual machines
Container A
Container B
Container C
Applications running in isolated containers
Each application sees a subset of the system's physical hardware.

In the first case, all three applications use the same kernel and aren’t isolated at all. In the second case, applications A and B run in the same VM and thus share the kernel, while application C is completely isolated from the other two, since it uses its own kernel. It only shares the hardware with the first two.

The third case shows the same three applications running in containers. Although they all use the same kernel, they are isolated from each other and completely unaware of the others' existence. The isolation is provided by the kernel itself. Each application sees only a part of the physical hardware and sees itself as the only process running in the OS, although they all run in the same OS.

Understanding the security-implications of container isolation

The main advantage of using virtual machines over containers is the complete isolation they provide, since each VM has its own Linux kernel, while containers all use the same kernel. This can clearly pose a security risk. If there's a bug in the kernel, an application in one container might use it to read the memory of applications in other containers. If the apps run in different VMs and therefore share only the hardware, the probability of such attacks is much lower. Of course, complete isolation is only achieved by running applications on separate physical machines.

Additionally, containers share memory space, whereas each VM uses its own chunk of memory. Therefore, if you don't limit the amount of memory that a container can use, this could cause other containers to run out of memory or cause their data to be swapped out to disk.

Note

This can't happen in Kubernetes, because it requires that swap is disabled on all the nodes.

Understanding what enables containers and what enables virtual machines

While virtual machines are enabled through virtualization support in the CPU and by virtualization software on the host, containers are enabled by the Linux kernel itself. You'll learn about container technologies later when you can try them out for yourself. You'll need to have Docker installed for that, so let's learn how it fits into the container story.

2.1.2 Introducing the Docker container platform

While container technologies have existed for a long time, they only became widely known with the rise of Docker. Docker was the first container system that made them easily portable across different computers. It simplified the process of packaging up the application and all its libraries and other dependencies - even the entire OS file system - into a simple, portable package that can be used to deploy the application on any computer running Docker.

Introducing containers, images and registries

Docker is a platform for packaging, distributing and running applications. As mentioned earlier, it allows you to package your application along with its entire environment. This can be just a few dynamically linked libraries required by the app, or all the files that are usually shipped with an operating system. Docker allows you to distribute this package via a public repository to any other Docker-enabled computer.

Figure 2.4 The three main Docker concepts are images, registries and containers

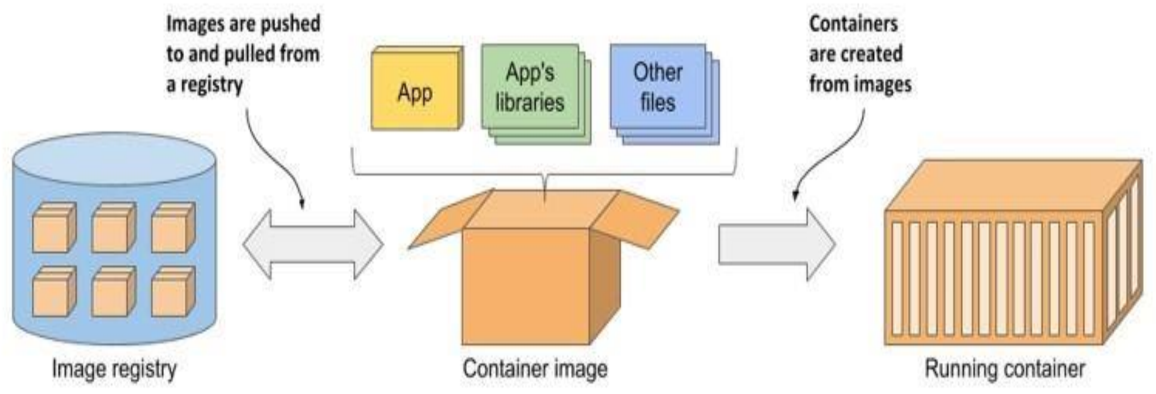

Images are pushed to and pulled from a registry
App
App's libraries
Other files
Containers are created from images
Image registry
Container image
Running container

Figure 2.4 shows three main Docker concepts that appear in the process I've just described. Here's what each of them is:

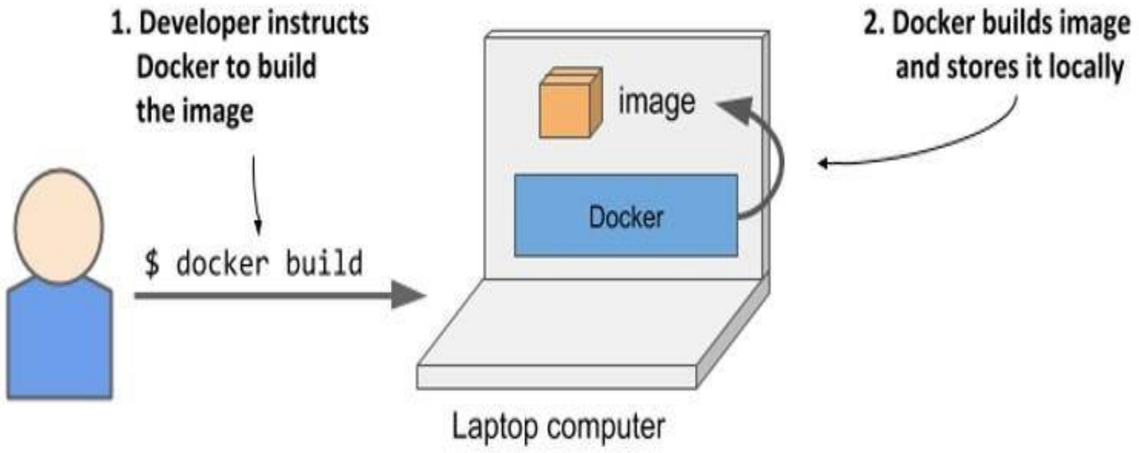
- *Images*—A container image is something you package your application and its environment into. Like a zip file or a tarball. It contains the whole filesystem that the application will use and additional metadata, such as the path to the executable file to run when the image is executed, the ports the application listens on, and other information about the image.

- *Registries*—A registry is a repository of container images that enables the exchange of images between different people and computers. After you build your image, you can either run it on the same computer, or *push* (upload) the image to a registry and then *pull* (download) it to another computer. Certain registries are public, allowing anyone to pull images from it, while others are private and only accessible to individuals, organizations or computers that have the required authentication credentials.
- *Containers*—A container is instantiated from a container image. A running container is a normal process running in the host operating system, but its environment is isolated from that of the host and the environments of other processes. The file system of the container originates from the container image, but additional file systems can also be mounted into the container. A container is usually resourcerestricted, meaning it can only access and use the amount of resources such as CPU and memory that have been allocated to it.

Building, distributing, and running a container image

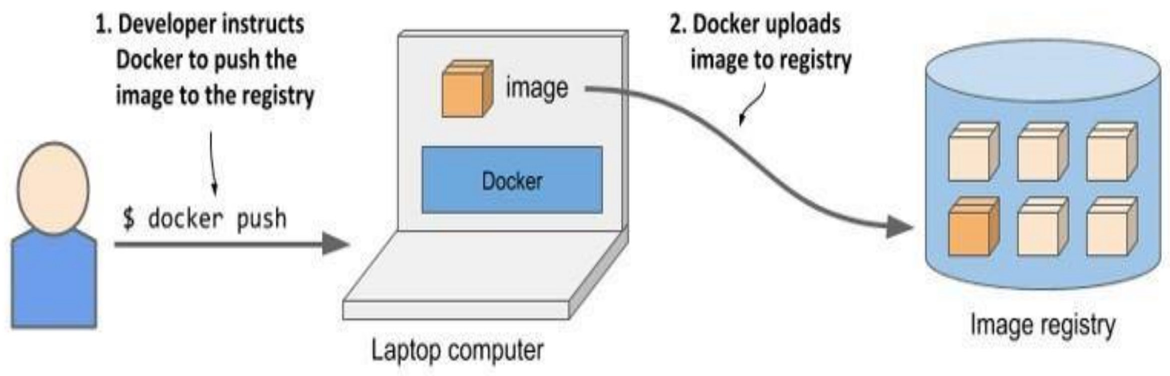
To understand how containers, images and registries relate to each other, let's look at how to build a container image, distribute it through a registry and create a running container from the image. These three processes are shown in figures 2.5 to 2.7.

Figure 2.5 Building a container image


1. Developer instructs Docker to build the image
2. Docker builds image and stores it locally
image
Docker
$ docker build
Laptop computer

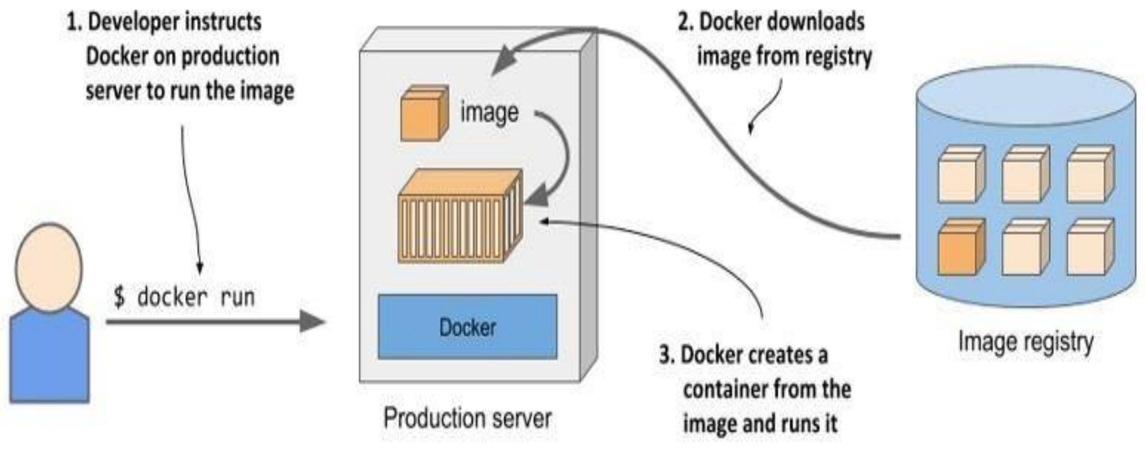
As shown in figure 2.5, the developer first builds an image, and then pushes it to a registry, as shown in figure 2.6. The image is now available to anyone who can access the registry.

Figure 2.6 Uploading a container image to a registry


1. Developer instructs Docker to push the image to the registry
$ docker push
image
Docker
Laptop computer
2. Docker uploads image to registry
Image registry

As the next figure shows, another person can now pull the image to any other computer running Docker and run it. Docker creates an isolated container based on the image and invokes the executable file specified in the image.

Figure 2.7 Running a container on a different computer


1. Developer instructs Docker on production server to run the image
$ docker run
image
Docker
Production server
2. Docker downloads image from registry
3. Docker creates a container from the image and runs it
Image registry

Running the application on any computer is made possible by the fact that the environment of the application is decoupled from the environment of the host.

Understanding the environment that the application sees

When you run an application in a container, it sees exactly the file system content you bundled into the container image, as well as any additional file systems you mount into the container. The application sees the same files whether it's running on your laptop or a full-fledged production server, even if the production server uses a completely different Linux distribution. The application typically has no access to the files in the host's operating system, so it doesn't matter if the server has a completely different set of installed libraries than your development computer.

For example, if you package your application with the files of the entire Red Hat Enterprise Linux (RHEL) operating system and then run it, the application will think it's running inside RHEL, whether you run it on your Fedora-based or a Debian-based computer. The Linux distribution installed on the host is irrelevant. The only thing that might be important is the kernel version and the kernel modules it loads. Later, I'll explain why.

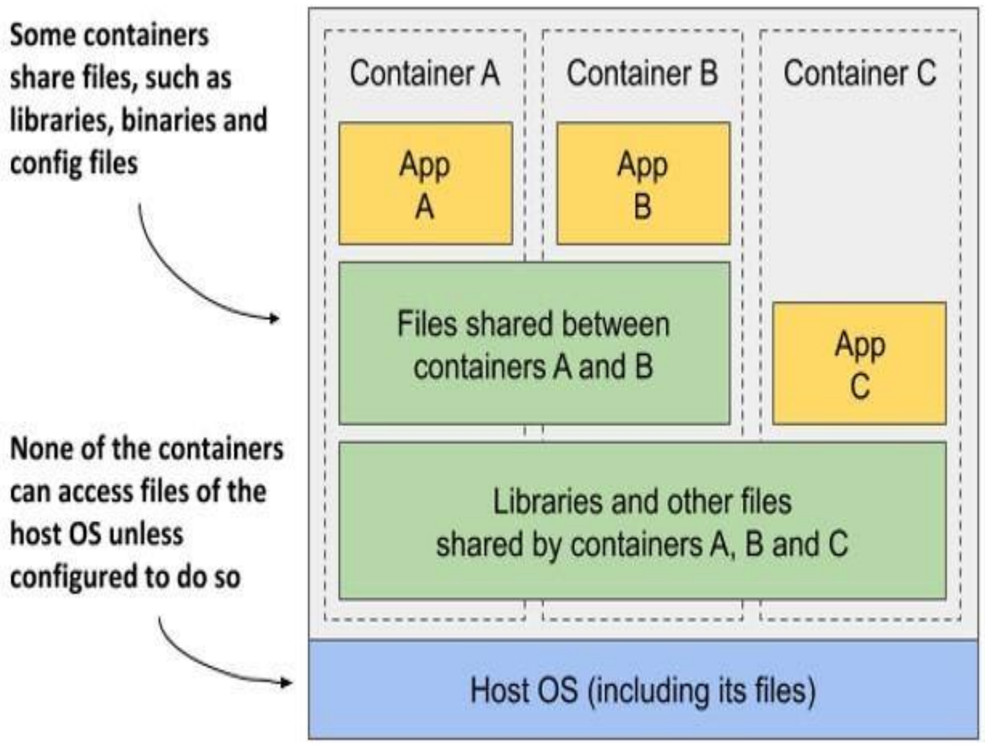
This is similar to creating a VM image by creating a new VM, installing an operating system and your app in it, and then distributing the whole VM image so that other people can run it on different hosts. Docker achieves the same effect, but instead of using VMs for app isolation, it uses Linux container technologies to achieve (almost) the same level of isolation.

Understanding image layers

Unlike virtual machine images, which are big blobs of the entire filesystem required by the operating system installed in the VM, container images consist of layers that are usually much smaller. These layers can be shared and reused across multiple images. This means that only certain layers of an image need to be downloaded if the rest were already downloaded to the host as part of another image containing the same layers.

Layers make image distribution very efficient but also help to reduce the storage footprint of images. Docker stores each layer only once. As you can see in the following figure, two containers created from two images that contain the same layers use the same files.

Figure 2.8 Containers can share image layers


Some containers share files, such as libraries, binaries and config files
None of the containers can access files of the host OS unless configured to do so
Container A
Container B
Container C
App A
App B
Files shared between containers A and B
App C
Libraries and other files shared by containers A, B and C
Host OS (including its files)

The figure shows that containers A and B share an image layer, which means that applications A and B read some of the same files. In addition, they also share the underlying layer with container C. But if all three containers have access to the same files, how can they be completely isolated from each other? Are changes that application A makes to a file stored in the shared layer not visible to application B? They aren't. Here's why.

The filesystems are isolated by the Copy-on-Write (CoW) mechanism. The filesystem of a container consists of read-only layers from the container image and an additional read/write layer stacked on top. When an application running in container A changes a file in one of the read-only layers, the entire file is copied into the container's read/write layer and the file contents are changed there. Since each container has its own writable layer, changes to shared files are not visible in any other container.

When you delete a file, it is only marked as deleted in the read/write layer, but it's still present in one or more of the layers below. What follows is that deleting files never reduces the size of the image.

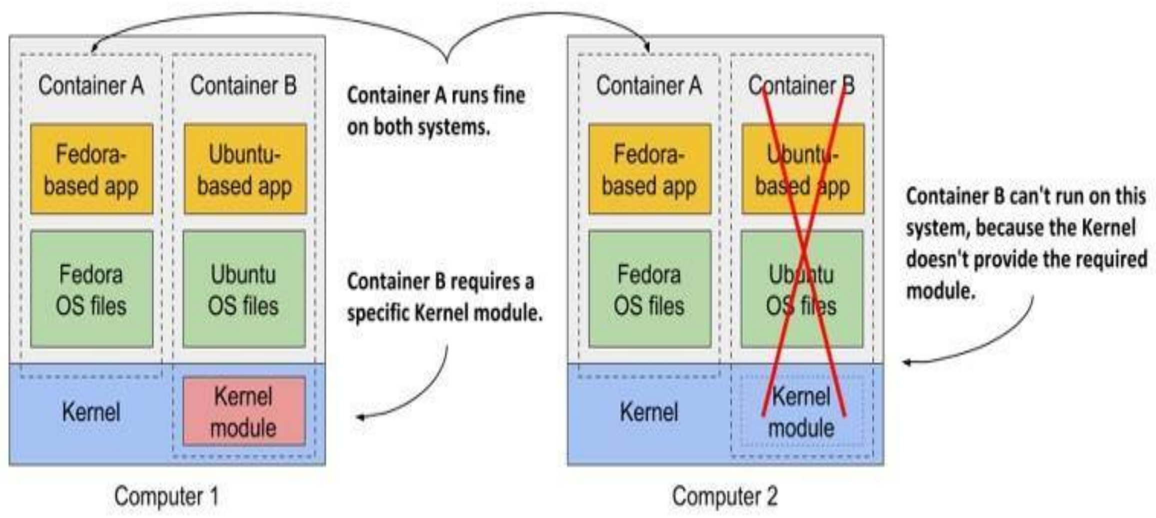
WARNING

Even seemingly harmless operations such as changing permissions or ownership of a file result in a new copy of the entire file being created in the read/write layer. If you perform this type of operation on a large file or many files, the image size may swell significantly.

Understanding the portability limitations of container images

In theory, a Docker-based container image can be run on any Linux computer running Docker, but one small caveat exists, because containers don’t have their own kernel. If a containerized application requires a particular kernel version, it may not work on every computer. If a computer is running a different version of the Linux kernel or doesn’t load the required kernel modules, the app can’t run on it. This scenario is illustrated in the following figure.

Figure 2.9 If a container requires specific kernel features or modules, it may not work everywhere



Container B requires a specific kernel module to run properly. This module is loaded in the kernel in the first computer, but not in the second. You can run the container image on the second computer, but it will break when it tries to use the missing module.

And it’s not just about the kernel and its modules. It should also be clear that a containerized app built for a specific hardware architecture can only run on computers with the same architecture. You can’t put an application compiled for the x86 CPU architecture into a container and expect to run it on an ARM-based computer just because Docker is available there. For this you would need a VM to emulate the x86 architecture.

2.1.3 Installing Docker and running a Hello World container

You should now have a basic understanding of what a container is, so let's use Docker to run one. You'll install Docker and run a Hello World container.

Installing Docker

Ideally, you'll install Docker directly on a Linux computer, so you won't have to deal with the additional complexity of running containers inside a VM running within your host OS. But, if you're using macOS or Windows and don't know how to set up a Linux VM, the Docker Desktop application will set it up for you. The Docker command-line (CLI) tool that you'll use to run containers will be installed in your host OS, but the Docker daemon will run inside the VM, as will all the containers it creates.

The Docker Platform consists of many components, but you only need to install Docker Engine to run containers. If you use macOS or Windows, install Docker Desktop. For details, follow the instructions at http://docs.docker.com/install.

Note

Docker Desktop for Windows can run either Windows or Linux containers. Make sure that you configure it to use Linux containers, as all the examples in this book assume that's the case.

Running a Hello World container

After the installation is complete, you use the `docker` CLI tool to run Docker commands. Let's try pulling and running an existing image from Docker Hub, the public image registry that contains ready-to-use container images for many well-known software packages. One of them is the `busybox` image, which you'll use to run a simple `echo "Hello world"` command in your first container.