Kubernetes IN ACTION

Marko Lukša

MANNING

Kubernetes resources covered in the book

	Resource (abbr.) [API version]	Description	Section
	Namespace* (ns) [v1]	Enables organizing resources into non-overlapping groups (for example, per tenant)	3.7
Deploying workloads	**Pod** (po) [v1]	The basic deployable unit containing one or more processes in co-located containers	3.1
	ReplicaSet (rs) [apps/v1beta2**]	Keeps one or more pod replicas running	4.3
	ReplicationController (rc) [v1]	The older, less-powerful equivalent of a ReplicaSet	4.2
	Job [batch/v1]	Runs pods that perform a completable task	4.5
	CronJob [batch/v1beta1]	Runs a scheduled job once or periodically	4.6
	DaemonSet (ds) [apps/v1beta2**]	Runs one pod replica per node (on all nodes or only on those matching a node selector)	4.4
	StatefulSet (sts) [apps/v1beta1**]	Runs stateful pods with a stable identity	10.2
	Deployment (deploy) [apps/v1beta1**]	Declarative deployment and updates of pods	9.3
Services	**Service** (svc) [v1]	Exposes one or more pods at a single and stable IP address and port pair	5.1
	Endpoints (ep) [v1]	Defines which pods (or other servers) are exposed through a service	5.2.1
	Ingress (ing) [extensions/v1beta1]	Exposes one or more services to external clients through a single externally reachable IP address	5.4
Config	**ConfigMap** (cm) [v1]	A key-value map for storing non-sensitive config options for apps and exposing it to them	7.4
	Secret [v1]	Like a ConfigMap, but for sensitive data	7.5
Storage	**PersistentVolume*** (pv) [v1]	Points to persistent storage that can be mounted into a pod through a PersistentVolumeClaim	6.5
	PersistentVolumeClaim (pvc) [v1]	A request for and claim to a PersistentVolume	6.5
	StorageClass* (sc) [storage.k8s.io/v1]	Defines the type of dynamically-provisioned storage claimable in a PersistentVolumeClaim	6.6

* Cluster-level resource (not namespaced)

** Also in other API versions; listed version is the one used in this book

(continues on inside back cover)

Kubernetes in Action

Kubernetes in Action

MARKO LUKŠA

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Elesha Hyde
Review editor: Aleksandar Dragosavljević
Technical development editor: Jeanne Boyarsky
Project editor: Kevin Sullivan
Copyeditor: Katie Petito
Proofreader: Melody Dolab
Technical proofreader: Antonio Magnaghi
Illustrator: Chuck Larson
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617293726
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – EBM – 22 21 20 19 18 17

To my parents,
who have always put their children's needs above their own

brief contents

PART 3 BEYOND THE BASICS

contents

PART 3 BEYOND THE BASICS

preface

After working at Red Hat for a few years, in late 2014 I was assigned to a newly-established team called Cloud Enablement. Our task was to bring the company's range of middleware products to the OpenShift Container Platform, which was then being developed on top of Kubernetes. At that time, Kubernetes was still in its infancy—version 1.0 hadn't even been released yet.

Our team had to get to know the ins and outs of Kubernetes quickly to set a proper direction for our software and take advantage of everything Kubernetes had to offer. When faced with a problem, it was hard for us to tell if we were doing things wrong or merely hitting one of the early Kubernetes bugs.

Both Kubernetes and my understanding of it have come a long way since then. When I first started using it, most people hadn't even heard of Kubernetes. Now, virtually every software engineer knows about it, and it has become one of the fastest-growing and most-widely-adopted ways of running applications in both the cloud and on-premises datacenters.

In my first month of dealing with Kubernetes, I wrote a two-part blog post about how to run a JBoss WildFly application server cluster in OpenShift/Kubernetes. At the time, I never could have imagined that a simple blog post would ultimately lead the people at Manning to contact me about whether I would like to write a book about Kubernetes. Of course, I couldn't say no to such an offer, even though I was sure they'd approached other people as well and would ultimately pick someone else.

And yet, here we are. After more than a year and a half of writing and researching, the book is done. It's been an awesome journey. Writing a book about a technology is

absolutely the best way to get to know it in much greater detail than you'd learn as just a user. As my knowledge of Kubernetes has expanded during the process and Kubernetes itself has evolved, I've constantly gone back to previous chapters I've written and added additional information. I'm a perfectionist, so I'll never really be absolutely satisfied with the book, but I'm happy to hear that a lot of readers of the Manning Early Access Program (MEAP) have found it to be a great guide to Kubernetes.

My aim is to get the reader to understand the technology itself and teach them how to use the tooling to effectively and efficiently develop and deploy apps to Kubernetes clusters. In the book, I don't put much emphasis on how to actually set up and maintain a proper highly available Kubernetes cluster, but the last part should give readers a very solid understanding of what such a cluster consists of and should allow them to easily comprehend additional resources that deal with this subject.

I hope you'll enjoy reading it, and that it teaches you how to get the most out of the awesome system that is Kubernetes.

acknowledgments

Before I started writing this book, I had no clue how many people would be involved in bringing it from a rough manuscript to a published piece of work. This means there are a lot of people to thank.

First, I'd like to thank Erin Twohey for approaching me about writing this book, and Michael Stephens from Manning, who had full confidence in my ability to write it from day one. His words of encouragement early on really motivated me and kept me motivated throughout the last year and a half.

I would also like to thank my initial development editor Andrew Warren, who helped me get my first chapter out the door, and Elesha Hyde, who took over from Andrew and worked with me all the way to the last chapter. Thank you for bearing with me, even though I'm a difficult person to deal with, as I tend to drop off the radar fairly regularly.

I would also like to thank Jeanne Boyarsky, who was the first reviewer to read and comment on my chapters while I was writing them. Jeanne and Elesha were instrumental in making the book as nice as it hopefully is. Without their comments, the book could never have received such good reviews from external reviewers and readers.

I'd like to thank my technical proofreader, Antonio Magnaghi, and of course all my external reviewers: Al Krinker, Alessandro Campeis, Alexander Myltsev, Csaba Sari, David DiMaria, Elias Rangel, Erisk Zelenka, Fabrizio Cucci, Jared Duncan, Keith Donaldson, Michael Bright, Paolo Antinori, Peter Perlepes, and Tiklu Ganguly. Their positive comments kept me going at times when I worried my writing was utterly awful and completely useless. On the other hand, their constructive criticism helped improve

sections that I'd quickly thrown together without enough effort. Thank you for pointing out the hard-to-understand sections and suggesting ways of improving the book. Also, thank you for asking the right questions, which made me realize I was wrong about two or three things in the initial versions of the manuscript.

I also need to thank readers who bought the early version of the book through Manning's MEAP program and voiced their comments in the online forum or reached out to me directly—especially Vimal Kansal, Paolo Patierno, and Roland Huß, who noticed quite a few inconsistencies and other mistakes. And I would like to thank everyone at Manning who has been involved in getting this book published. Before I finish, I also need to thank my colleague and high school friend Aleš Justin, who brought me to Red Hat, and my wonderful colleagues from the Cloud Enablement team. If I hadn't been at Red Hat or in the team, I wouldn't have been the one to write this book.

Lastly, I would like to thank my wife and my son, who were way too understanding and supportive over the last 18 months, while I was locked in my office instead of spending time with them.

Thank you all!

about this book

Kubernetes in Action aims to make you a proficient user of Kubernetes. It teaches you virtually all the concepts you need to understand to effectively develop and run applications in a Kubernetes environment.

Before diving into Kubernetes, the book gives an overview of container technologies like Docker, including how to build containers, so that even readers who haven't used these technologies before can get up and running. It then slowly guides you through most of what you need to know about Kubernetes—from basic concepts to things hidden below the surface.

Who should read this book

The book focuses primarily on application developers, but it also provides an overview of managing applications from the operational perspective. It's meant for anyone interested in running and managing containerized applications on more than just a single server.

Both beginner and advanced software engineers who want to learn about container technologies and orchestrating multiple related containers at scale will gain the expertise necessary to develop, containerize, and run their applications in a Kubernetes environment.

No previous exposure to either container technologies or Kubernetes is required. The book explains the subject matter in a progressively detailed manner, and doesn't use any application source code that would be too hard for non-expert developers to understand.

Readers, however, should have at least a basic knowledge of programming, computer networking, and running basic commands in Linux, and an understanding of well-known computer protocols like HTTP.

How this book is organized: a roadmap

This book has three parts that cover 18 chapters.

Part 1 gives a short introduction to Docker and Kubernetes, how to set up a Kubernetes cluster, and how to run a simple application in it. It contains two chapters:

- Chapter 1 explains what Kubernetes is, how it came to be, and how it helps to solve today's problems of managing applications at scale.
- Chapter 2 is a hands-on tutorial on how to build a container image and run it in a Kubernetes cluster. It also explains how to run a local single-node Kubernetes cluster and a proper multi-node cluster in the cloud.

Part 2 introduces the key concepts you must understand to run applications in Kubernetes. The chapters are as follows:

- Chapter 3 introduces the fundamental building block in Kubernetes—the pod—and explains how to organize pods and other Kubernetes objects through labels.
- Chapter 4 teaches you how Kubernetes keeps applications healthy by automatically restarting containers. It also shows how to properly run managed pods, horizontally scale them, make them resistant to failures of cluster nodes, and run them at a predefined time in the future or periodically.
- Chapter 5 shows how pods can expose the service they provide to clients running both inside and outside the cluster. It also shows how pods running in the cluster can discover and access services, regardless of whether they live in or out of the cluster.
- Chapter 6 explains how multiple containers running in the same pod can share files and how you can manage persistent storage and make it accessible to pods.
- Chapter 7 shows how to pass configuration data and sensitive information like credentials to apps running inside pods.
- Chapter 8 describes how applications can get information about the Kubernetes environment they're running in and how they can talk to Kubernetes to alter the state of the cluster.
- Chapter 9 introduces the concept of a Deployment and explains the proper way of running and updating applications in a Kubernetes environment.
- Chapter 10 introduces a dedicated way of running stateful applications, which usually require a stable identity and state.

Part 3 dives deep into the internals of a Kubernetes cluster, introduces some additional concepts, and reviews everything you've learned in the first two parts from a higher perspective. This is the last group of chapters:

- Chapter 11 goes beneath the surface of Kubernetes and explains all the components that make up a Kubernetes cluster and what each of them does. It also

explains how pods communicate through the network and how services perform load balancing across multiple pods.

- Chapter 12 explains how to secure your Kubernetes API server, and by extension the cluster, using authentication and authorization.
- Chapter 13 teaches you how pods can access the node's resources and how a cluster administrator can prevent pods from doing that.
- Chapter 14 dives into constraining the computational resources each application is allowed to consume, configuring the applications' Quality of Service guarantees, and monitoring the resource usage of individual applications. It also teaches you how to prevent users from consuming too many resources.
- Chapter 15 discusses how Kubernetes can be configured to automatically scale the number of running replicas of your application, and how it can also increase the size of your cluster when your current number of cluster nodes can't accept any additional applications.
- Chapter 16 shows how to ensure pods are scheduled only to certain nodes or how to prevent them from being scheduled to others. It also shows how to make sure pods are scheduled together or how to prevent that from happening.
- Chapter 17 teaches you how you should develop your applications to make them good citizens of your cluster. It also gives you a few pointers on how to set up your development and testing workflows to reduce friction during development.
- Chapter 18 shows you how you can extend Kubernetes with your own custom objects and how others have done it and created enterprise-class application platforms.

As you progress through these chapters, you'll not only learn about the individual Kubernetes building blocks, but also progressively improve your knowledge of using the `kubectl` command-line tool.

About the code

While this book doesn't contain a lot of actual source code, it does contain a lot of manifests of Kubernetes resources in YAML format and shell commands along with their outputs. All of this is formatted in a `fixed-width font like this` to separate it from ordinary text.

Shell commands are mostly **`in bold`**, to clearly separate them from their output, but sometimes only the most important parts of the command or parts of the command's output are in bold for emphasis. In most cases, the command output has been reformatted to make it fit into the limited space in the book. Also, because the Kubernetes CLI tool `kubectl` is constantly evolving, newer versions may print out more information than what's shown in the book. Don't be confused if they don't match exactly.

Listings sometimes include a line-continuation marker (➥) to show that a line of text wraps to the next line. They also include annotations, which highlight and explain the most important parts.

Within text paragraphs, some very common elements such as Pod, ReplicationController, ReplicaSet, DaemonSet, and so forth are set in regular font to avoid overproliferation of code font and help readability. In some places, "Pod" is capitalized to refer to the Pod resource, and lowercased to refer to the actual group of running containers.

All the samples in the book have been tested with Kubernetes version 1.8 running in Google Kubernetes Engine and in a local cluster run with Minikube. The complete source code and YAML manifests can be found at https://github.com/luksa/kubernetes-in-action or downloaded from the publisher's website at www.manning.com/books/kubernetes-in-action.

Book forum

Purchase of *Kubernetes in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://forums.manning.com/forums/kubernetes-in-action. You can also learn more about Manning's forums and the rules of conduct at https://forums.manning.com/forums/about.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

You can find a wide range of additional Kubernetes resources at the following locations:

- The Kubernetes website at https://kubernetes.io
- The Kubernetes Blog, which regularly posts interesting info (http://blog.kubernetes.io)
- The Kubernetes community's Slack channel at http://slack.k8s.io
- The Kubernetes and Cloud Native Computing Foundation's YouTube channels:
  - https://www.youtube.com/channel/UCZ2bu0qutTOM0tHYa_jkIwg
  - https://www.youtube.com/channel/UCvqbFHwN-nwalWPjPUKpvTA

To gain a deeper understanding of individual topics or even to help contribute to Kubernetes, you can also check out any of the Kubernetes Special Interest Groups (SIGs) at https://github.com/kubernetes/kubernetes/wiki/Special-Interest-Groups-(SIGs).

And, finally, as Kubernetes is open source, there's a wealth of information available in the Kubernetes source code itself. You'll find it at https://github.com/kubernetes/kubernetes and related repositories.

about the author

Marko Lukša is a software engineer with more than 20 years of professional experience developing everything from simple web applications to full ERP systems, frameworks, and middleware software. He took his first steps in programming back in 1985, at the age of six, on a second-hand ZX Spectrum computer his father had bought for him. In primary school, he was the national champion in the Logo programming competition and attended summer coding camps, where he learned to program in Pascal. Since then, he has developed software in a wide range of programming languages.

In high school, he started building dynamic websites when the web was still relatively young. He then moved on to developing software for the healthcare and telecommunications industries at a local company, while studying computer science at the University of Ljubljana, Slovenia. Eventually, he ended up working for Red Hat, initially developing an open source implementation of the Google App Engine API, which utilized Red Hat's JBoss middleware products underneath. He also worked in or contributed to projects like CDI/Weld, Infinispan/JBoss DataGrid, and others.

Since late 2014, he has been part of Red Hat's Cloud Enablement team, where his responsibilities include staying up-to-date on new developments in Kubernetes and related technologies and ensuring the company's middleware software utilizes the features of Kubernetes and OpenShift to their full potential.

about the cover illustration

The figure on the cover of *Kubernetes in Action* is a "Member of the Divan," the Turkish Council of State or governing body. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book's table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book ... 200 years later.

The collection was purchased by a Manning editor at an antiquarian flea market in the "Garage" on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn't have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person's trust in one of us. It recalls something that might have happened a long time ago. We at Manning celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

1 Introducing Kubernetes

This chapter covers

- Understanding how software development and deployment has changed over recent years
- Isolating applications and reducing environment differences using containers
- Understanding how containers and Docker are used by Kubernetes
- Making developers' and sysadmins' jobs easier with Kubernetes

Years ago, most software applications were big monoliths, running either as a single process or as a small number of processes spread across a handful of servers. These legacy systems are still widespread today. They have slow release cycles and are updated relatively infrequently. At the end of every release cycle, developers package up the whole system and hand it over to the ops team, who then deploys and monitors it. In case of hardware failures, the ops team manually migrates it to the remaining healthy servers.

Today, these big monolithic legacy applications are slowly being broken down into smaller, independently running components called microservices. Because

microservices are decoupled from each other, they can be developed, deployed, updated, and scaled individually. This enables you to change components quickly and as often as necessary to keep up with today's rapidly changing business requirements.

But with bigger numbers of deployable components and increasingly larger datacenters, it becomes increasingly difficult to configure, manage, and keep the whole system running smoothly. It's much harder to figure out where to put each of those components to achieve high resource utilization and thereby keep the hardware costs down. Doing all this manually is hard work. We need automation, which includes automatic scheduling of those components to our servers, automatic configuration, supervision, and failure-handling. This is where Kubernetes comes in.

Kubernetes enables developers to deploy their applications themselves and as often as they want, without requiring any assistance from the operations (ops) team. But Kubernetes doesn't benefit only developers. It also helps the ops team by automatically monitoring and rescheduling those apps in the event of a hardware failure. The focus for system administrators (sysadmins) shifts from supervising individual apps to mostly supervising and managing Kubernetes and the rest of the infrastructure, while Kubernetes itself takes care of the apps.

> **NOTE** *Kubernetes* is Greek for pilot or helmsman (the person holding the ship's steering wheel). People pronounce Kubernetes in a few different ways. Many pronounce it as *Koo-ber-nay-tace*, while others pronounce it more like *Koo-ber-netties*. No matter which form you use, people will understand what you mean.

Kubernetes abstracts away the hardware infrastructure and exposes your whole datacenter as a single enormous computational resource. It allows you to deploy and run your software components without having to know about the actual servers underneath. When deploying a multi-component application through Kubernetes, it selects a server for each component, deploys it, and enables it to easily find and communicate with all the other components of your application.

This makes Kubernetes great for most on-premises datacenters, but where it starts to shine is when it's used in the largest datacenters, such as the ones built and operated by cloud providers. Kubernetes allows them to offer developers a simple platform for deploying and running any type of application, while not requiring the cloud provider's own sysadmins to know anything about the tens of thousands of apps running on their hardware.

With more and more big companies accepting the Kubernetes model as the best way to run apps, it's becoming the standard way of running distributed apps both in the cloud, as well as on local on-premises infrastructure.

1.1 Understanding the need for a system like Kubernetes

Before you start getting to know Kubernetes in detail, let's take a quick look at how the development and deployment of applications has changed in recent years. This change is both a consequence of splitting big monolithic apps into smaller microservices

and of the changes in the infrastructure that runs those apps. Understanding these changes will help you better see the benefits of using Kubernetes and container technologies such as Docker.

1.1.1 Moving from monolithic apps to microservices

Monolithic applications consist of components that are all tightly coupled together and have to be developed, deployed, and managed as one entity, because they all run as a single OS process. Changes to one part of the application require a redeployment of the whole application, and over time the lack of hard boundaries between the parts results in the increase of complexity and consequential deterioration of the quality of the whole system because of the unconstrained growth of inter-dependencies between these parts.

Running a monolithic application usually requires a small number of powerful servers that can provide enough resources for running the application. To deal with increasing loads on the system, you then either have to vertically scale the servers (also known as scaling up) by adding more CPUs, memory, and other server components, or scale the whole system horizontally, by setting up additional servers and running multiple copies (or replicas) of an application (scaling out). While scaling up usually doesn't require any changes to the app, it gets expensive relatively quickly and in practice always has an upper limit. Scaling out, on the other hand, is relatively cheap hardware-wise, but may require big changes in the application code and isn't always possible—certain parts of an application are extremely hard or next to impossible to scale horizontally (relational databases, for example). If any part of a monolithic application isn't scalable, the whole application becomes unscalable, unless you can split up the monolith somehow.

SPLITTING APPS INTO MICROSERVICES

These and other problems have forced us to start splitting complex monolithic applications into smaller independently deployable components called microservices. Each microservice runs as an independent process (see figure 1.1) and communicates with other microservices through simple, well-defined interfaces (APIs).

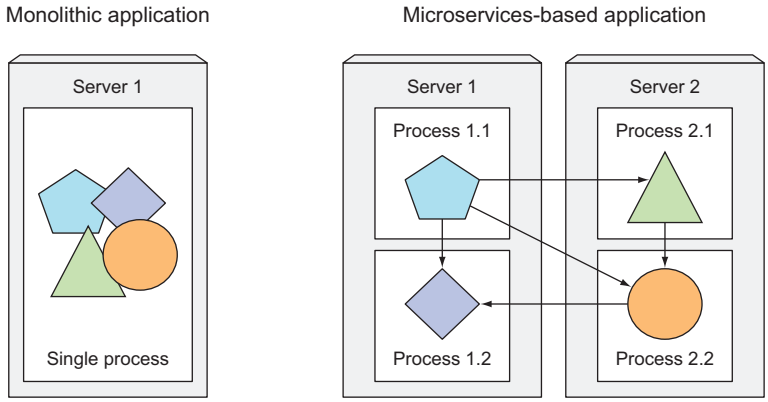

Figure 1.1 Components inside a monolithic application vs. standalone microservices

Microservices communicate through synchronous protocols such as HTTP, over which they usually expose RESTful (REpresentational State Transfer) APIs, or through asynchronous protocols such as AMQP (Advanced Message Queueing Protocol). These protocols are simple, well understood by most developers, and not tied to any specific programming language. Each microservice can be written in the language that's most appropriate for implementing that specific microservice.

Because each microservice is a standalone process with a relatively static external API, it's possible to develop and deploy each microservice separately. A change to one of them doesn't require changes or redeployment of any other service, provided that the API doesn't change or changes only in a backward-compatible way.

SCALING MICROSERVICES

Scaling microservices, unlike monolithic systems, where you need to scale the system as a whole, is done on a per-service basis, which means you have the option of scaling only those services that require more resources, while leaving others at their original scale. Figure 1.2 shows an example. Certain components are replicated and run as multiple processes deployed on different servers, while others run as a single application process. When a monolithic application can't be scaled out because one of its parts is unscalable, splitting the app into microservices allows you to horizontally scale the parts that allow scaling out, and scale the parts that don't, vertically instead of horizontally.

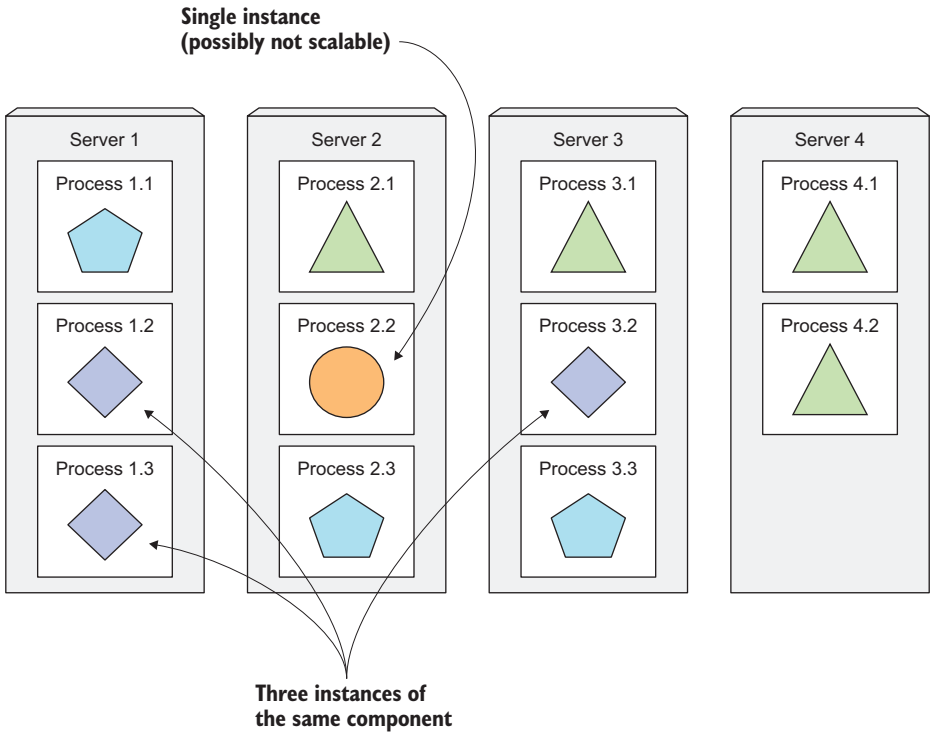

Figure 1.2 Each microservice can be scaled individually.

Deploying microservices

As always, microservices also have drawbacks. When your system consists of only a small number of deployable components, managing those components is easy. It's trivial to decide where to deploy each component, because there aren't that many choices. When the number of those components increases, deployment-related decisions become increasingly difficult because not only does the number of deployment combinations increase, but the number of inter-dependencies between the components increases by an even greater factor.

Microservices perform their work together as a team, so they need to find and talk to each other. When deploying them, someone or something needs to configure all of them properly to enable them to work together as a single system. With increasing numbers of microservices, this becomes tedious and error-prone, especially when you consider what the ops/sysadmin teams need to do when a server fails.

Microservices also bring other problems, such as making it hard to debug and trace execution calls, because they span multiple processes and machines. Luckily, these problems are now being addressed with distributed tracing systems such as Zipkin.

Understanding the divergence of environment requirements

As I've already mentioned, components in a microservices architecture aren't only deployed independently, but are also developed that way. Because of their independence and the fact that it's common to have separate teams developing each component, nothing impedes each team from using different libraries and replacing them whenever the need arises. The divergence of dependencies between application components, like the one shown in figure 1.3, where applications require different versions of the same libraries, is inevitable.

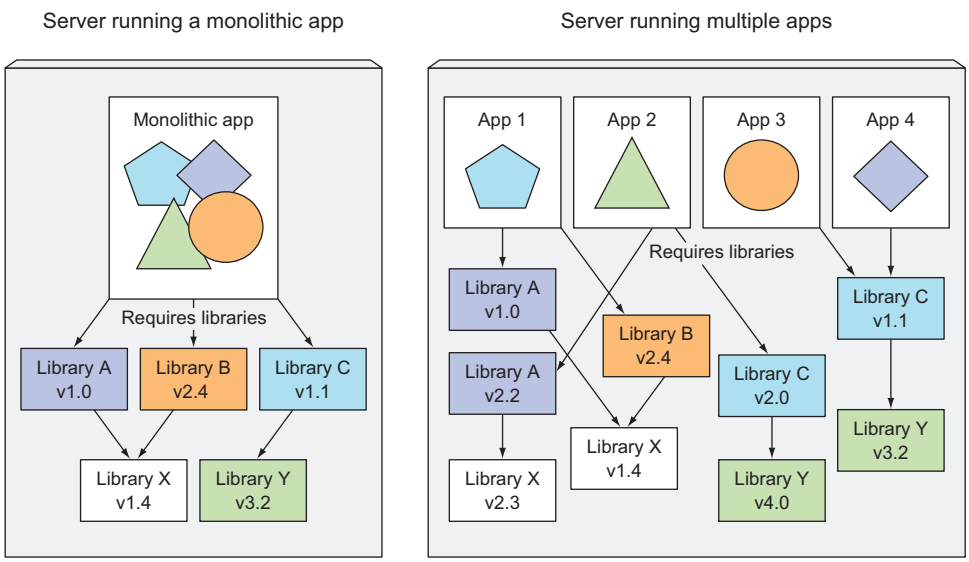

Figure 1.3 Multiple applications running on the same host may have conflicting dependencies.

Deploying dynamically linked applications that require different versions of shared libraries, and/or require other environment specifics, can quickly become a nightmare for the ops team who deploys and manages them on production servers. The bigger the number of components you need to deploy on the same host, the harder it will be to manage all their dependencies to satisfy all their requirements.

1.1.2 Providing a consistent environment to applications

Regardless of how many individual components you're developing and deploying, one of the biggest problems that developers and operations teams always have to deal with is the differences in the environments they run their apps in. Not only is there a huge difference between development and production environments, differences even exist between individual production machines. Another unavoidable fact is that the environment of a single production machine will change over time.

These differences range from hardware to the operating system to the libraries that are available on each machine. Production environments are managed by the operations team, while developers often take care of their development laptops on their own. The difference is how much these two groups of people know about system administration, and this understandably leads to relatively big differences between those two systems, not to mention that system administrators give much more emphasis on keeping the system up to date with the latest security patches, while a lot of developers don't care about that as much.

Also, production systems can run applications from multiple developers or development teams, which isn't necessarily true for developers' computers. A production system must provide the proper environment to all applications it hosts, even though they may require different, even conflicting, versions of libraries.

To reduce the number of problems that only show up in production, it would be ideal if applications could run in the exact same environment during development and in production so they have the exact same operating system, libraries, system configuration, networking environment, and everything else. You also don't want this environment to change too much over time, if at all. Also, if possible, you want the ability to add applications to the same server without affecting any of the existing applications on that server.

1.1.3 Moving to continuous delivery: DevOps and NoOps

In the last few years, we've also seen a shift in the whole application development process and how applications are taken care of in production. In the past, the development team's job was to create the application and hand it off to the operations team, who then deployed it, tended to it, and kept it running. But now, organizations are realizing it's better to have the same team that develops the application also take part in deploying it and taking care of it over its whole lifetime. This means the developer, QA, and operations teams now need to collaborate throughout the whole process. This practice is called DevOps.

Understanding the benefits

Having the developers more involved in running the application in production leads to them having a better understanding of both the users' needs and issues and the problems faced by the ops team while maintaining the app. Application developers are now also much more inclined to give users the app earlier and then use their feedback to steer further development of the app.

To release newer versions of applications more often, you need to streamline the deployment process. Ideally, you want developers to deploy the applications themselves without having to wait for the ops people. But deploying an application often requires an understanding of the underlying infrastructure and the organization of the hardware in the datacenter. Developers don't always know those details and, most of the time, don't even want to know about them.

Letting developers and sysadmins do what they do best

Even though developers and system administrators both work toward achieving the same goal of running a successful software application as a service to its customers, they have different individual goals and motivating factors. Developers love creating new features and improving the user experience. They don't normally want to be the ones making sure that the underlying operating system is up to date with all the security patches and things like that. They prefer to leave that up to the system administrators.

The ops team is in charge of the production deployments and the hardware infrastructure they run on. They care about system security, utilization, and other aspects that aren't a high priority for developers. The ops people don't want to deal with the implicit interdependencies of all the application components and don't want to think about how changes to either the underlying operating system or the infrastructure can affect the operation of the application as a whole, but they must.

Ideally, you want the developers to deploy applications themselves without knowing anything about the hardware infrastructure and without dealing with the ops team. This is referred to as *NoOps*. Obviously, you still need someone to take care of the hardware infrastructure, but ideally, without having to deal with peculiarities of each application running on it.

As you'll see, Kubernetes enables us to achieve all of this. By abstracting away the actual hardware and exposing it as a single platform for deploying and running apps, it allows developers to configure and deploy their applications without any help from the sysadmins and allows the sysadmins to focus on keeping the underlying infrastructure up and running, while not having to know anything about the actual applications running on top of it.

1.2 Introducing container technologies

In section 1.1 I presented a non-comprehensive list of problems facing today's development and ops teams. While you have many ways of dealing with them, this book will focus on how they're solved with Kubernetes.

Kubernetes uses Linux container technologies to provide isolation of running applications, so before we dig into Kubernetes itself, you need to become familiar with the basics of containers to understand what Kubernetes does itself, and what it offloads to container technologies like *Docker* or *rkt* (pronounced "rock-it").

1.2.1 Understanding what containers are

In section 1.1.1 we saw how different software components running on the same machine will require different, possibly conflicting, versions of dependent libraries or have other different environment requirements in general.

When an application is composed of only smaller numbers of large components, it's completely acceptable to give a dedicated Virtual Machine (VM) to each component and isolate their environments by providing each of them with their own operating system instance. But when these components start getting smaller and their numbers start to grow, you can't give each of them their own VM if you don't want to waste hardware resources and keep your hardware costs down. But it's not only about wasting hardware resources. Because each VM usually needs to be configured and managed individually, rising numbers of VMs also lead to wasting human resources, because they increase the system administrators' workload considerably.

ISOLATING COMPONENTS WITH LINUX CONTAINER TECHNOLOGIES

Instead of using virtual machines to isolate the environments of each microservice (or software processes in general), developers are turning to Linux container technologies. They allow you to run multiple services on the same host machine, while not only exposing a different environment to each of them, but also isolating them from each other, similarly to VMs, but with much less overhead.

A process running in a container runs inside the host's operating system, like all the other processes (unlike VMs, where processes run in separate operating systems). But the process in the container is still isolated from other processes. To the process itself, it looks like it's the only one running on the machine and in its operating system.

COMPARING VIRTUAL MACHINES TO CONTAINERS

Compared to VMs, containers are much more lightweight, which allows you to run higher numbers of software components on the same hardware, mainly because each VM needs to run its own set of system processes, which requires additional compute resources in addition to those consumed by the component's own process. A container, on the other hand, is nothing more than a single isolated process running in the host OS, consuming only the resources that the app consumes and without the overhead of any additional processes.

Because of the overhead of VMs, you often end up grouping multiple applications into each VM because you don't have enough resources to dedicate a whole VM to each app. When using containers, you can (and should) have one container for each

application, as shown in figure 1.4. The end-result is that you can fit many more applications on the same bare-metal machine.

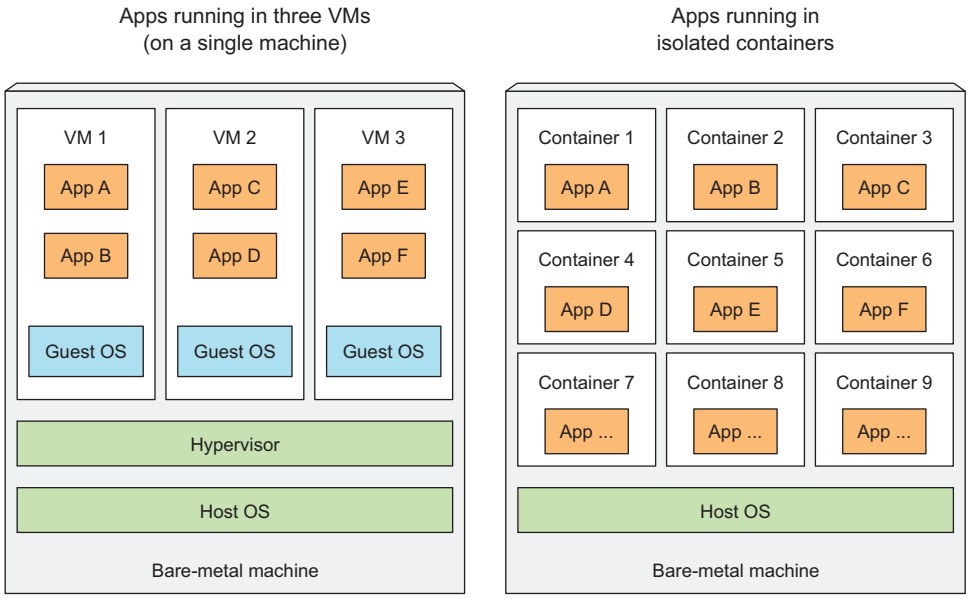

Figure 1.4 Using VMs to isolate groups of applications vs. isolating individual apps with containers

When you run three VMs on a host, you have three completely separate operating systems running on and sharing the same bare-metal hardware. Underneath those VMs is the host's OS and a hypervisor, which divides the physical hardware resources into smaller sets of virtual resources that can be used by the operating system inside each VM. Applications running inside those VMs perform system calls to the guest OS' kernel in the VM, and the kernel then performs x86 instructions on the host's physical CPU through the hypervisor.

> **NOTE** Two types of hypervisors exist. Type 1 hypervisors don't use a host OS, while Type 2 do.

Containers, on the other hand, all perform system calls on the exact same kernel running in the host OS. This single kernel is the only one performing x86 instructions on the host's CPU. The CPU doesn't need to do any kind of virtualization the way it does with VMs (see figure 1.5).

The main benefit of virtual machines is the full isolation they provide, because each VM runs its own Linux kernel, while containers all call out to the same kernel, which can clearly pose a security risk. If you have a limited amount of hardware resources, VMs may only be an option when you have a small number of processes that

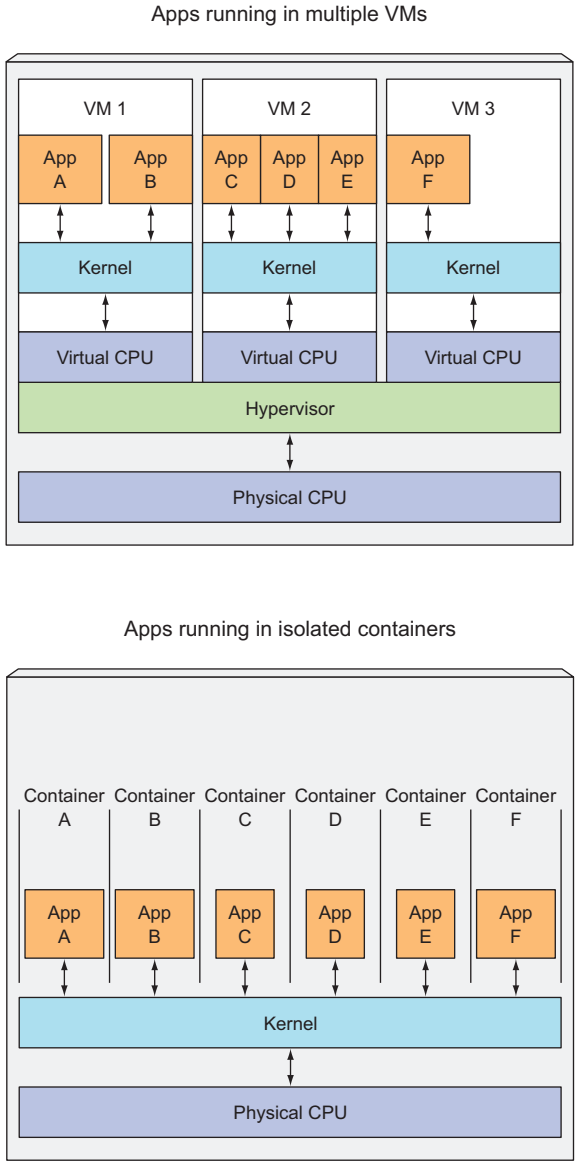

Figure 1.5 The difference between how apps in VMs use the CPU vs. how they use them in containers

you want to isolate. To run greater numbers of isolated processes on the same machine, containers are a much better choice because of their low overhead. Remember, each VM runs its own set of system services, while containers don't, because they all run in the same OS. That also means that to run a container, nothing needs to be booted up, as is the case in VMs. A process run in a container starts up immediately.

Introducing the mechanisms that make container isolation possible

By this point, you're probably wondering how exactly containers can isolate processes if they're running on the same operating system. Two mechanisms make this possible. The first one, *Linux Namespaces*, makes sure each process sees its own personal view of the system (files, processes, network interfaces, hostname, and so on). The second one is *Linux Control Groups (cgroups)*, which limit the amount of resources the process can consume (CPU, memory, network bandwidth, and so on).

Isolating processes with Linux Namespaces

By default, each Linux system initially has one single namespace. All system resources, such as filesystems, process IDs, user IDs, network interfaces, and others, belong to the single namespace. But you can create additional namespaces and organize resources across them. When running a process, you run it inside one of those namespaces. The process will only see resources that are inside the same namespace. Well, multiple kinds of namespaces exist, so a process doesn't belong to one namespace, but to one namespace of each kind.

The following kinds of namespaces exist:

- Mount (mnt)
- Process ID (pid)
- Network (net)
- Inter-process communication (ipc)
- UTS
- User ID (user)

Each namespace kind is used to isolate a certain group of resources. For example, the UTS namespace determines what hostname and domain name the process running inside that namespace sees. By assigning two different UTS namespaces to a pair of processes, you can make them see different local hostnames. In other words, to the two processes, it will appear as though they are running on two different machines (at least as far as the hostname is concerned).

Likewise, what Network namespace a process belongs to determines which network interfaces the application running inside the process sees. Each network interface belongs to exactly one namespace, but can be moved from one namespace to another. Each container uses its own Network namespace, and therefore each container sees its own set of network interfaces.

This should give you a basic idea of how namespaces are used to isolate applications running in containers from each other.

Limiting resources available to a process

The other half of container isolation deals with limiting the amount of system resources a container can consume. This is achieved with cgroups, a Linux kernel feature that limits the resource usage of a process (or a group of processes). A process can't use more than the configured amount of CPU, memory, network bandwidth,

and so on. This way, processes cannot hog resources reserved for other processes, which is similar to when each process runs on a separate machine.

1.2.2 Introducing the Docker container platform

While container technologies have been around for a long time, they've become more widely known with the rise of the Docker container platform. Docker was the first container system that made containers easily portable across different machines. It simplified the process of packaging up not only the application but also all its libraries and other dependencies, even the whole OS file system, into a simple, portable package that can be used to provision the application to any other machine running Docker.

When you run an application packaged with Docker, it sees the exact filesystem contents that you've bundled with it. It sees the same files whether it's running on your development machine or a production machine, even if it the production server is running a completely different Linux OS. The application won't see anything from the server it's running on, so it doesn't matter if the server has a completely different set of installed libraries compared to your development machine.

For example, if you've packaged up your application with the files of the whole Red Hat Enterprise Linux (RHEL) operating system, the application will believe it's running inside RHEL, both when you run it on your development computer that runs Fedora and when you run it on a server running Debian or some other Linux distribution. Only the kernel may be different.

This is similar to creating a VM image by installing an operating system into a VM, installing the app inside it, and then distributing the whole VM image around and running it. Docker achieves the same effect, but instead of using VMs to achieve app isolation, it uses Linux container technologies mentioned in the previous section to provide (almost) the same level of isolation that VMs do. Instead of using big monolithic VM images, it uses container images, which are usually smaller.

A big difference between Docker-based container images and VM images is that container images are composed of layers, which can be shared and reused across multiple images. This means only certain layers of an image need to be downloaded if the other layers were already downloaded previously when running a different container image that also contains the same layers.

UNDERSTANDING DOCKER CONCEPTS

Docker is a platform for packaging, distributing, and running applications. As we've already stated, it allows you to package your application together with its whole environment. This can be either a few libraries that the app requires or even all the files that are usually available on the filesystem of an installed operating system. Docker makes it possible to transfer this package to a central repository from which it can then be transferred to any computer running Docker and executed there (for the most part, but not always, as we'll soon explain).

Three main concepts in Docker comprise this scenario:

- *Images*—A Docker-based container image is something you package your application and its environment into. It contains the filesystem that will be available to the application and other metadata, such as the path to the executable that should be executed when the image is run.
- *Registries*—A Docker Registry is a repository that stores your Docker images and facilitates easy sharing of those images between different people and computers. When you build your image, you can either run it on the computer you've built it on, or you can *push* (upload) the image to a registry and then *pull* (download) it on another computer and run it there. Certain registries are public, allowing anyone to pull images from it, while others are private, only accessible to certain people or machines.
- *Containers*—A Docker-based container is a regular Linux container created from a Docker-based container image. A running container is a process running on the host running Docker, but it's completely isolated from both the host and all other processes running on it. The process is also resource-constrained, meaning it can only access and use the amount of resources (CPU, RAM, and so on) that are allocated to it.

BUILDING, DISTRIBUTING, AND RUNNING A DOCKER IMAGE

Figure 1.6 shows all three concepts and how they relate to each other. The developer first builds an image and then pushes it to a registry. The image is thus available to anyone who can access the registry. They can then pull the image to any other machine running Docker and run the image. Docker creates an isolated container based on the image and runs the binary executable specified as part of the image.

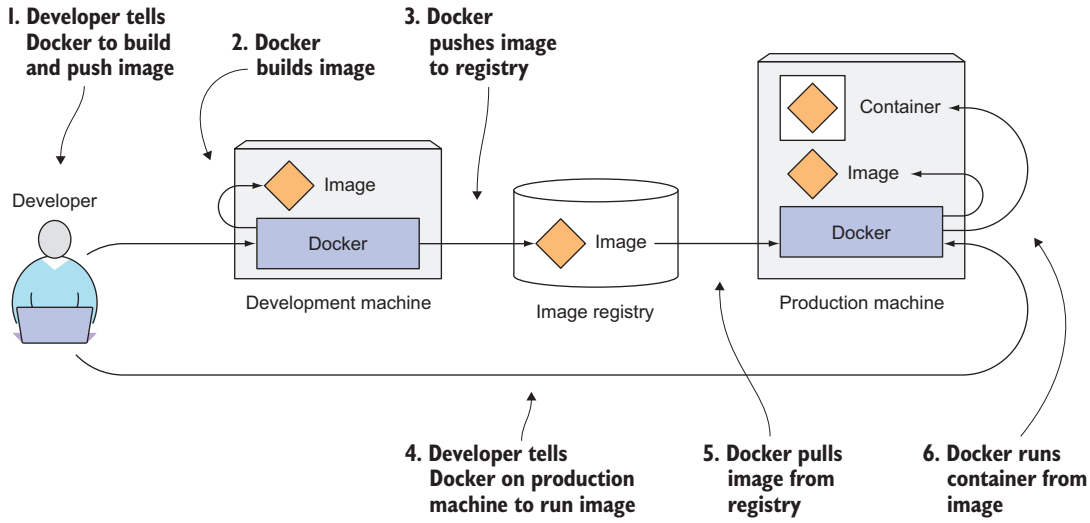

Figure 1.6 Docker images, registries, and containers

COMPARING VIRTUAL MACHINES AND DOCKER CONTAINERS

I've explained how Linux containers are generally like virtual machines, but much more lightweight. Now let's look at how Docker containers specifically compare to virtual machines (and how Docker images compare to VM images). Figure 1.7 again shows the same six applications running both in VMs and as Docker containers.

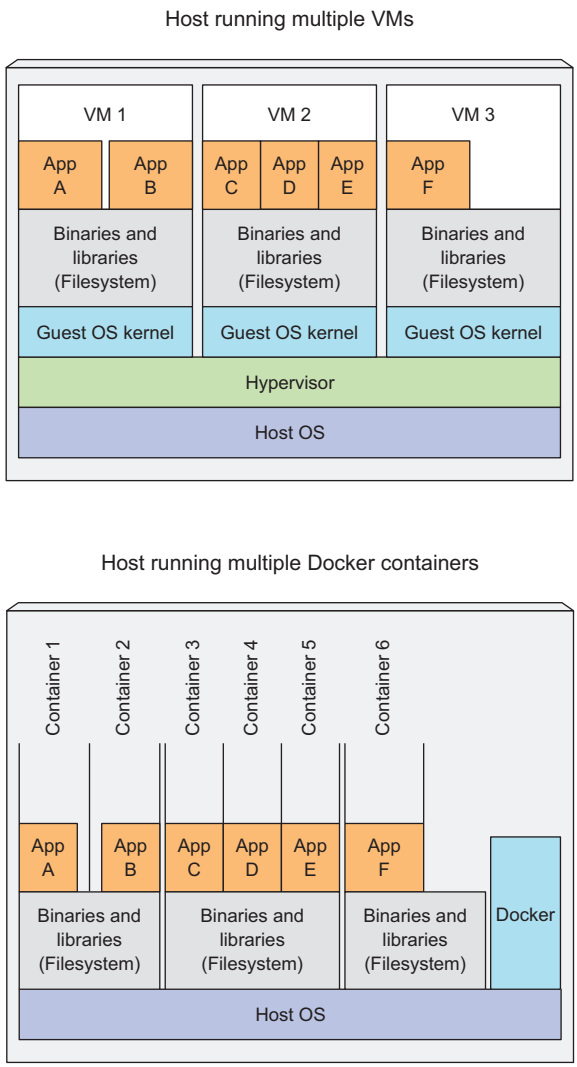

Figure 1.7 Running six apps on three VMs vs. running them in Docker containers

You'll notice that apps A and B have access to the same binaries and libraries both when running in a VM and when running as two separate containers. In the VM, this is obvious, because both apps see the same filesystem (that of the VM). But we said

that each container has its own isolated filesystem. How can both app A and app B share the same files?

Understanding image layers

I've already said that Docker images are composed of layers. Different images can contain the exact same layers because every Docker image is built on top of another image and two different images can both use the same parent image as their base. This speeds up the distribution of images across the network, because layers that have already been transferred as part of the first image don't need to be transferred again when transferring the other image.

But layers don't only make distribution more efficient, they also help reduce the storage footprint of images. Each layer is only stored once. Two containers created from two images based on the same base layers can therefore read the same files, but if one of them writes over those files, the other one doesn't see those changes. Therefore, even if they share files, they're still isolated from each other. This works because container image layers are read-only. When a container is run, a new writable layer is created on top of the layers in the image. When the process in the container writes to a file located in one of the underlying layers, a copy of the whole file is created in the top-most layer and the process writes to the copy.

Understanding the portability limitations of container images

In theory, a container image can be run on any Linux machine running Docker, but one small caveat exists—one related to the fact that all containers running on a host use the host's Linux kernel. If a containerized application requires a specific kernel version, it may not work on every machine. If a machine runs a different version of the Linux kernel or doesn't have the same kernel modules available, the app can't run on it.

While containers are much more lightweight compared to VMs, they impose certain constraints on the apps running inside them. VMs have no such constraints, because each VM runs its own kernel.

And it's not only about the kernel. It should also be clear that a containerized app built for a specific hardware architecture can only run on other machines that have the same architecture. You can't containerize an application built for the x86 architecture and expect it to run on an ARM-based machine because it also runs Docker. You still need a VM for that.

1.2.3 Introducing rkt—an alternative to Docker

Docker was the first container platform that made containers mainstream. I hope I've made it clear that Docker itself doesn't provide process isolation. The actual isolation of containers is done at the Linux kernel level using kernel features such as Linux Namespaces and cgroups. Docker only makes it easy to use those features.

After the success of Docker, the Open Container Initiative (OCI) was born to create open industry standards around container formats and runtime. Docker is part of that initiative, as is *rkt* (pronounced "rock-it"), which is another Linux container engine.

Like Docker, rkt is a platform for running containers. It puts a strong emphasis on security, composability, and conforming to open standards. It uses the OCI container image format and can even run regular Docker container images.

This book focuses on using Docker as the container runtime for Kubernetes, because it was initially the only one supported by Kubernetes. Recently, Kubernetes has also started supporting rkt, as well as others, as the container runtime.

The reason I mention rkt at this point is so you don't make the mistake of thinking Kubernetes is a container orchestration system made specifically for Docker-based containers. In fact, over the course of this book, you'll realize that the essence of Kubernetes isn't orchestrating containers. It's much more. Containers happen to be the best way to run apps on different cluster nodes. With that in mind, let's finally dive into the core of what this book is all about—Kubernetes.

1.3 Introducing Kubernetes

We've already shown that as the number of deployable application components in your system grows, it becomes harder to manage them all. Google was probably the first company that realized it needed a much better way of deploying and managing their software components and their infrastructure to scale globally. It's one of only a few companies in the world that runs hundreds of thousands of servers and has had to deal with managing deployments on such a massive scale. This has forced them to develop solutions for making the development and deployment of thousands of software components manageable and cost-efficient.

1.3.1 Understanding its origins

Through the years, Google developed an internal system called *Borg* (and later a new system called *Omega*), that helped both application developers and system administrators manage those thousands of applications and services. In addition to simplifying the development and management, it also helped them achieve a much higher utilization of their infrastructure, which is important when your organization is that large. When you run hundreds of thousands of machines, even tiny improvements in utilization mean savings in the millions of dollars, so the incentives for developing such a system are clear.

After having kept Borg and Omega secret for a whole decade, in 2014 Google introduced Kubernetes, an open-source system based on the experience gained through Borg, Omega, and other internal Google systems.

1.3.2 Looking at Kubernetes from the top of a mountain

Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it. It relies on the features of Linux containers to run heterogeneous applications without having to know any internal details of these applications and without having to manually deploy these applications on each host. Because these apps run in containers, they don't affect other apps running on the

same server, which is critical when you run applications for completely different organizations on the same hardware. This is of paramount importance for cloud providers, because they strive for the best possible utilization of their hardware while still having to maintain complete isolation of hosted applications.

Kubernetes enables you to run your software applications on thousands of computer nodes as if all those nodes were a single, enormous computer. It abstracts away the underlying infrastructure and, by doing so, simplifies development, deployment, and management for both development and the operations teams.

Deploying applications through Kubernetes is always the same, whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply represent an additional amount of resources available to deployed apps.

Understanding the core of what Kubernetes does

Figure 1.8 shows the simplest possible view of a Kubernetes system. The system is composed of a master node and any number of worker nodes. When the developer submits a list of apps to the master, Kubernetes deploys them to the cluster of worker nodes. What node a component lands on doesn't (and shouldn't) matter—neither to the developer nor to the system administrator.

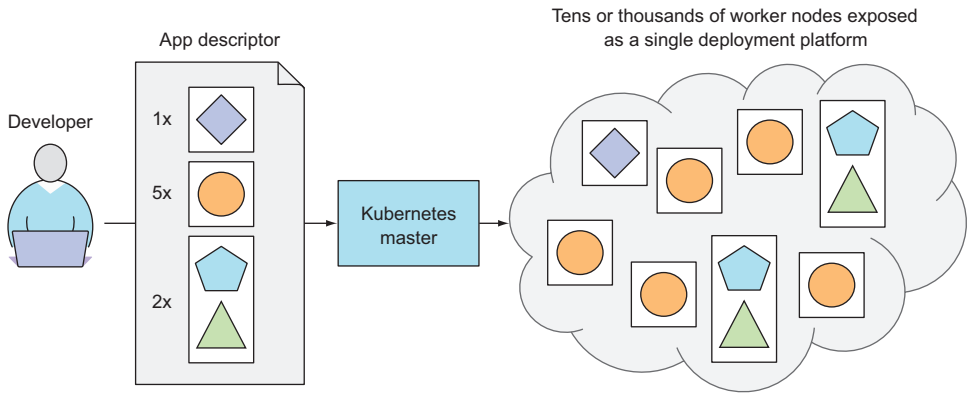

Figure 1.8 Kubernetes exposes the whole datacenter as a single deployment platform.

The developer can specify that certain apps must run together and Kubernetes will deploy them on the same worker node. Others will be spread around the cluster, but they can talk to each other in the same way, regardless of where they're deployed.

Helping developers focus on the core app features

Kubernetes can be thought of as an operating system for the cluster. It relieves application developers from having to implement certain infrastructure-related services into their apps; instead they rely on Kubernetes to provide these services. This includes things such as service discovery, scaling, load-balancing, self-healing, and even leader

election. Application developers can therefore focus on implementing the actual features of the applications and not waste time figuring out how to integrate them with the infrastructure.

HELPING OPS TEAMS ACHIEVE BETTER RESOURCE UTILIZATION

Kubernetes will run your containerized app somewhere in the cluster, provide information to its components on how to find each other, and keep all of them running. Because your application doesn't care which node it's running on, Kubernetes can relocate the app at any time, and by mixing and matching apps, achieve far better resource utilization than is possible with manual scheduling.

1.3.3 Understanding the architecture of a Kubernetes cluster

We've seen a bird's-eye view of Kubernetes' architecture. Now let's take a closer look at what a Kubernetes cluster is composed of. At the hardware level, a Kubernetes cluster is composed of many nodes, which can be split into two types:

- The *master* node, which hosts the *Kubernetes Control Plane* that controls and manages the whole Kubernetes system
- Worker *nodes* that run the actual applications you deploy

Figure 1.9 shows the components running on these two sets of nodes. I'll explain them next.

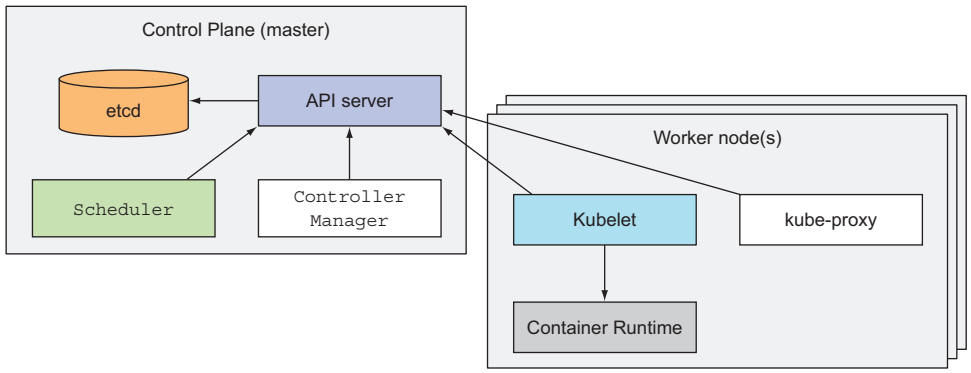

Figure 1.9 The components that make up a Kubernetes cluster

THE CONTROL PLANE

The Control Plane is what controls the cluster and makes it function. It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability. These components are

- The *Kubernetes API Server*, which you and the other Control Plane components communicate with