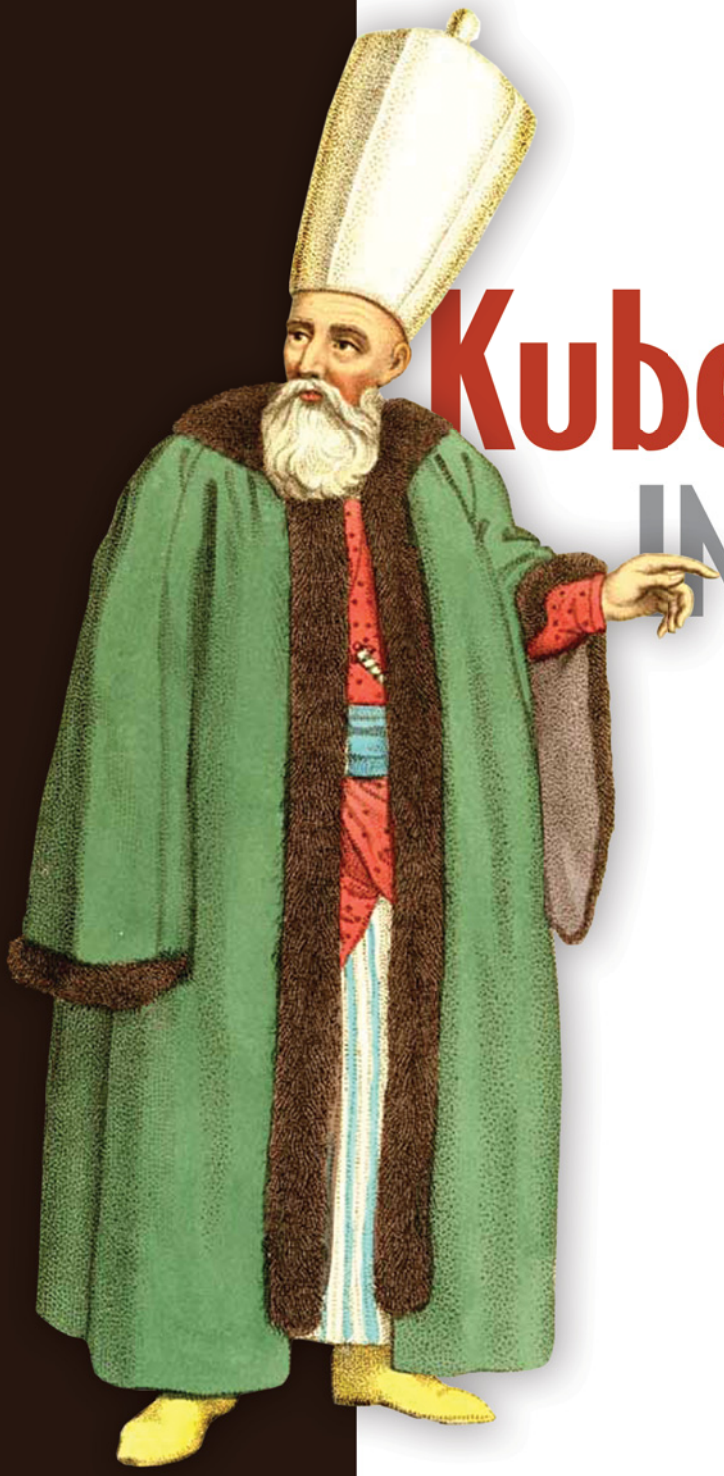
SAMPLE CHAPTER



Kubernetes IN ACTION

Marko Lukša

MANNING

Kubernetes in Action

by Marko Lukša

Chapter 1

brief contents

PART 3 BEYOND THE BASICS

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).

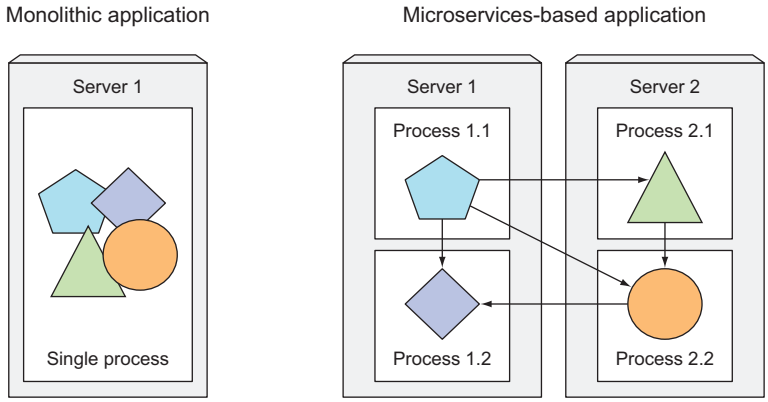


Figure 1.1 Components inside a monolithic application vs. standalone microservices

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.

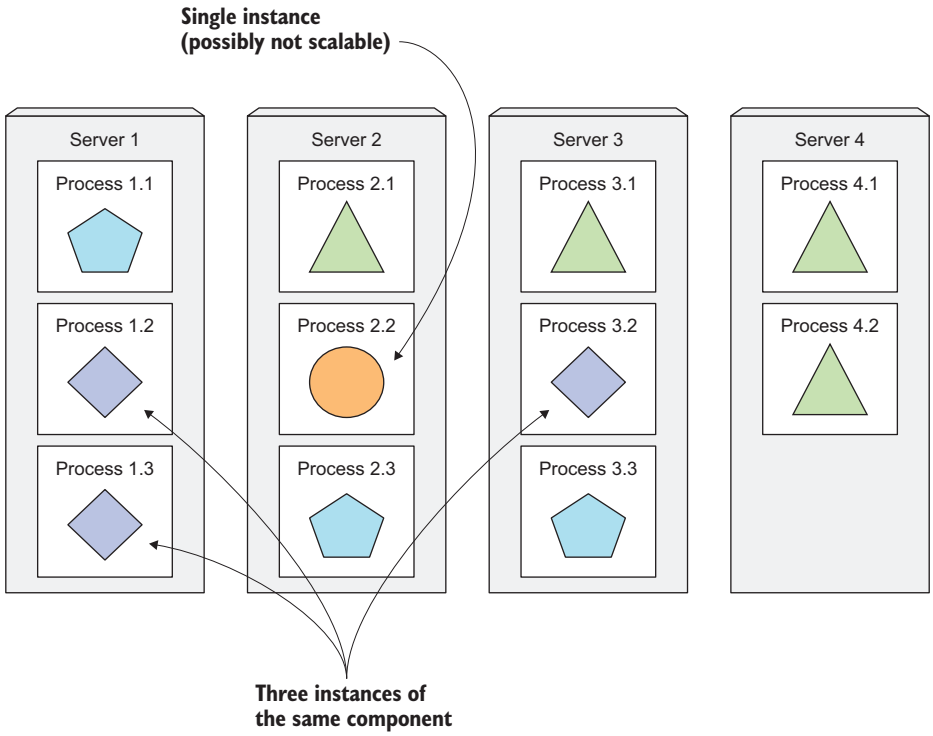


Figure 1.2 Each microservice can be scaled individually.

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.

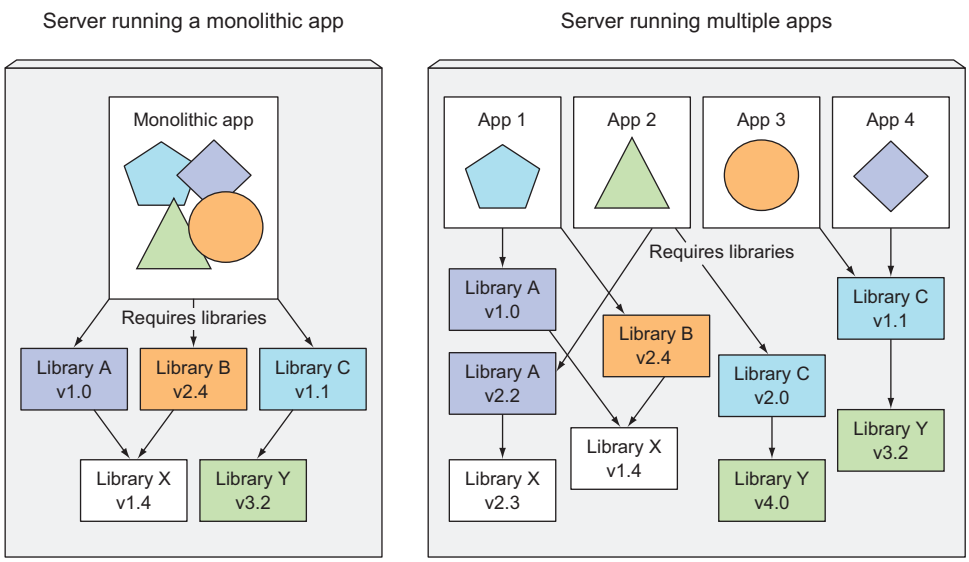


Figure 1.3 Multiple applications running on the same host may have conflicting dependencies.

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.

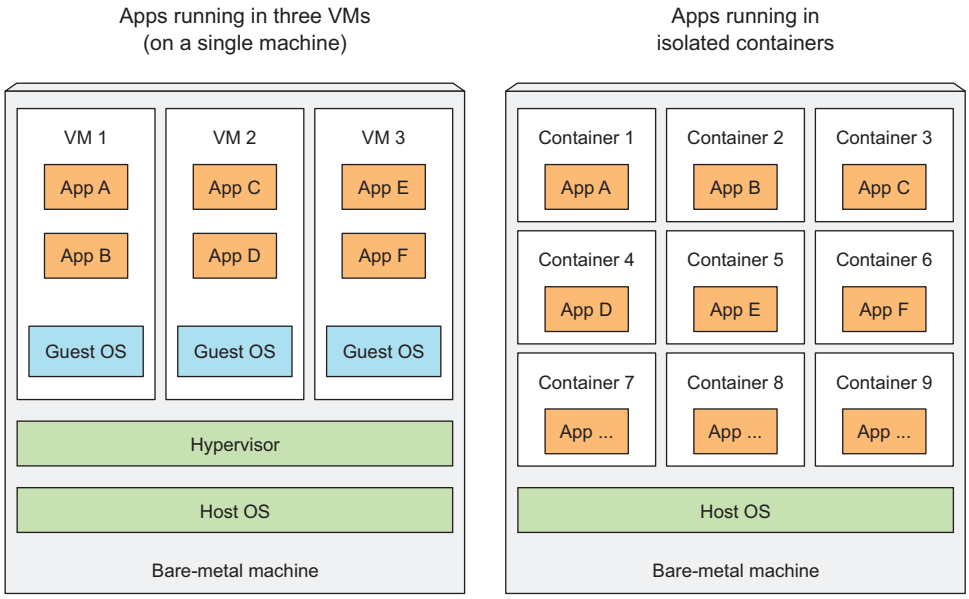


Figure 1.4 Using VMs to isolate groups of applications vs. isolating individual apps with containers

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

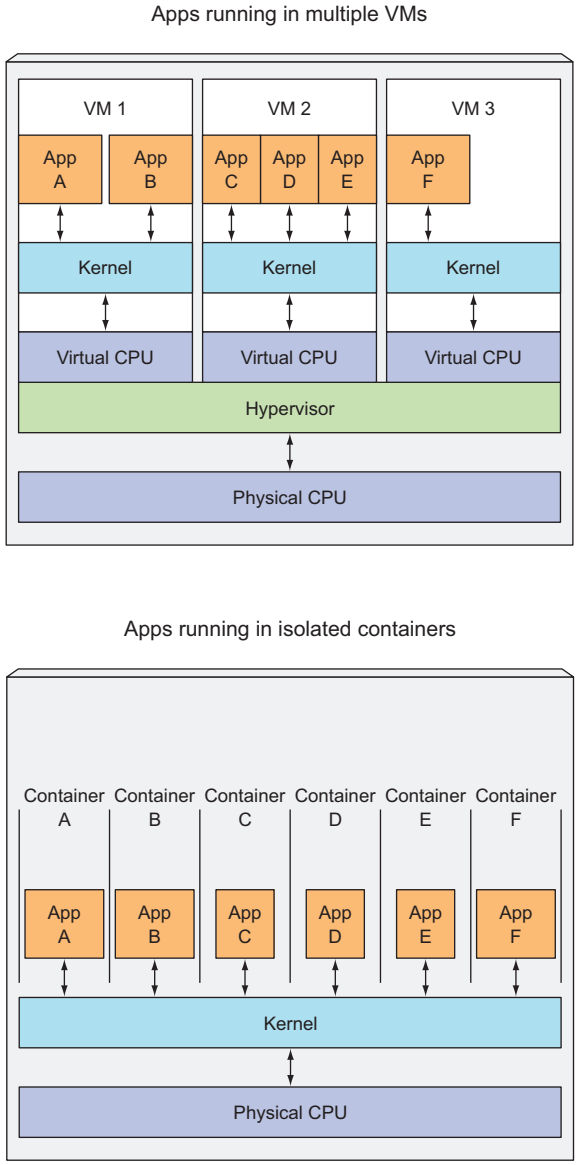


Figure 1.5 The difference between how apps in VMs use the CPU vs. how they use them in containers

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.

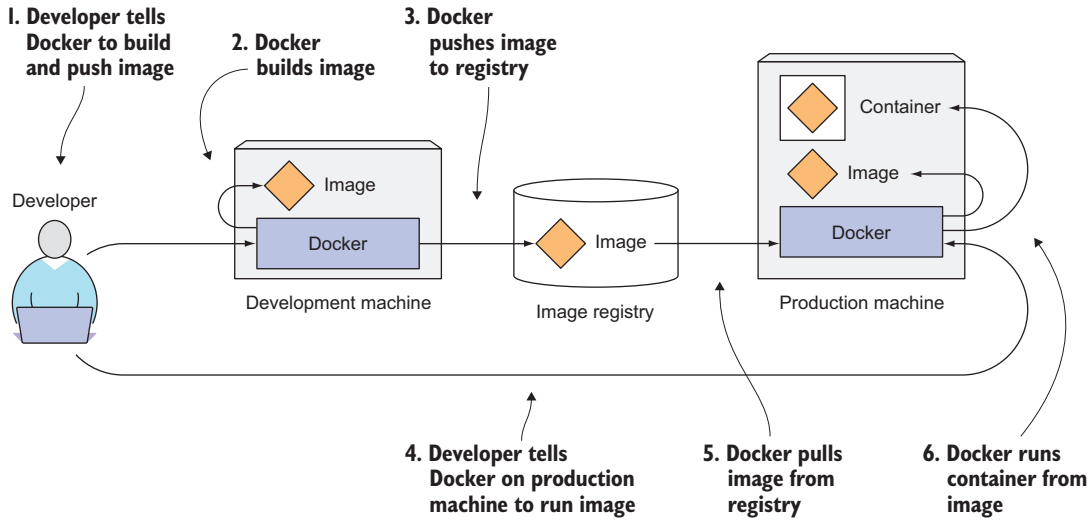


Figure 1.6 Docker images, registries, and containers

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.

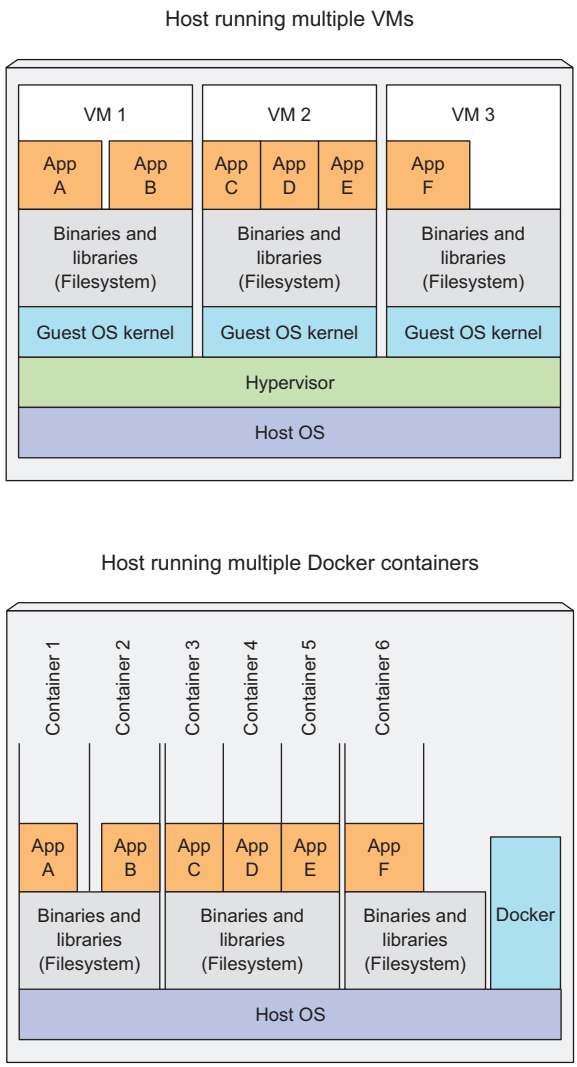


Figure 1.7 Running six apps on three VMs vs. running them in Docker containers

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.

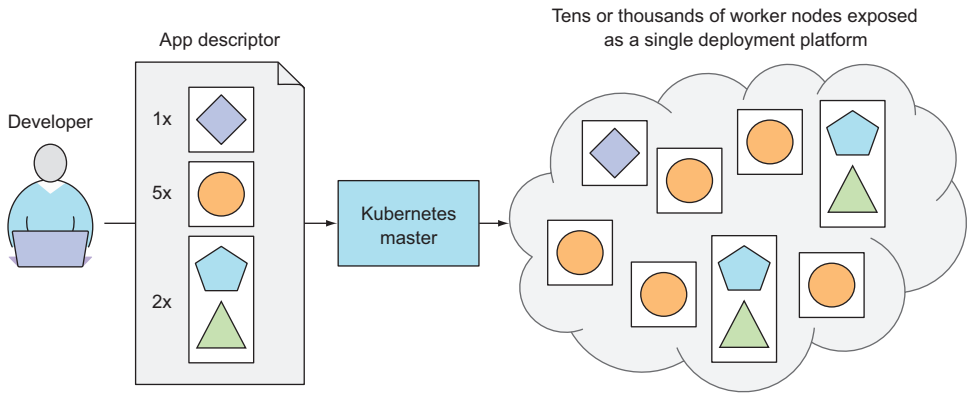


Figure 1.8 Kubernetes exposes the whole datacenter as a single deployment platform.

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.

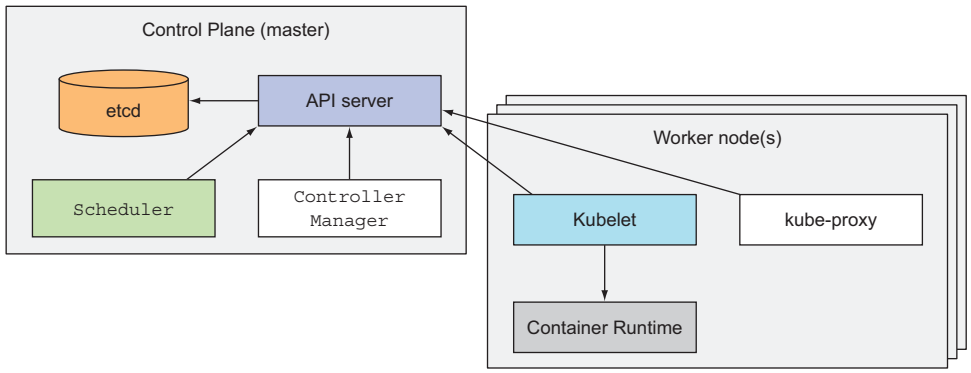


Figure 1.9 The components that make up a Kubernetes cluster

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

- The *Scheduler*, which schedules your apps (assigns a worker node to each deployable component of your application)
- The *Controller Manager*, which performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures, and so on
- *etcd*, a reliable distributed data store that persistently stores the cluster configuration.

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.

THE NODES

The worker nodes are the machines that run your containerized applications. The task of running, monitoring, and providing services to your applications is done by the following components:

- Docker, rkt, or another *container runtime*, which runs your containers
- The *Kubelet*, which talks to the API server and manages containers on its node
- The *Kubernetes Service Proxy (kube-proxy)*, which load-balances network traffic between application components

We'll explain all these components in detail in chapter 11. I'm not a fan of explaining how things work before first explaining *what* something does and teaching people to use it. It's like learning to drive a car. You don't want to know what's under the hood. You first want to learn how to drive it from point A to point B. Only after you learn how to do that do you become interested in how a car makes that possible. After all, knowing what's under the hood may someday help you get the car moving again after it breaks down and leaves you stranded at the side of the road.

1.3.4 Running an application in Kubernetes

To run an application in Kubernetes, you first need to package it up into one or more container images, push those images to an image registry, and then post a description of your app to the Kubernetes API server.

The description includes information such as the container image or images that contain your application components, how those components are related to each other, and which ones need to be run co-located (together on the same node) and which don't. For each component, you can also specify how many copies (or *replicas*) you want to run. Additionally, the description also includes which of those components provide a service to either internal or external clients and should be exposed through a single IP address and made discoverable to the other components.

UNDERSTANDING HOW THE DESCRIPTION RESULTS IN A RUNNING CONTAINER

When the API server processes your app's description, the Scheduler schedules the specified groups of containers onto the available worker nodes based on computational resources required by each group and the unallocated resources on each node

at that moment. The Kubelet on those nodes then instructs the Container Runtime (Docker, for example) to pull the required container images and run the containers.

Examine figure 1.10 to gain a better understanding of how applications are deployed in Kubernetes. The app descriptor lists four containers, grouped into three sets (these sets are called *pods*; we'll explain what they are in chapter 3). The first two pods each contain only a single container, whereas the last one contains two. That means both containers need to run co-located and shouldn't be isolated from each other. Next to each pod, you also see a number representing the number of replicas of each pod that need to run in parallel. After submitting the descriptor to Kubernetes, it will schedule the specified number of replicas of each pod to the available worker nodes. The Kubelets on the nodes will then tell Docker to pull the container images from the image registry and run the containers.

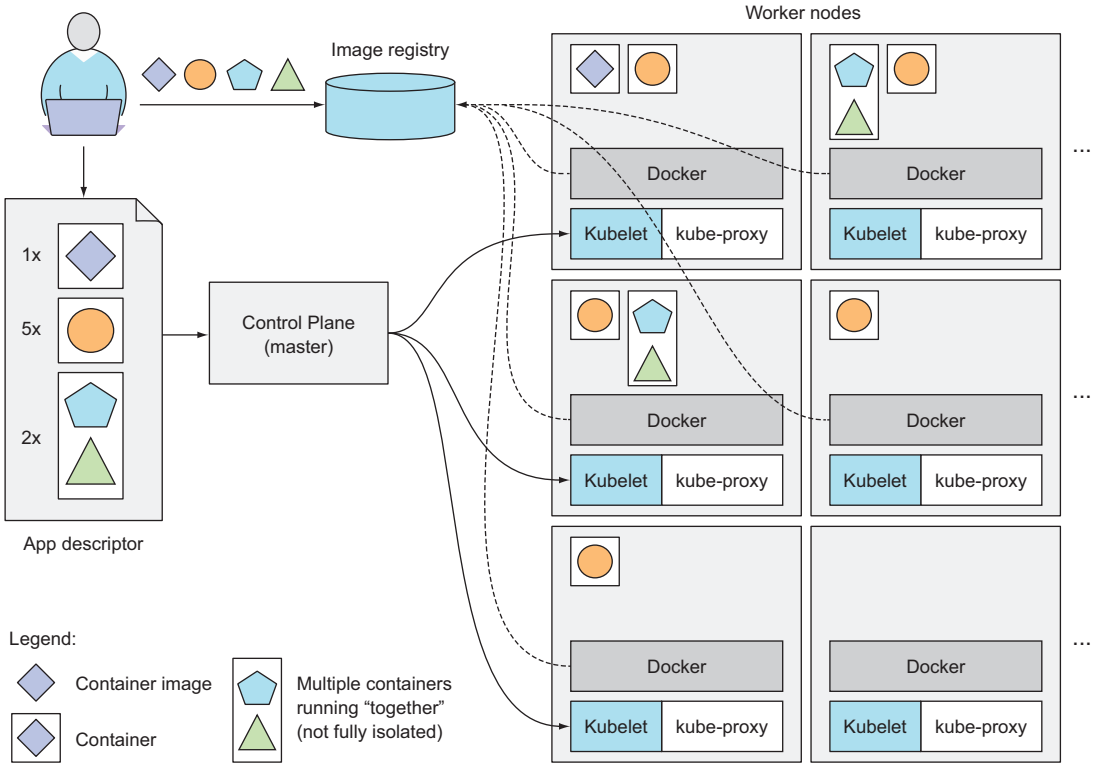


Figure 1.10 A basic overview of the Kubernetes architecture and an application running on top of it

KEEPING THE CONTAINERS RUNNING

Once the application is running, Kubernetes continuously makes sure that the deployed state of the application always matches the description you provided. For example, if

you specify that you always want five instances of a web server running, Kubernetes will always keep exactly five instances running. If one of those instances stops working properly, like when its process crashes or when it stops responding, Kubernetes will restart it automatically.

Similarly, if a whole worker node dies or becomes inaccessible, Kubernetes will select new nodes for all the containers that were running on the node and run them on the newly selected nodes.

Scaling the number of copies

While the application is running, you can decide you want to increase or decrease the number of copies, and Kubernetes will spin up additional ones or stop the excess ones, respectively. You can even leave the job of deciding the optimal number of copies to Kubernetes. It can automatically keep adjusting the number, based on real-time metrics, such as CPU load, memory consumption, queries per second, or any other metric your app exposes.

Hitting a moving target

We've said that Kubernetes may need to move your containers around the cluster. This can occur when the node they were running on has failed or because they were evicted from a node to make room for other containers. If the container is providing a service to external clients or other containers running in the cluster, how can they use the container properly if it's constantly moving around the cluster? And how can clients connect to containers providing a service when those containers are replicated and spread across the whole cluster?

To allow clients to easily find containers that provide a specific service, you can tell Kubernetes which containers provide the same service and Kubernetes will expose all of them at a single static IP address and expose that address to all applications running in the cluster. This is done through environment variables, but clients can also look up the service IP through good old DNS. The kube-proxy will make sure connections to the service are load balanced across all the containers that provide the service. The IP address of the service stays constant, so clients can always connect to its containers, even when they're moved around the cluster.

1.3.5 Understanding the benefits of using Kubernetes

If you have Kubernetes deployed on all your servers, the ops team doesn't need to deal with deploying your apps anymore. Because a containerized application already contains all it needs to run, the system administrators don't need to install anything to deploy and run the app. On any node where Kubernetes is deployed, Kubernetes can run the app immediately without any help from the sysadmins.

Simplifying application deployment

Because Kubernetes exposes all its worker nodes as a single deployment platform, application developers can start deploying applications on their own and don't need to know anything about the servers that make up the cluster.

In essence, all the nodes are now a single bunch of computational resources that are waiting for applications to consume them. A developer doesn't usually care what kind of server the application is running on, as long as the server can provide the application with adequate system resources.

Certain cases do exist where the developer does care what kind of hardware the application should run on. If the nodes are heterogeneous, you'll find cases when you want certain apps to run on nodes with certain capabilities and run other apps on others. For example, one of your apps may require being run on a system with SSDs instead of HDDs, while other apps run fine on HDDs. In such cases, you obviously want to ensure that particular app is always scheduled to a node with an SSD.

Without using Kubernetes, the sysadmin would select one specific node that has an SSD and deploy the app there. But when using Kubernetes, instead of selecting a specific node where your app should be run, it's more appropriate to tell Kubernetes to only choose among nodes with an SSD. You'll learn how to do that in chapter 3.

ACHIEVING BETTER UTILIZATION OF HARDWARE

By setting up Kubernetes on your servers and using it to run your apps instead of running them manually, you've decoupled your app from the infrastructure. When you tell Kubernetes to run your application, you're letting it choose the most appropriate node to run your application on based on the description of the application's resource requirements and the available resources on each node.

By using containers and not tying the app down to a specific node in your cluster, you're allowing the app to freely move around the cluster at any time, so the different app components running on the cluster can be mixed and matched to be packed tightly onto the cluster nodes. This ensures the node's hardware resources are utilized as best as possible.

The ability to move applications around the cluster at any time allows Kubernetes to utilize the infrastructure much better than what you can achieve manually. Humans aren't good at finding optimal combinations, especially when the number of all possible options is huge, such as when you have many application components and many server nodes they can be deployed on. Computers can obviously perform this work much better and faster than humans.

HEALTH CHECKING AND SELF-HEALING

Having a system that allows moving an application across the cluster at any time is also valuable in the event of server failures. As your cluster size increases, you'll deal with failing computer components ever more frequently.

Kubernetes monitors your app components and the nodes they run on and automatically reschedules them to other nodes in the event of a node failure. This frees the ops team from having to migrate app components manually and allows the team to immediately focus on fixing the node itself and returning it to the pool of available hardware resources instead of focusing on relocating the app.

If your infrastructure has enough spare resources to allow normal system operation even without the failed node, the ops team doesn't even need to react to the failure

immediately, such as at 3 a.m. They can sleep tight and deal with the failed node during regular work hours.

Automatic scaling

Using Kubernetes to manage your deployed applications also means the ops team doesn't need to constantly monitor the load of individual applications to react to sudden load spikes. As previously mentioned, Kubernetes can be told to monitor the resources used by each application and to keep adjusting the number of running instances of each application.

If Kubernetes is running on cloud infrastructure, where adding additional nodes is as easy as requesting them through the cloud provider's API, Kubernetes can even automatically scale the whole cluster size up or down based on the needs of the deployed applications.

Simplifying application development

The features described in the previous section mostly benefit the operations team. But what about the developers? Does Kubernetes bring anything to their table? It definitely does.

If you turn back to the fact that apps run in the same environment both during development and in production, this has a big effect on when bugs are discovered. We all agree the sooner you discover a bug, the easier it is to fix it, and fixing it requires less work. It's the developers who do the fixing, so this means less work for them.

Then there's the fact that developers don't need to implement features that they would usually implement. This includes discovery of services and/or peers in a clustered application. Kubernetes does this instead of the app. Usually, the app only needs to look up certain environment variables or perform a DNS lookup. If that's not enough, the application can query the Kubernetes API server directly to get that and/or other information. Querying the Kubernetes API server like that can even save developers from having to implement complicated mechanisms such as leader election.

As a final example of what Kubernetes brings to the table, you also need to consider the increase in confidence developers will feel knowing that when a new version of their app is going to be rolled out, Kubernetes can automatically detect if the new version is bad and stop its rollout immediately. This increase in confidence usually accelerates the continuous delivery of apps, which benefits the whole organization.

1.4 Summary

In this introductory chapter, you've seen how applications have changed in recent years and how they can now be harder to deploy and manage. We've introduced Kubernetes and shown how it, together with Docker and other container platforms, helps deploy and manage applications and the infrastructure they run on. You've learned that

- Monolithic apps are easier to deploy, but harder to maintain over time and sometimes impossible to scale.

- Microservices-based application architectures allow easier development of each component, but are harder to deploy and configure to work as a single system.
- Linux containers provide much the same benefits as virtual machines, but are far more lightweight and allow for much better hardware utilization.
- Docker improved on existing Linux container technologies by allowing easier and faster provisioning of containerized apps together with their OS environments.
- Kubernetes exposes the whole datacenter as a single computational resource for running applications.
- Developers can deploy apps through Kubernetes without assistance from sysadmins.
- Sysadmins can sleep better by having Kubernetes deal with failed nodes automatically.

In the next chapter, you'll get your hands dirty by building an app and running it in Docker and then in Kubernetes.

SOFTWARE DEVELOPMENT/OPERATIONS

Kubernetes IN ACTION

Marko Lukša

Free eBook
SEE INSERT

Kubernetes is Greek for "helmsman," your guide through unknown waters. The Kubernetes container orchestration system safely manages the structure and flow of a distributed application, organizing containers and services for maximum efficiency. Kubernetes serves as an operating system for your clusters, eliminating the need to factor the underlying network and server infrastructure into your designs.

Kubernetes in Action teaches you to use Kubernetes to deploy container-based distributed applications. You'll start with an overview of Docker and Kubernetes before building your first Kubernetes cluster. You'll gradually expand your initial application, adding features and deepening your knowledge of Kubernetes architecture and operation. As you navigate this comprehensive guide, you'll explore high-value topics like monitoring, tuning, and scaling.

What's Inside

- Kubernetes' internals
- Deploying containers across a cluster
- Securing clusters
- Updating applications with zero downtime

Written for intermediate software developers with little or no familiarity with Docker or container orchestration systems.

Marko Lukša is an engineer at Red Hat working on Kubernetes and OpenShift.

"Authoritative and exhaustive. In a hands-on style, the author teaches how to manage the complete lifecycle of any distributed and scalable application."
—Antonio Magnaghi, System1

"The best parts are the real-world examples. They don't just apply the concepts, they road test them."
—Paolo Antinori, Red Hat

"An in-depth discussion of Kubernetes and related technologies. A must-have!"
—Al Krinker, USPTO

"The full path to becoming a professional Kubernaut. Fundamental reading."
—Csaba Sári
Chimera Entertainment

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/kubernetes-in-action

ISBN-13: 978-1-61729-372-6
ISBN-10: 1-61729-372-5
55999
9 781617 293726

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]