Chapter 1. Introduction

Kubernetes is an open source orchestrator for deploying containerized applications. Kubernetes was originally developed by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs.[1]

But Kubernetes is much more than simply exporting technology developed at Google. Kubernetes has grown to be the product of a rich and growing open source community. This means that Kubernetes is a product that is suited not just to the needs of internet-scale companies but to cloud-native developers of all scales, from a cluster of Raspberry Pi computers to a warehouse full of the latest machines. Kubernetes provides the software necessary to successfully build and deploy reliable, scalable distributed systems.

You may be wondering what we mean when we say "reliable, scalable distributed systems." More and more services are delivered over the network via APIs. These APIs are often delivered by a *distributed system*, the various pieces that implement the API running on different machines, connected via the network and coordinating their actions via network communication. Because we rely on these APIs increasingly for all aspects of our daily lives (e.g., finding directions to the nearest hospital), these systems must be highly *reliable*. They cannot fail, even if a part of the system crashes or otherwise fails. Likewise, they must maintain *availability* even during software rollouts or other maintenance events. Finally, because more and more of the world is coming online and using such services, they must be highly *scalable* so that they can grow their capacity to keep up with ever-increasing usage without radical redesign of the distributed system that implements the services.

Depending on when and why you have come to hold this book in your hands, you may have varying degrees of experience with containers, distributed systems, and Kubernetes. Regardless of what your experience is, we believe this book will enable you to make the most of your use of Kubernetes.

There are many reasons why people come to use containers and container APIs like Kubernetes, but we believe they effectively all can be traced back to one of

these benefits:

- Velocity
- Scaling (of both software and teams)
- Abstracting your infrastructure
- Efficiency

In the following sections we describe how Kubernetes can help provide each of these benefits.

Velocity

Velocity is the key component in nearly all software development today. The changing nature of software from boxed software shipped on CDs to web-based services that change every few hours means that the difference between you and your competitors is often the speed with which you can develop and deploy new components and features.

It is important to note, however, that this velocity is not defined in terms of simply raw speed. While your users are always looking for iterative improvement, they are more interested in a highly reliable service. Once upon a time, it was OK for a service to be down for maintenance at midnight every night. But today, our users expect constant uptime, even if the software they are running is changing constantly.

Consequently, velocity is measured not in terms of the raw number of features you can ship per hour or day, but rather in terms of the number of things you can ship while maintaining a highly available service.

In this way, containers and Kubernetes can provide the tools that you need to move quickly, while staying available. The core concepts that enable this are immutability, declarative configuration, and online self-healing systems. These ideas all interrelate to radically improve the speed with which you can reliably deploy software.

The Value of Immutability

Containers and Kubernetes encourage developers to build distributed systems that adhere to the principles of immutable infrastructure. With immutable infrastructure, once an artifact is created in the system it does not change via user modifications.

Traditionally, computers and software systems have been treated as *mutable* infrastructure. With mutable infrastructure, changes are applied as incremental updates to an existing system. A system upgrade via the `apt-get update` tool is a good example of an update to a mutable system. Running `apt` sequentially downloads any updated binaries, copies them on top of older binaries, and makes incremental updates to configuration files. With a mutable system, the current state of the infrastructure is not represented as a single artifact, but rather an accumulation of incremental updates and changes. On many systems these incremental updates come from not just system upgrades but operator modifications as well.

In contrast, in an immutable system, rather than a series of incremental updates and changes, an entirely new, complete image is built, where the update simply replaces the entire image with the newer image in a single operation. There are no incremental changes. As you can imagine, this is a significant shift from the more traditional world of configuration management.

To make this more concrete in the world of containers, consider two different ways to upgrade your software:

1. You can log into a container, run a command to download your new software, kill the old server, and start the new one.

2. You can build a new container image, push it to a container registry, kill the existing container, and start a new one.

At first blush, these two approaches might seem largely indistinguishable. So what is it about the act of building a new container that improves reliability?

The key differentiation is the artifact that you create, and the record of how you created it. These records make it easy to understand exactly the differences in some new version and, if something goes wrong, determine what has changed

and how to fix it.

Additionally, building a new image rather than modifying an existing one means the old image is still around, and can quickly be used for a rollback if an error occurs. In contrast, once you copy your new binary over an existing binary, such rollback is nearly impossible.

Immutable container images are at the core of everything that you will build in Kubernetes. It is possible to imperatively change running containers, but this is an antipattern to be used only in extreme cases where there are no other options (e.g., if it is the only way to temporarily repair a mission-critical production system). And even then, the changes must also be recorded through a declarative configuration update at some later time, after the fire is out.

Declarative Configuration

Immutability extends beyond containers running in your cluster to the way you describe your application to Kubernetes. Everything in Kubernetes is a *declarative configuration object* that represents the desired state of the system. It is Kubernetes's job to ensure that the actual state of the world matches this desired state.

Much like mutable versus immutable infrastructure, declarative configuration is an alternative to *imperative* configuration, where the state of the world is defined by the execution of a series of instructions rather than a declaration of the desired state of the world. While imperative commands define actions, declarative configurations define state.

To understand these two approaches, consider the task of producing three replicas of a piece of software. With an imperative approach, the configuration would say: "run A, run B, and run C." The corresponding declarative configuration would be "replicas equals three."

Because it describes the state of the world, declarative configuration does not have to be executed to be understood. Its impact is concretely declared. Since the effects of declarative configuration can be understood before they are executed, declarative configuration is far less error-prone. Further, the traditional tools of software development, such as source control, code review, and unit testing, can be used in declarative configuration in ways that are impossible for imperative instructions.

The combination of declarative state stored in a version control system and Kubernetes's ability to make reality match this declarative state makes rollback of a change trivially easy. It is simply restating the previous declarative state of the system. With imperative systems this is usually impossible, since while the imperative instructions describe how to get you from point *A* to point *B*, they rarely include the reverse instructions that can get you back.

Self-Healing Systems

Kubernetes is an online, self-healing system. When it receives a desired state configuration, it does not simply take actions to make the current state match the desired state a single time. It continuously takes actions to ensure that the current state matches the desired state. This means that not only will Kubernetes initialize your system, but it will guard it against any failures or perturbations that might destabilize your system and affect reliability.

A more traditional operator repair involves a manual series of mitigation steps, or human intervention performed in response to some sort of alert. Imperative repair like this is more expensive (since it generally requires an on-call operator to be available to enact the repair). It is also generally slower, since a human must often wake up and log in to respond. Furthermore, it is less reliable since the imperative series of repair operations suffer from all of the problems of imperative management described in the previous section. Self-healing systems like Kubernetes both reduce the burden on operators and improve the overall reliability of the system by performing reliable repairs more quickly.

As a concrete example of this self-healing behavior, if you assert a desired state of three replicas to Kubernetes, it does not just create three replicas — it continuously ensures that there are exactly three replicas. If you manually create a fourth replica Kubernetes will destroy one to bring the number back to three. If you manually destroy a replica, Kubernetes will create one to again return you to the desired state.

Online self-healing systems improve developer velocity because the time and energy you might otherwise have spent on operations and maintenance can instead be spent on developing and testing new features.

Scaling Your Service and Your Teams

As your product grows, its inevitable that you will need to scale both your software and the teams that develop it. Fortunately, Kubernetes can help with both of these goals. Kubernetes achieves scalability by favoring *decoupled* architectures.

Decoupling

In a decoupled architecture each component is separated from other components by defined APIs and service load balancers. APIs and load balancers isolate each piece of the system from the others. APIs provide a buffer between implementer and consumer, and load balancers provide a buffer between running instances of each service.

Decoupling components via load balancers makes it easy to scale the programs that make up your service, because increasing the size (and therefore the capacity) of the program can be done without adjusting or reconfiguring any of the other layers of your service.

Decoupling servers via APIs makes it easier to scale the development teams because each team can focus on a single, smaller *microservice* with a comprehensible surface area. Crisp APIs between microservices limit the amount of cross-team communication overhead required to build and deploy software. This communication overhead is often the major restricting factor when scaling teams.

Easy Scaling for Applications and Clusters

Concretely, when you need to scale your service, the immutable, declarative nature of Kubernetes makes this scaling trivial to implement. Because your containers are immutable, and the number of replicas is simply a number in a declarative config, scaling your service upward is simply a matter of changing a number in a configuration file, asserting this new declarative state to Kubernetes, and letting it take care of the rest. Alternately, you can set up autoscaling and simply let Kubernetes take care of it for you.

Of course, that sort of scaling assumes that there are resources available in your cluster to consume. Sometimes you actually need to scale up the cluster itself. Here again, Kubernetes makes this task easier. Because each machine in a cluster is entirely identical to every other machine, and the applications themselves are decoupled from the details of the machine by containers, adding additional resources to the cluster is simply a matter of imaging a new machine and joining it into the cluster. This can be accomplished via a few simple commands or via a prebaked machine image.

One of the challenges of scaling machine resources is predicting their use. If you are running on physical infrastructure, the time to obtain a new machine is measured in days or weeks. On both physical and cloud infrastructure, predicting future costs is difficult because it is hard to predict the growth and scaling needs of specific applications.

Kubernetes can simplify forecasting future compute costs. To understand why this is true, consider scaling up three teams, A, B, and C. Historically you have seen that each team's growth is highly variable and thus hard to predict. If you are provisioning individual machines for each service, you have no choice but to forecast based on the maximum expected growth for each service, since machines dedicated to one team cannot be used for another team. If instead you use Kubernetes to decouple the teams from the specific machines they are using, you can forecast growth based on the aggregate growth of all three services. Combining three variable growth rates into a single growth rate reduces statistical noise and produces a more reliable forecast of expected growth. Furthermore, decoupling the teams from specific machines means that teams can share fractional parts of each other's machines, reducing even further the

overheads associated with forecasting growth of computing resources.

Scaling Development Teams with Microservices

As noted in a variety of research, the ideal team size is the “two-pizza team,” or roughly six to eight people, because this group size often results in good knowledge sharing, fast decision making, and a common sense of purpose. Larger teams tend to suffer from hierarchy, poor visibility, and infighting, which hinder agility and success.

However, many projects require significantly more resources to be successful and achieve their goals. Consequently, there is a tension between the ideal team size for agility and the necessary team size for the product’s end goals.

The common solution to this tension has been the development of decoupled, service-oriented teams that each build a single microservice. Each small team is responsible for the design and delivery of a service that is consumed by other small teams. The aggregation of all of these services ultimately provides the implementation of the overall product’s surface area.

Kubernetes provides numerous abstractions and APIs that make it easier to build these decoupled microservice architectures.

- Pods, or groups of containers, can group together container images developed by different teams into a single deployable unit.
- Kubernetes services provide load balancing, naming, and discovery to isolate one microservice from another.
- Namespaces provide isolation and access control, so that each microservice can control the degree to which other services interact with it.
- Ingress objects provide an easy-to-use frontend that can combine multiple microservices into a single externalized API surface area.

Finally, decoupling the application container image and machine means that different microservices can colocate on the same machine without interfering with each other, reducing the overhead and cost of microservice architectures. The health-checking and rollout features of Kubernetes guarantee a consistent approach to application rollout and reliability that ensures that a proliferation of microservice teams does not also result in a proliferation of different approaches

to service production lifecycle and operations.

Separation of Concerns for Consistency and Scaling

In addition to the consistency that Kubernetes brings to operations, the decoupling and separation of concerns produced by the Kubernetes stack lead to significantly greater consistency for the lower levels of your infrastructure. This enables your operations function to scale to managing many machines with a single small, focused team. We have talked at length about the decoupling of application container and machine/operating system (OS), but an important aspect of this decoupling is that the container orchestration API becomes a crisp contract that separates the responsibilities of the application operator from the cluster orchestration operator. We call this the "not my monkey, not my circus" line. The application developer relies on the service-level agreement (SLA) delivered by the container orchestration API, without worrying about the details of how this SLA is achieved. Likewise, the container orchestration API reliability engineer focuses on delivering the orchestration API's SLA without worrying about the applications that are running on top of it.

This decoupling of concerns means that a small team running a Kubernetes cluster can be responsible for supporting hundreds or even thousands of teams running applications within that cluster (Figure 1-1). Likewise, a small team can be responsible for tens (or more) of clusters running around the world. It's important to note that the same decoupling of containers and OS enables the OS reliability engineers to focus on the SLA of the individual machine's OS. This becomes another line of separate responsibility, with the Kubernetes operators relying on the OS SLA, and the OS operators worrying solely about delivering that SLA. Again, this enables you to scale a small team of OS experts to a fleet of thousands of machines.

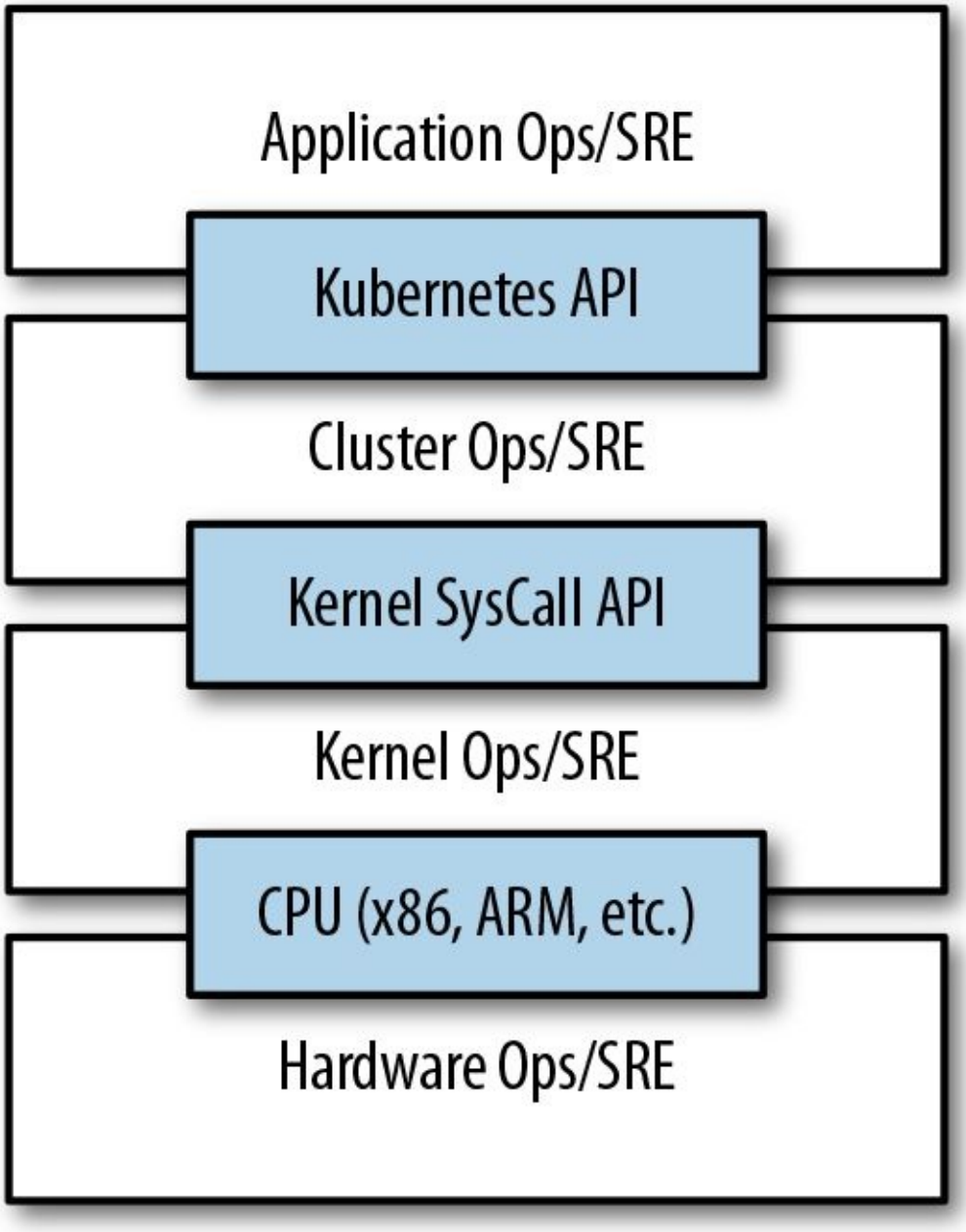

Figure 1-1. An illustration of how different operations teams are decoupled using APIs

Of course, devoting even a small team to managing an OS is beyond the scale of many organizations. In these environments, a managed Kubernetes-as-a-Service (KaaS) provided by a public cloud provider is a great option.

NOTE

At the time of writing, you can use managed KaaS on Microsoft Azure, with Azure Container Service, as well as on the Google Cloud Platform via the Google Container Engine (GCE). There is no equivalent service available on Amazon Web Services (AWS), though the `kops` project provides tools for easy installation and management of Kubernetes on AWS (see "Installing Kubernetes on Amazon Web Services").

The decision of whether to use KaaS or manage it yourself is one each user needs to make based on the skills and demands of their situation. Often for small organizations, KaaS provides an easy-to-use solution that enables them to focus their time and energy on building the software to support their work rather than managing a cluster. For a larger organization that can afford a dedicated team for managing its Kubernetes cluster, it may make sense to manage it yourself since it enables greater flexibility in terms of cluster capabilities and operations.

Abstracting Your Infrastructure

The goal of the public cloud is to provide easy-to-use, self-service infrastructure for developers to consume. However, too often cloud APIs are oriented around mirroring the infrastructure that IT expects, not the concepts (e.g., “virtual machines” instead of “applications”) that developers want to consume. Additionally, in many cases the cloud comes with particular details in implementation or services that are specific to the cloud provider. Consuming these APIs directly makes it difficult to run your application in multiple environments, or spread between cloud and physical environments.

The move to application-oriented container APIs like Kubernetes has two concrete benefits. First, as we described previously, it separates developers from specific machines. This not only makes the machine-oriented IT role easier, since machines can simply be added in aggregate to scale the cluster, but in the context of the cloud it also enables a high degree of portability since developers are consuming a higher-level API that is implemented in terms of the specific cloud infrastructure APIs.

When your developers build their applications in terms of container images and deploy them in terms of portable Kubernetes APIs, transferring your application between environments, or even running in hybrid environments, is simply a matter of sending the declarative config to a new cluster. Kubernetes has a number of plug-ins that can abstract you from a particular cloud. For example, Kubernetes services know how to create load balancers on all major public clouds as well as several different private and physical infrastructures. Likewise, Kubernetes `PersistentVolumes` and `PersistentVolumeClaims` can be used to abstract your applications away from specific storage implementations. Of course, to achieve this portability you need to avoid cloud-managed services (e.g., Amazon’s DynamoDB or Google’s Cloud Spanner), which means that you will be forced to deploy and manage open source storage solutions like Cassandra, MySQL, or MongoDB.

Putting it all together, building on top of Kubernetes’s application-oriented abstractions ensures that the effort that you put into building, deploying, and managing your application is truly portable across a wide variety of

environments.

Efficiency

In addition to the developer and IT management benefits that containers and Kubernetes provide, there is also a concrete economic benefit to the abstraction. Because developers no longer think in terms of machines, their applications can be colocated on the same machines without impacting the applications themselves. This means that tasks from multiple users can be packed tightly onto fewer machines.

Efficiency can be measured by the ratio of the useful work performed by a machine or process to the total amount of energy spent doing so. When it comes to deploying and managing applications, many of the available tools and processes (e.g., bash scripts, `apt` updates, or imperative configuration management) are somewhat inefficient. When discussing efficiency it's often helpful to think of both the cost of running a server and the human cost required to manage it.

Running a server incurs a cost based on power usage, cooling requirements, data center space, and raw compute power. Once a server is racked and powered on (or clicked and spun up), the meter literally starts running. Any idle CPU time is money wasted. Thus, it becomes part of the system administrator's responsibilities to keep utilization at acceptable levels, which requires ongoing management. This is where containers and the Kubernetes workflow come in. Kubernetes provides tools that automate the distribution of applications across a cluster of machines, ensuring higher levels of utilization than are possible with traditional tooling.

A further increase in efficiency comes from the fact that a developer's test environment can be quickly and cheaply created as a set of containers running in a personal view of a shared Kubernetes cluster (using a feature called *namespaces*). In the past, turning up a test cluster for a developer might have meant turning up three machines. With Kubernetes it is simple to have all developers share a single test cluster, aggregating their usage onto a much smaller set of machines. Reducing the overall number of machines used in turn drives up the efficiency of each system: since more of the resources (CPU, RAM, etc.) on each individual machine are used, the overall cost of each

container becomes much lower.

Reducing the cost of development instances in your stack enables development practices that might previously have been cost-prohibitive. For example, with your application deployed via Kubernetes it becomes conceivable to deploy and test every single commit contributed by every developer throughout your entire stack.

When the cost of each deployment is measured in terms of a small number of containers, rather than multiple complete virtual machines (VMs), the cost you incur for such testing is dramatically lower. Returning to the original value of Kubernetes, this increased testing also increases velocity, since you have both strong signals as to the reliability of your code as well as the granularity of detail required to quickly identify where a problem may have been introduced.

Summary

Kubernetes was built to radically change the way that applications are built and deployed in the cloud. Fundamentally, it was designed to give developers more velocity, efficiency, and agility. We hope the preceding sections have given you an idea of why you should deploy your applications using Kubernetes. Now that you are convinced of that, the following chapters will teach you *how* to deploy your application.

---

1 Brendan Burns et al., "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade," *ACM Queue* 14 (2016): 70–93, available at *http://bit.ly/2vIrL4S*.