Uncomplicate Data on Kubernetes
Make your data services scalable, available and secure on Kubernetes

Get Started Today
Managing Cloud Native Data on Kubernetes

Architecting Cloud Native Data Services Using Open Source Technology

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jeff Carpenter and Patrick McFadin
# Table of Contents

1. **Introduction to Cloud Native Data Infrastructure: Persistence, Streaming, and Batch Analytics** ........................................... 9
   Infrastructure Types .......................................................... 10
   What is Cloud Native Data? ................................................. 12
   More Infrastructure, More Problems .................................... 14
   Kubernetes Leading the Way ............................................... 16
      Managing Compute on Kubernetes .................................. 17
      Managing Network on Kubernetes .................................. 17
      Managing Storage on Kubernetes .................................. 18
   Cloud native data components ......................................... 18
   Looking forward ............................................................. 19
   Getting ready for the revolution ....................................... 21
      Adopt an SRE mindset .................................................. 21
      Embrace Distributed Computing .................................... 22
   Summary ....................................................................... 25

2. **Managing Data Storage on Kubernetes** ........................................ 27
   Docker, Containers, and State .......................................... 28
      Managing State in Docker ............................................ 29
   Bind mounts .................................................................. 29
   Volumes ....................................................................... 30
   Tmpfs Mounts ................................................................ 31
   Volume Drivers .............................................................. 32
   Kubernetes Resources for Data Storage ............................. 34
      Pods and Volumes ......................................................... 34
      PersistentVolumes ......................................................... 41
      PersistentVolumeClaims .............................................. 45
      StorageClasses ............................................................ 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes Storage Architecture</td>
<td>50</td>
</tr>
<tr>
<td>- Flexvolume</td>
<td>51</td>
</tr>
<tr>
<td>- Container Storage Interface (CSI)</td>
<td>51</td>
</tr>
<tr>
<td>- Container Attached Storage</td>
<td>53</td>
</tr>
<tr>
<td>- Container Object Storage Interface (COSI)</td>
<td>56</td>
</tr>
<tr>
<td>- Summary</td>
<td>58</td>
</tr>
<tr>
<td>3. Databases on Kubernetes the Hard Way</td>
<td>59</td>
</tr>
<tr>
<td>- The Hard Way</td>
<td>60</td>
</tr>
<tr>
<td>- Prerequisites for running data infrastructure on Kubernetes</td>
<td>61</td>
</tr>
<tr>
<td>- Running MySQL on Kubernetes</td>
<td>61</td>
</tr>
<tr>
<td>- ReplicaSets</td>
<td>62</td>
</tr>
<tr>
<td>- Deployments</td>
<td>64</td>
</tr>
<tr>
<td>- Services</td>
<td>68</td>
</tr>
<tr>
<td>- Accessing MySQL</td>
<td>71</td>
</tr>
<tr>
<td>- Running Apache Cassandra on Kubernetes</td>
<td>73</td>
</tr>
<tr>
<td>- StatefulSets</td>
<td>75</td>
</tr>
<tr>
<td>- Accessing Cassandra</td>
<td>86</td>
</tr>
<tr>
<td>- Summary</td>
<td>88</td>
</tr>
<tr>
<td>4. Automating Database Deployment on Kubernetes with Helm.</td>
<td>89</td>
</tr>
<tr>
<td>- Deploying Applications with Helm charts</td>
<td>90</td>
</tr>
<tr>
<td>- Using Helm to deploy MySQL</td>
<td>91</td>
</tr>
<tr>
<td>- How Helm Works</td>
<td>95</td>
</tr>
<tr>
<td>- Labels</td>
<td>97</td>
</tr>
<tr>
<td>- ServiceAccounts</td>
<td>98</td>
</tr>
<tr>
<td>- Secrets</td>
<td>98</td>
</tr>
<tr>
<td>- ConfigMaps</td>
<td>99</td>
</tr>
<tr>
<td>- Updating Helm Charts</td>
<td>101</td>
</tr>
<tr>
<td>- Uninstalling Helm charts</td>
<td>102</td>
</tr>
<tr>
<td>- Using Helm to deploy Apache Cassandra</td>
<td>103</td>
</tr>
<tr>
<td>- Affinity and Anti-Affinity</td>
<td>105</td>
</tr>
<tr>
<td>- Helm Limitations</td>
<td>108</td>
</tr>
<tr>
<td>- Summary</td>
<td>109</td>
</tr>
<tr>
<td>5. Automating Database Management on Kubernetes with Operators</td>
<td>111</td>
</tr>
<tr>
<td>- Extending the Kubernetes Control Plane</td>
<td>112</td>
</tr>
<tr>
<td>- Extending Kubernetes Clients</td>
<td>113</td>
</tr>
<tr>
<td>- Extending Kubernetes Master Node Components</td>
<td>113</td>
</tr>
<tr>
<td>- Extending Kubernetes Worker Node Components</td>
<td>115</td>
</tr>
<tr>
<td>- The Operator Pattern</td>
<td>115</td>
</tr>
<tr>
<td>- Controllers</td>
<td>116</td>
</tr>
</tbody>
</table>
6. **Streaming Data on Kubernetes** ............................................................. 143
   - Introduction to Streaming ......................................................... 143
   - Types of delivery ................................................................. 144
   - Delivery Guarantees ............................................................... 145
   - Feature scope ..................................................................... 146
   - The Role of Streaming in Kubernetes ..................................... 147
   - Streaming on Kubernetes with Apache Pulsar™ ....................... 150
     - Preparing Your Environment ............................................... 153
     - Securing Communications by Default with Cert-manager ..... 155
     - Using Helm to Deploy Apache Pulsar™ .......................... 159
   - Stream Analytics with Apache Flink™ .................................... 160
     - Deploying Apache Flink™ on Kubernetes ........................... 162
   - Conclusion ........................................................................... 165
CHAPTER 1

Introduction to Cloud Native Data Infrastructure: Persistence, Streaming, and Batch Analytics

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that this book’s code examples are available at https://github.com/data-on-k8s-book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

Do you work at solving data problems and find yourself faced with the need for modernization? Is your cloud native application limited to the use of microservices and service mesh? If you deploy applications on Kubernetes without including data, you haven't fully embraced cloud native. Every element of your application should embody the cloud native principles of scale, elasticity, self-healing, and observability, including how you handle data. Engineers that work with data are primarily concerned with stateful services, and this will be our focus: increasing your skills to manage data in Kubernetes. By reading this book, our goal is to enrich your journey to cloud native data. If you are just starting with cloud native applications, then there is no better time to include every aspect of the stack. This convergence is the future of how we will consume cloud resources.
So what is this future we are creating together?

For too long, data has been something that has lived outside of Kubernetes, creating a lot of extra effort and complexity. We will get into valid reasons for this, but now is the time to combine the entire stack to build applications faster at the needed scale. Based on current technology, this is very much possible. We've moved away from the past of deploying individual servers and towards the future where we will be able to deploy entire virtual data centers. Development cycles that once took months and years can now be managed in days and weeks. Open source components can now be combined into a single deployment on Kubernetes that is portable from your laptop to the largest cloud provider.

The open source contribution isn't a tiny part of this either. Kubernetes and the projects we talk about in this book are under the Apache License 2.0, unless otherwise noted. And for a good reason. If we build infrastructure that can run anywhere, we need a license model that gives us the freedom of choice. Open source is both free-as-in-beer and free-as-in-freedom, and both count when building cloud native applications on Kubernetes. Open source has been the fuel of many revolutions in infrastructure, and this is no exception.

That's what we are building. This is the near future reality of fully realized Kubernetes applications. The final component is the most important, and that is you. As a reader of this book, you are one of the people that will create this future. Creating is what we do as engineers. We continuously re-invent the way we deploy complicated infrastructure to respond to the increased demand. When the first electronic database system was put online in 1960 for American Airlines, you know there was a small army of engineers who made sure it stayed online and worked around the clock. Progress took us from mainframes to minicomputers, to microcomputers, and eventually to the fleet management we find ourselves doing today. Now, that same progression is continuing into cloud native and Kubernetes.

This chapter will examine the components of cloud native applications, the challenges of running stateful workloads, and the essential areas covered in this book. To get started, let's turn to the building blocks that make up data infrastructure.

**Infrastructure Types**

In the past twenty years, the approach to infrastructure has slowly forked into two areas that reflect how we deploy distributed applications, as shown in Figure 1-1.
Stateless services

These are services that maintain information only for the immediate life cycle of the active request—for example, a service for sending formatted shopping cart information to a mobile client. A typical example is an application server that performs the business logic for the shopping cart. However, the information about the shopping cart contents resides external to these services. They only need to be online for a short duration from request to response. The infrastructure used to provide the service can easily grow and shrink with little impact on the overall application. Scaling compute and network resources on-demand when needed. Since we are not storing critical data in the individual service, they can be created and destroyed quickly with little coordination. Stateless services are a crucial architecture element in distributed systems.

Stateful services

These services need to maintain information from one request to the next. Disks and memory store data for use across multiple requests. An example is a database or file system. Scaling stateful services is much more complex since the information typically requires replication for high availability, which creates the need for consistency and mechanisms to keep data in sync. These services usually have different scaling methods, both vertical and horizontal. As a result, they require different sets of operational tasks than stateless services.
In addition to how information is stored, we’ve also seen a shift towards developing systems that embrace automated infrastructure deployment. These recent advances include:

- Physical servers gave way to virtual machines that could be deployed and maintained easily.
- Virtual machines eventually became greatly simplified and focused on specific applications to what we now call containers.
- Containers have allowed infrastructure engineers to package an application’s operating system requirements into a single executable.

The use of containers has undoubtedly increased the consistency of deployments, which has made it easier to deploy and run infrastructure in bulk. Few systems emerged to orchestrate the explosion of containers like Kubernetes which is evident in the incredible growth. This speaks to how well it solves the problem. According to the Kubernetes documentation:

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Kubernetes was originally designed for stateless workloads, and that is what it has traditionally done best. Kubernetes has developed a reputation as a “platform for building platforms” in a cloud-native way. However, there’s a reasonable argument that a complete cloud-native solution has to take data into account. That’s the goal of this book: exploring how we make it possible to build cloud-native data solutions on Kubernetes. But first, let’s unpack what that term means.

**What is Cloud Native Data?**

Let’s begin defining the aspects of cloud native data that can help us with a final definition. First, let’s start with the definition of cloud native from the Cloud Native Computing Foundation (CNCF):

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

Note that this definition describes a goal state, desirable characteristics, and examples of technologies that embody both. Based on this formal definition, we can synthesize
the qualities that make a cloud native application differentiated from other types of deployments in terms of how it handles data. Let's take a closer look at these qualities.

**Scalability**

If a service can produce a unit of work for a unit of resources, then adding more resources should increase the amount of work a service can perform. Scalability is how we describe the service's ability to apply additional resources to produce additional work. Ideally, services should scale infinitely given an infinite amount of resources of compute, network and storage. For data this means scale without the need for downtime. Legacy systems required a maintenance period while adding new resources which all services had to be shutdown. With the needs of cloud native applications, downtime is no longer acceptable.

**Elasticity**

Where scale is adding resources to meet demand, elastic infrastructure is the ability to free those resources when no longer needed. The difference between scalability and elasticity is highlighted in Figure 1-2. Elasticity can also be called on-demand infrastructure. In a constrained environment such as a private data center, this is critical for sharing limited resources. For cloud infrastructure that charges for every resource used, this is a way to prevent paying for running services you don't need. When it comes to managing data, this means that we need capabilities to reclaim storage space and optimize our use, such as moving older data to less expensive storage tiers.

**Self-healing**

Bad things happen and when they do, how will your infrastructure respond? Self-healing infrastructure will re-route traffic, re-allocate resources, and maintain service levels. With larger and more complex distributed applications being deployed, this is an increasingly important attribute of a cloud-native application. This is what keeps you from getting that 3 AM wake-up call. For data, this means we need capabilities to detect issues with data such as missing data and data quality.

**Observability**

If something fails and you aren't monitoring it, did it happen? Unfortunately, the answer is not only yes, but that can be an even worse scenario. Distributed applications are highly dynamic and visibility into every service is critical for maintaining service levels. Interdependencies can create complex failure scenarios which is why observability is a key part of building cloud native applications. In data systems the volumes that are commonplace need efficient ways of monitoring the flow and state of infrastructure. In most cases, early warning for issues can help operators avoid costly downtime.
With all of the previous definitions in place, let’s try a definition that expresses these properties.

**Cloud Native Data**

Cloud Native Data approaches empower organizations that have adopted the cloud native application methodology to incorporate data holistically rather than employ the legacy of people, process, technology, so that data can scale up and down elastically, and promote observability and self-healing.

This is exemplified by containerized data, declarative data, data APIs, data-meshes, and cloud-native data infrastructure (that is, databases, streaming, and analytics technologies that are themselves architected as cloud-native applications).

In order for data infrastructure to keep parity with the rest of our application, we need to incorporate each piece. This includes automation of scale, elasticity and self-healing, APIs are needed to decouple services and increase developer velocity, and also the ability to observe the entire stack of your application to make critical decisions. Taken as a whole, your application and data infrastructure should appear as one unit.

**More Infrastructure, More Problems**

Whether your infrastructure is in a cloud, on-premises, or both (commonly referred to as hybrid), you could spend a lot of time doing manual configuration. Typing things into an editor and doing incredibly detailed configuration work requires deep knowledge of each technology. Over the past twenty years, there have been significant advances in the DevOps community to code and how we deploy our infrastructure. This is a critical step in the evolution of modern infrastructure. DevOps has kept us ahead of the scale required, but just barely. Arguably, the same amount of knowledge
is needed to fully script a single database server deployment. It’s just that now we can do it a million times over if needed with templates and scripts. What has been lacking is a connectedness between the components and a holistic view of the entire application stack. Foreshadowing: this is a problem that needed to be solved.

Like any good engineering problem, let’s break it down into manageable parts. The first is resource management. Regardless of the many ways we have developed to work at scale, fundamentally, we are trying to manage three things as efficiently as possible: compute, network and storage, as shown in Figure 1-3. These are the critical resources that every application needs and the fuel that’s burned during growth. Not surprisingly, these are also the resources that carry the monetary component to a running application. We get rewarded when we use the resources wisely and pay a literal high price if we don’t. Anywhere you run your application, these are the most primitive units. When on-prem, everything is bought and owned. When using the cloud, we’re renting.

![Figure 1-3. Fundamental resources of cloud applications: compute, network, and storage](image)

The second part of this problem is the issue of an entire stack acting as a single entity. DevOps has already given us many tools to manage individual components, but the connective tissue between them provides the potential for incredible efficiency. Similar to how applications are packaged for the desktop but working at data center scales. That potential has launched an entire community around cloud native applications. These applications are very similar to what we have always deployed. The difference is that modern cloud applications aren’t a single process with business logic. They are a complex coordination of many containerized processes that need to communicate securely and reliably. Storage has to match the current needs of the application but remains aware of how it contributes to the stability of the application. When we think of deploying stateless applications without data managed in the same control plane, it sounds incomplete because it is. Breaking up your application components into different control planes creates more complexity and is counter to the ideals of cloud native.
Kubernetes Leading the Way

As mentioned before, DevOps automation has kept us on the leading edge of meeting scale needs. Containers created the need for much better orchestration, and the answer has been Kubernetes. For operators, describing a complete application stack in a deployment file makes a reproducible and portable infrastructure. This is because Kubernetes has gone far beyond simply the deployment management that has been popular in the DevOps tool bag. The Kubernetes control plane applies the deployment requirement across the underlying compute, network, and storage to manage the entire application infrastructure lifecycle. The desired state of your application is maintained even when the underlying hardware changes. Instead of deploying virtual machines, we are now deploying virtual datacenters as a complete definition as shown in Figure 1-4.

Figure 1-4. Moving from virtual servers to virtual data centers

The rise in popularity of Kubernetes has eclipsed all other container orchestration tools used in DevOps. It has overtaken every other way we deploy infrastructure, and it will be even more so in the future. There's no sign of it slowing down. However, the bulk of early adoption was primarily in stateless services.

Managing data infrastructure at a large scale was a problem well before the move to containers and Kubernetes. Stateful services like databases took a different track parallel to the Kubernetes adoption curve. Many recommended that Kubernetes was the wrong way to run stateful services based on an architecture that favored ephemeral workloads. That worked until it didn't and is now driving the needed changes in Kubernetes to converge the application stack.

So what are the challenges of stateful services? Why has it been hard to deploy data infrastructure with Kubernetes? Let's consider each component of our infrastructure.
Managing Compute on Kubernetes

In data infrastructure, counting on Moore's law has made upgrading a regular event. If you aren't familiar, Moore's law predicted that computing capacity doubles every 18 months. If your requirements double every 18 months, you can keep up by replacing hardware. Eventually, raw compute power started leveling out. Vendors started adding more processors and cores to keep up with Moore's law, leading to single server resource sharing with virtual machines and containers. Enabling us to tap into the vast pools of computing power left stranded in islands of physical servers. Kubernetes expanded the scope of compute resource management by considering the total datacenter as one large resource pool across multiple physical devices.

Sharing compute resources with other services has been somewhat taboo in the data world. Data workloads are typically resource intensive, and the potential of one service impacting another (known as the noisy neighbor problem) has led to policies of keeping them isolated from other workloads. This one-size fits all approach eliminates the possibility for more significant benefits. First is the assumption that all data service resource requirements are the same. Apache Pulsar™ brokers can have far fewer requirements than an Apache Spark™ worker, and neither are similar to a sizeable MySQL instance used for OLAP reporting. Second, the ability to decouple your underlying hardware from running applications gives operators a lot of undervalued flexibility. Cloud native applications that need scale, elasticity, and self-healing need what Kubernetes can deliver. Data is no exception.

Managing Network on Kubernetes

Building a distributed application, by nature, requires a reliable and secure network. Cloud native applications increase the complexity of adding and subtracting services making dynamic network configuration a new requirement. Kubernetes manages all of this inside of your virtual data center automatically. When new services come online, it’s like a virtual network team springs to action. IP addresses are assigned, routes are created, DNS entries are added, then the virtual security team ensures firewall rules are in place, and when asked, TLS certificates provide end-to-end encryption.

Data infrastructure tends to be far less dynamic than something like microservices. A fixed IP with a hostname has been the norm for databases. Analytic systems like Apache Flink™ are dynamic in processing but have fixed hardware addressing assignments. Quality of service is typically at the top of the requirements list and, as a result, the desire for dedicated hardware and dedicated networks has turned administrators off of Kubernetes.

The advantage of data infrastructure running in Kubernetes is less about the past requirements and more about what’s needed for the future. Scaling resources dynamically can create a waterfall of dependencies. Automation is the only way to maintain
clean and efficient networks, which are the lifeblood of distributed stateless systems. The future of cloud native applications will only include more components and new challenges such as where applications run. We can add regulatory compliance and data sovereignty to previous concerns about latency and throughput. The declarative nature of Kubernetes networks make it a perfect fit for data infrastructure.

Managing Storage on Kubernetes

Any service that provides persistence or analytics over large volumes of data will need the right kind of storage device. Early versions of Kubernetes considered storage a basic commodity part of the stack and assumed that most workloads were ephemeral. For data, this was a huge mismatch. If your Postgres data files get deleted every time a container is moved, that just doesn't work. Additionally, implementing the underlying block storage can be a broad spectrum. From high performance NVMe disks to old 5400 RPM spinning disks. You may not know what you'll get. Thankfully this was an essential focus of Kubernetes over the past few years and has been significantly improved.

With the addition of features like Storage Classes, it is possible to address specific requirements for performance or capacity or both. With automation, we can avoid the point when you don't have enough of either. Avoiding surprises is the domain of capacity management—both initializing the needed capacity and growing when required. When you run out of capacity in your storage, everything grinds to a halt.

Coupling the distributed nature of Kubernetes with data storage opens up more possibilities for self healing. Automated backups and snapshots keep you ready for potential data loss scenarios. Placing compute and storage together to minimize hardware failure risks and automatic recovery to the desired state when the inevitable failure occurs. All of which makes the data storage aspects of Kubernetes much more attractive.

Cloud native data components

Now that we have defined the resources consumed in cloud native applications let's clarify the types of data infrastructure that powers them. Instead of a comprehensive list of every possible product, we'll break them down into larger buckets with similar characteristics.

Persistence

This is probably assumed when we talk about data infrastructure. Systems that store data and provide access by some method of a query. Relational databases like MySQL and Postgres. NoSQL systems like Cassandra and MongoDB. In the world of Kubernetes these have been the strongest, last holdouts due to the strictest resource requirements. This has been for good reasons too. Databases are
usually critical to a running application and central to every other part of the system.

**Streaming**

The most basic function of streaming is facilitating the high-speed movement of data from one point to another. Streaming systems provide a variety of delivery semantics based on a use case. In some cases, data can be delivered to many clients or when strict controls are needed, delivered only once. A further enhancement of streaming is the addition of processing. Altering or enhancing data while in mid-transport. The need for faster insights into data has propelled streaming analytics into mission critical status catching up with persistence systems for importance. Examples of streaming systems that move data are Apache Flink™ and Apache Kafka™, where processing system examples are Apache Flink™ and Apache Storm™.

**Batch Analytics**

One of the first big data problems. Analyzing large sets of data to gain insights or re-purpose into new data. Apache Hadoop™ was the first large scale system for batch analytics that set the expectations around using large volumes of compute and storage, coordinated in a way to produce a final result. Typically, these are issued as jobs distributed throughout the cluster which is something that is found in Apache Spark™. The concern with costs can be much more prevalent in these systems due to the sheer volume of resources needed. Orchestration systems help mitigate the costs by intelligent allocation.

**Looking forward**

There is a very compelling future with cloud native data, both with what we have available today and what we can have in the future. The path we take between those two points is up to us: the community of people responsible for data infrastructure. Just as we have always done, we see a new challenge and take it on. There is plenty for everyone to do here, but the result could be pretty amazing and raise the bar, yet again.

---

**A call for databases to modernize on Kubernetes**

*With Rick Vasquez, Senior Director, Strategic Initiatives, Western Digital*

Kubernetes is the catalyst for this current macro trend of change. Data infrastructure has to run the same as the rest of the application infrastructure. In a conference talk, Rick Vasquez, a leader in data infrastructure for years, wrote an open letter to the database community on the need for change. Here is a summary of that talk:
This is something for anyone working with databases in the 2020s. Kubernetes is leading the charge in building cloud native and distributed systems. Data systems aren't leveraging the full capacity and feature set possible if they were better integrated with Kubernetes. I'm a convert from the “you should never run a database in a container” way of thinking. Now I think we should be pushing everybody to have the main deployment in Kubernetes. My background has always been on scale enterprise use cases. I don't see this as a passing fad, I'm looking at the applicability to global scale for some of the largest companies in the world.

One line of thinking we need to overcome is treating Kubernetes like an operating system that enables other applications to run on it. That's the wrong way to look at running data workloads. If your system runs in a container, then of course it will work on Kubernetes, right? No! It will react to how the control plane deploys and runs your application, and it may or may not be what you want. What if data systems were more tightly integrated with Kubernetes and could offload functions to be handled by the Kubernetes control plane? Service discovery, load balancing, storage orchestration, automated rollouts, and rollbacks, automated bin packing, self-healing, secret and config management are all powerful things that allow for you to have a consistent developer and SRE experience. The name of the game with Kubernetes is driving consistency. You can use Kubernetes to become globally consistent across all your deployments and do them the same way over and over. But that needs to include database systems. Imagine if you have Postgres, MongoDB, MySQL, or Cassandra and it was built natively on Kubernetes. What would you do?

Having the access to use different storage tiers, either local or remote disk. All of it declarative in some configuration objects. I want to configure that in and with the database. If I’m using MySQL, I want logs to be on the local disk, because I don't want any bottlenecks. I want certain tables to be on a slower disk that may be over the network. And, I want the last seven days of data to be in hot, local NVMe disk. Using every single bit of capacity that you have with replicas actually doing things like off-loading reads or multiple write nodes, and one big aggregate for analytics. All of those things should be possible with a Kubernetes based deployment with a cloud native database.

Databases don't reason about or have an opinion about how big they are. If you make it bigger, it just needs more resources. You can set up auto-scaling to get you bigger or horizontal scaling. What happens whenever you want to use the true elasticity that's given to you by Kubernetes? It's not just the scale up and out. It's the scale back and down! Why don't databases just do that? That is so important to maximize the value that you're getting out of a Kubernetes based deployment or more broadly, a cloud native based deployment. We have a lot of work to do but the future is worth it.

This talk was specifically about databases, but we can extrapolate his call to action for our data infrastructure running on Kubernetes. Unlike deploying a data application
on physical servers, introducing the Kubernetes control plane requires a conversation with the services it runs.

**Getting ready for the revolution**

As engineers that create and run data infrastructure, we have to be ready for the changes coming. Both in how we operate and the mindset we have about the role of data infrastructure. The following sections are meant to describe what you can do to be ready for the future of cloud native data running in Kubernetes.

**Adopt an SRE mindset**

The role of Site Reliability Engineer (SRE) has grown with the adoption of cloud native methodologies. If we intend our infrastructure to converge, we as data infrastructure engineers must learn new skills and adopt new practices.

*Site reliability engineering* is a set of principles and practices that incorporates aspects of software engineering and applies them to infrastructure and operations problems. The main goals are to create scalable and highly reliable software systems. Site reliability engineering is closely related to DevOps, a set of practices that combine software development and IT operations, and SRE has also been described as a specific implementation of DevOps.

Deploying data infrastructure has been primarily concerned with the specific components deployed - the “what.” For example, you may find yourself focused on deploying MySQL at scale or using Apache Spark to analyze large volumes of data. Adopting an SRE mindset means going beyond what you are deploying and putting a greater focus on the how. How will all of the pieces work together to meet the goals of the application? A holistic view of a deployment considers how each piece will interact, the required access including security, and the observability of every aspect to ensure meeting service levels.

If your current primary or secondary role is Database Administrator, there is no better time to make the transition. The trend on LinkedIn shows a year-over-year decrease in the DBA role and a massive increase for SREs. Engineers that have learned the skills required to run critical database infrastructure have an essential baseline that translates into what’s needed to manage cloud native data. These include:

- Availability
- Latency
- Change Management
- Emergency response
• Capacity Management

New skills need to be added to this list to become better adapted to the more significant responsibility of the entire application. These are skills you may already have, but they include:

**CI/CD pipelines**

Embrace the big picture of taking code from repository to production. There’s nothing that accelerates application development more in an organization. Continuous Integration (CI) builds new code into the application stack and automates all testing to ensure quality. Continuous Deployment (CD) takes the fully tested and certified builds and automatically deploys them into production. Used in combination (Pipeline), organizations can drastically increase developer velocity and productivity.

**Observability**

Monitoring is something anyone with experience in infrastructure is familiar with. In the “what” part of DevOps you know services are healthy and have the information needed to diagnose problems. Observability expands monitoring into the “how” of your application by considering everything as a whole. For example, tracing the source of latency in a highly distributed application by giving insight into every hop data takes.

**Knowing the code**

When things go bad in a large distributed application it’s not always a process failure. In many cases, it could be a bug in the code or subtle implementation detail. Being responsible for the entire health of the application, you will need to understand the code that is executing in the provided environment. Properly implemented observability will help you find problems and that includes the software instrumentation. SREs and development teams need to have clear and regular communication and code is common ground.

**Embrace Distributed Computing**

Deploying your applications in Kubernetes means embracing all of what distributed computing offers. When you are accustomed to single system thinking, it can be a hard transition. Mainly in the shift in thinking around expectations and understanding where problems crop up. For example, with every process contained in a single system, latency will be close to zero. It’s not what you have to manage. CPU and memory resources are the primary concern there. In the 1990s, Sun Microsystems was leading in the growing field of distributed computing and published this list of common fallacies:

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn’t change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

These items most likely have an interesting story behind them where somebody assumed one of these fallacies and found themselves very disappointed. The result wasn’t what they expected and endless hours were lost trying to figure out the wrong problem.

Embracing distributed methodologies is worth the effort in the long run. It is how we build large scale applications and will be for a very long time. The challenge is worth the reward, and for those of us who do this daily, it can be a lot of fun too! Kubernetes applications will test each of these fallacies given its default distributed nature. When you plan your deployment, considering things such as the cost of transport from one place to another or latency implications. They will save you a lot of wasted time and re-design.

**Principles of Cloud Native Data Infrastructure**

As engineering professionals, we seek standards and best-practices to build upon. To make data the most “cloud native” it can be, we need to embrace everything Kubernetes offers. A truly cloud native approach means adopting key elements of the Kubernetes design paradigm and building from there. An entire cloud native application that includes data must be one that can run effectively on Kubernetes. Let’s explore a few Kubernetes design principles that point the way.

**Principle 1: Leverage compute, network, and storage as commodity APIs**

One of the keys to the success of cloud computing is the commoditization of computation, networking, and storage as resources we can provision via simple APIs. Consider this sampling of AWS services.

**Compute**

We allocate virtual machines through EC2 and Autoscaling Groups (ASGs).

**Network**

We manage traffic using Elastic Load Balancers (ELB), Route 53, and VPC peering.
Storage
We persist data using options such as the Simple Storage Service (S3) for long-term object storage, or Elastic Block Storage (EBS) volumes for our compute instances.

Kubernetes offers its own APIs to provide similar services for a world of containerized applications:

Compute
Pods, Deployments, and ReplicaSets manage the scheduling and life cycle of containers on computing hardware.

Network
Services and Ingress expose a container’s networked interfaces.

Storage
PersistentVolumes and StatefulSets enable flexible association of containers to storage.

Kubernetes resources promote the portability of applications across Kubernetes distributions and service providers. What does this mean for databases? They are simply applications that leverage computation, networking, and storage resources to provide the services of data persistence and retrieval:

Compute
A database needs sufficient processing power to process incoming data and queries. Each database node is deployed as a pod and grouped in StatefulSets, enabling Kubernetes to manage scaling out and scaling in.

Network
A database needs to expose interfaces for data and control. We can use Kubernetes Services and Ingress Controllers to expose these interfaces.

Storage
A database uses persistent volumes of a specified storage class to store and retrieve data.

Thinking of databases in terms of their compute, network, and storage needs removes much of the complexity involved in deployment on Kubernetes.

Principle 2: Separate the control and data planes
Kubernetes promotes the separation of control and data planes. The Kubernetes API server is the key data plane interface used to request computing resources, while the control plane manages the details of mapping those requests onto an underlying IaaS platform.
We can apply this same pattern to databases. For example, a database data plane consists of ports exposed for clients, and for distributed databases, ports used for communication between database nodes. The control plane includes interfaces provided by the database for administration and metrics collection and tooling that performs operational maintenance tasks. Much of this capability can and should be implemented via the Kubernetes operator pattern. Operators define custom resources (CRDs) and provide control loops that observe the state of those resources and take actions to move them toward the desired state, helping extend Kubernetes with domain-specific logic.

**Principle 3: Make observability easy**

The three pillars of observable systems are logging, metrics, and tracing. Kubernetes provides a great starting point by exposing the logs of each container to third-party log aggregation solutions. There are multiple solutions available for metrics, tracing, and visualization, and we'll explore several of them in this book.

**Principle 4: Make the default configuration secure**

Kubernetes networking is secure by default: ports must be explicitly exposed in order to be accessed externally to a pod. This sets a valuable precedent for database deployment, forcing us to think carefully about how each control plane and data plane interface will be exposed and which interfaces should be exposed via a Kubernetes Service. Kubernetes also provides facilities for secret management which can be used for sharing encryption keys and configuring administrative accounts.

**Principle 5: Prefer declarative configuration**

In the Kubernetes declarative approach, you specify the desired state of resources, and controllers manipulate the underlying infrastructure in order to achieve that state. Operators for data infrastructure can manage the details of how to scale up intelligently, for example, deciding how to reallocate shards or partitions when scaling out additional nodes or selecting which nodes to remove to scale down elastically.

The next generation of operators should enable us to specify rules for stored data size, number of transactions per second, or both. Perhaps we'll be able to specify maximum and minimum cluster sizes, and when to move less frequently used data to object storage. This will allow for more automation and efficiency in our data infrastructure.

**Summary**

At this point, we hope you are ready for the exciting journey in the pages ahead. The move to cloud native applications must include data, and to do this, we will leverage Kubernetes to include stateless and stateful services. This chapter covered cloud
native data infrastructure that can scale elastically and resist any downtime due to system failures and how to build these systems. We as engineers must embrace the principles of cloud native infrastructure and in some cases, learn new skills. Congratulations, you have begun a fantastic journey into the future of building cloud native applications. Turn the page, and let’s go!
A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that this book’s code examples are available at https://github.com/data-on-k8s-book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

There is no such thing as a stateless architecture. All applications store state somewhere
—Alex Chircop, CEO, StorageOS

In the previous chapter, we painted a picture of a possible near future with powerful, stateful, data-intensive applications running on Kubernetes. To get there, we're going to need data infrastructure for persistence, streaming, and analytics, and to build out this infrastructure, we'll need to leverage the primitives that Kubernetes provides to help manage the three commodities of cloud computing: compute, network, and storage. In the next several chapters we begin to look at these primitives, starting with storage, in order to see how they can be combined to create the data infrastructure we need.

To echo the point raised by Alex Chircop in the quote above, all applications must store their state somewhere, which is why we'll focus in this chapter on the basic abstractions Kubernetes provides for interacting with storage. We'll also look at the emerging innovations being offered by storage vendors and open source projects that
are creating storage infrastructure for Kubernetes that itself embodies cloud-native principles.

Let’s start our exploration with a look at managing persistence in containerized applications in general and use that as a jumping off point for our investigation into data storage on Kubernetes.

**Docker, Containers, and State**

The problem of managing state in distributed, cloud-native applications is not unique to Kubernetes. A quick search will show that stateful workloads have been an area of concern on other container orchestration platforms such as Mesos and Docker Swarm. Part of this has to do with the nature of container orchestration, and part is driven by the nature of containers themselves.

First, let’s consider containers. One of the key value propositions of containers is their ephemeral nature. Containers are designed to be disposable and replaceable, so they need to start quickly and use as few resources for overhead processing as possible. For this reason, most container images are built from base images containing streamlined, Linux-based, open-source operating systems such as Ubuntu, that boot quickly and incorporate only essential libraries for the contained application or microservice. As the name implies, containers are designed to be self-contained, incorporating all their dependencies in immutable images, while their configuration and data is externalized. These properties make containers portable so that we can run them anywhere a compatible container runtime is available.

As shown in Figure 2-1, containers require less overhead than traditional virtual machines, which run a guest operating system per virtual machine, with a hypervisor layer to implement system calls onto the underlying host operating system.
Although containers have made applications more portable, it’s proven a bigger challenge to make their data portable. We’ll examine the idea of portable data sets in Chapter 12. Since a container itself is ephemeral, any data that is to survive beyond the life of the container must by definition reside externally. The key feature for a container technology is to provide mechanisms to link to persistent storage, and the key feature for a container orchestration technology is the ability to schedule containers in such a way that they can access persistent storage efficiently.

Managing State in Docker

Let’s take a look at the most popular container technology, Docker, to see how containers can store data. The key storage concept in Docker is the volume. From the perspective of a Docker container, a volume is a directory that can support read-only or read-write access. Docker supports the mounting of multiple different data stores as volumes. We’ll introduce several options so we can later note their equivalents in Kubernetes.

Bind mounts

The simplest approach for creating a volume is to bind a directory in the container to a directory on the host system. This is called a bind mount, as shown in Figure 2-2.
When starting a container within Docker, you specify a bind mount with the \(-v\) option and the local filesystem path and container path to use. For example, you could start an instance of the Nginx web server, and map a local project folder from your development machine into the container. This is a command you can test out in your own environment if you have Docker installed:

```
docker run -it --rm --name web -v ~/site-content:/usr/share/nginx/html nginx
```

If the local path directory does not already exist, the Docker runtime will create it. Docker allows you to create bind mounts with read-only or read-write permissions. Because the volume is represented as a directory, the application running in the container can put anything that can be represented as a file into the volume - even a database.

Bind mounts are quite useful for development work. However, using bind mounts is not suitable for a production environment since this leads to a container being dependent on a file being present in a specific host. This might be fine for a single machine deployment, but production deployments tend to be spread across multiple hosts. Another concern is the potential security hole that is presented by opening up access from the container to the host filesystem. For these reasons, we need another approach for production deployments.

**Volumes**

The preferred option within Docker is to use volumes. Docker volumes are created and managed by Docker under a specific directory on the host filesystem. The Docker volume create command is used to create a volume. For example, you might create a volume called site-content to store files for a website:
If no name is specified, Docker assigns a random name. After creation, the resulting volume is available to mount in a container using the form `-v VOLUME-NAME:CONTAINER-PATH`. For example, you might use a volume like the one just created to allow an Nginx container to read the content, while allowing another container to edit the content, using the to option:

```bash
docker run -it --rm -d --name web -v site-content:/usr/share/nginx/html:ro nginx
```

**Docker Volume mount syntax**

Docker also supports a `--mount` syntax which allows you to specify the source and target folders more explicitly. This notation is considered more modern, but it is also more verbose. The syntax shown above is still valid and is the more commonly used syntax.

As implied above, a Docker volume can be mounted in more than one container at once, as shown in Figure 2-3.

![Figure 2-3. Creating Docker Volumes to share data between containers on the host](image)

The advantage of using Docker volumes is that Docker manages the filesystem access for containers, which makes it much simpler to enforce capacity and security restrictions on containers.

**Tmpfs Mounts**

Docker supports two types of mounts that are specific to the operating system used by the host system: tmpfs (or “temporary filesystem”) and named pipes. Named pipes are available on Docker for Windows, but since they are typically not used in K8s, we won’t give much consideration to them here.
Tmpfs mounts are available when running Docker on Linux. A tmpfs mount exists only in memory for the lifespan of the container, so the contents are never present on disk, as shown in Figure 2-4. Tmpfs mounts are useful for applications that are written to persist a relatively small amount of data, especially sensitive data that you don’t want written to the host filesystem. Because the data is stored in memory, there is a side benefit of faster access.

![Diagram of Tmpfs Mount](image)

*Figure 2-4. Creating a temporary volume using Docker tmpfs*

To create a tmpfs mount, you use the `docker run --tmpfs` option. For example, you could use a command like this to specify a tmpfs volume to store Nginx logs for a webserver processing sensitive data:

```
docker run -it --rm -d --name web -tmpfs /var/log/nginx nginx
```

The `--mount` option may also be used for more control over configurable options.

## Volume Drivers

The Docker Engine has an extensible architecture which allows you to add customized behavior via plugins for capabilities including networking, storage, and authorization. Third-party storage plugins are available for multiple open-source and commercial providers, including the public clouds and various networked file systems. Taking advantage of these involves installing the plugin with Docker engine and then specifying the associated volume driver when starting Docker containers using that storage, as shown in Figure 2-5.
Figure 2-5. Using Docker Volume Drivers to access networked storage

For more information on working with the various types of volumes supported in Docker, see the Docker Storage documentation, as well as the documentation for the `docker run` command.

**File, Block, and Object Storage**

In our modern era of cloud architectures, the three main formats in which storage is traditionally provided to applications are files, blocks, and objects. Each of these store and provide access to data in different ways.

- File storage represents data as a hierarchy of folders, each of which can contain files. The file is the basic unit of access for both storage and retrieval. The root directory that is to be accessed by a container is mounted into the container filesystem such that it looks like any other directory. Each of the public clouds provides their own file storage, for example Google Cloud Filestore, or Amazon Elastic Filestore. **Gluster** is an open-source distributed file system. Many of these systems are compatible with the Network File System (NFS), a distributed file system protocol invented at Sun Microsystems dating back to 1984 that is still in common use.

- Block storage organizes data in chunks and allocates those chunks across a set of managed volumes. When you provide data to a block storage system, it divides it up into chunks of varying sizes and distributes those chunks in order to use the
underlying volumes the most efficiently. When you query a block storage system, it retrieves the chunks from their various locations and provides the data back to you. This flexibility makes block storage a great solution when you have a heterogeneous set of storage devices available. Block storage doesn't provide a lot of metadata handling, which can place more burden on the application.

- Object storage organizes data in units known as objects. Each object is referenced by a unique identifier or “key”, and can support rich metadata tagging that enables searching. Objects are organized in buckets. This flat, non-hierarchical organization makes object storage easy to scale. Amazon's Simple Storage Service (S3) is the canonical example of object storage and most object storage products will claim compatibility with the S3 API.

If you're tasked with building or selecting data infrastructure, you'll want to understand the strengths and weaknesses of each of these patterns.

Kubernetes Resources for Data Storage

Now that you understand basic concepts of container and cloud storage, let's see what Kubernetes brings to the table. In this section, we'll introduce some of the key Kubernetes concepts or “resources” in the API for attaching storage to containerized applications. Even if you are already somewhat familiar with these resources, you'll want to stay tuned, as we'll take a special focus on how each one relates to stateful data.

Pods and Volumes

One of the first Kubernetes resources new users encounter is the pod. The pod is the basic unit of deployment of a Kubernetes workload. A pod provides an environment for running containers, and the Kubernetes control plane is responsible for deploying pods to Kubernetes worker nodes. The Kubelet is a component of the Kubernetes control plane that runs on each worker node. It is responsible for running pods on a node, as well as monitoring the health of these pods and the containers inside them. These elements are summarized in Figure 2-6.

While a pod can contain multiple containers, the best practice is for a pod to contain a single application container, along with optional additional helper containers, as shown in the figure. These helper containers might include init containers that run prior to the main application container in order to perform configuration tasks, or sidecar containers that run alongside the main application container to provide helper services such as observability or management. In future chapters we'll demonstrate how data infrastructure deployments can take advantage of these architectural patterns.
Now let’s consider how persistence is supported within this pod architecture. As with Docker, the “on disk” data in a container is lost when a container crashes. The kubelet is responsible for restarting the container, but this new container is really a replacement for the original container - it will have a distinct identity, and start with a completely new state.

In Kubernetes, the term *volume* is used to represent access to storage within a pod. By using a volume, the container has the ability to persist data that will outlive the container (and potentially the pod as well, as we’ll see shortly). A volume may be accessed by multiple containers in a pod. Each container has its own *volumeMount* within the pod that specifies the directory to which it should be mounted, allowing the mount point to differ between containers.

There are multiple cases where you might want to share data between multiple containers in a pod:

- An init container creates a custom configuration file for the particular environment that the application container mounts in order to obtain configuration values.
- The application pod writes logs, and a sidecar pod reads those logs to identify alert conditions that are reported to an external monitoring tool.

However, you’ll likely want to avoid situations in which multiple containers are writing to the same volume, because you’ll have to ensure the multiple writers don’t conflict - Kubernetes does not do that for you.
Preparing to run sample code

The examples in this chapter (and the rest of the book) assume you have access to a running Kubernetes cluster. For the examples in this chapter, a development cluster on your local machine such as Kind, K3s, or Docker Desktop should be sufficient. The source code used in this section is located at Kubernetes Storage Examples.

```yaml
apiVersion: v1
category: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-app
      image: nginx
      volumeMounts:
        - name: web-data
          mountPath: /app/config
  volumes:
    - name: web-data

Notice the two parts of the configuration: the volume is defined under spec.volumes, and the usage of the volumes is defined under spec.containers.volumeMounts. First, the name of the volume is referenced under the volumeMounts, and the directory where it is to be mounted is specified by the mountPath. When declaring a pod specification, volumes and volume mounts go together. For your configuration to be valid, a volume must be declared before being referenced, and a volume must be used by at least one container in the pod.

You may have also noticed that the volume only has a name. You haven’t specified any additional information. What do you think this will do? You could try this out for yourself by using the example source code file nginx-pod.yaml or cutting and pasting the configuration above to a file with that name, and executing the kubectl command against a configured Kubernetes cluster:

```bash
cubectl apply -f nginx-pod.yaml
```

You can get more information about the pod that was created using the kubectl get pod command, for example:

```bash
cubectl get pod my-pod -o yaml | grep -A 5 " volumes:"
```

And the results might look something like this:

```yaml
volumes:
  - emptyDir: {}  
    name: web-data  
    name: default-token-2fp89
```
As you can see, Kubernetes supplied some additional information when creating the requested volume, defaulting it to a type of `emptyDir`. Other default attributes may differ depending on what Kubernetes engine you are using and we won’t discuss them further here.

There are several different types of volumes that can be mounted in a container, let’s have a look.

**Ephemeral volumes**

You’ll remember tmpfs volumes from our discussion of Docker volumes above, which provide temporary storage for the lifespan of a single container. Kubernetes provides the concept of an **ephemeral volumes**, which is similar, but at the scope of a pod. The `emptyDir` introduced in the example above is a type of ephemeral volume.

Ephemeral volumes can be useful for data infrastructure or other applications that want to create a cache for fast access. Although they do not persist beyond the lifespan of a pod, they can still exhibit some of the typical properties of other volumes for longer-term persistence, such as the ability to snapshot. Ephemeral volumes are slightly easier to set up than PersistentVolumes because they are declared entirely inline in the pod definition without reference to other Kubernetes resources. As you will see below, creating and using PersistentVolumes is a bit more involved.

**Other ephemeral storage providers**

Some of the in-tree and CSI storage drivers we’ll discuss below that provide PersistentVolumes also provide an ephemeral volume option. You’ll want to check the documentation of the specific provider in order to see what options are available.

**Configuration volumes**

Kubernetes provides several constructs for injecting configuration data into a pod as a volume. These volume types are also considered ephemeral in the sense that they do not provide a mechanism for allowing applications to persist their own data.

These volume types are relevant to our exploration in this book since they provide a useful means of configuring applications and data infrastructure running on Kubernetes. We’ll describe each of them briefly:

**ConfigMap Volumes**

A ConfigMap is a Kubernetes resource that is used to store configuration values external to an application as a set of name-value pairs. For example, an application might require connection details for an underlying database such as an IP
address and port number. Defining these in a ConfigMap is a good way to externalize this information from the application. The resulting configuration data can be mounted into the application as a volume, where it will appear as a directory. Each configuration value is represented as a file, where the filename is the key, and the contents of the file contain the value. See the Kubernetes documentation for more information on mounting ConfigMaps as volumes.

**Secret Volumes**

A Secret is similar to a ConfigMap, only it is intended for securing access to sensitive data that requires protection. For example, you might want to create a secret containing database access credentials such as a username and password. Configuring and accessing Secrets is similar to using ConfigMap, with the additional benefit that Kubernetes helps decrypt the secret upon access within the pod. See the Kubernetes documentation for more information on mounting Secrets as volumes.

**Downward API Volumes**

The Kubernetes Downward API exposes metadata about pods and containers, either as environment variables or as volumes. This is the same metadata that is used by kubectl and other clients.

The available pod metadata includes the pod’s name, ID, namespace, labels, and annotations. The containerized application might wish to use the pod information for logging and metrics reporting, or to determine database or table names.

The available container metadata includes the requested and maximum amounts of resources such as CPU, memory, and ephemeral storage. The containerized application might wish to use this information in order to throttle its own resource usage. See the Kubernetes documentation for an example of injecting pod information as a volume.

**Hostpath volumes**

A hostPath volume mounts a file or directory into a pod from the Kubernetes worker node where it is running. This is analogous to the bind mount concept in Docker discussed above. Using a hostPath volume has one advantage over an emptyDir volume: the data will survive the restart of a pod.

However, there are some disadvantages to using hostPath volumes. First, in order for a replacement pod to access the data of the original pod, it will need to be restarted on the same worker node. While Kubernetes does give you the ability to control which node a pod is placed on using affinity, this tends to constrain the Kubernetes scheduler from optimal placement of pods, and if the node goes down for some reason, the data in the hostPath volume is lost. Second, similar to Docker bind mounts, there is a security concern with hostPath volumes in terms of allowing access to the
local filesystem. For these reasons, **hostPath** volumes are only recommended for development deployments.

**Cloud Volumes**

It is possible to create Kubernetes volumes that reference storage locations beyond just the worker node where a pod is running, as shown in Figure 2-7. These can be grouped into volume types that are provided by named cloud providers, and those that attempt to provide a more generic interface.

![Figure 2-7. Kubernetes pods directly mounting cloud provider storage](image)

These include the following:

- The **awsElasticBlockStore** volume type is used to mount volumes on Amazon Web Services (AWS) Elastic Block Store (EBS). Many databases use block storage as their underlying storage layer.
- The **gcePersistentDisk** volume type is used to mount Google Compute Engine (GCE) persistent disks (PD), another example of block storage.
- Two types of volumes are supported for Microsoft Azure: **azureDisk** for Azure Disk Volumes, and **azureFile** for Azure File Volumes
- For OpenStack deployments, the **cinder** volume type can be used to access OpenStack Cinder volumes

Usage of these types typically requires configuration on the cloud provider, and access from Kubernetes clusters is typically confined to storage in the same cloud.
region and account. Check your cloud provider’s documentation for additional details.

Additional Volume Providers

There are a number of additional volume providers that vary in the types of storage provided. Here are a few examples:

- The `fibreChannel` volume type can be used for SAN solutions implementing the FibreChannel protocol.
- The `gluster` volume type is used to access file storage using the Gluster distributed file system referenced above.
- An `iscsi` volume mounts an existing iSCSI (SCSI over IP) volume into your Pod.
- An `nfs` volume allows an existing NFS (Network File System) share to be mounted into a Pod.

We’ll examine more volume providers below that implement the Container Attached Storage pattern.

Table 2-1 provides a comparison of Docker and Kubernetes storage concepts we’ve covered so far.

Table 2-1. Table 2-1: Comparing Docker and Kubernetes storage options

<table>
<thead>
<tr>
<th>Type of Storage</th>
<th>Docker</th>
<th>Kubernetes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to persistent storage from various providers</td>
<td>Volume (accessed via Volume drivers)</td>
<td>Volume (accessed via in-tree or CSI drivers)</td>
</tr>
<tr>
<td>Access to host filesystem (not recommended for production)</td>
<td>Bind mount</td>
<td>Hostpath volume</td>
</tr>
<tr>
<td>Temporary storage available while container (or pod) is running</td>
<td>tmpfs</td>
<td>emptyDir and other ephemeral volumes</td>
</tr>
<tr>
<td>Configuration and environment data (read-only)</td>
<td>(no direct equivalent)</td>
<td>ConfigMap, Secret, Downward API</td>
</tr>
</tbody>
</table>

How do you choose a Kubernetes storage solution?

Given the number of storage options available, it can certainly be an intimidating task to try to determine what kind of storage you should use for your application. Along with determining whether you need file, block, or object storage, you’ll want to consider your latency and throughput requirements, as well as your expected storage volume. For example, If your read latency requirements are aggressive, you’ll most likely need a storage solution that keeps data in the same data center where it is accessed.

Next, you’ll want to consider any existing commitments or resources you have. Perhaps your organization has a mandate or bias toward using services from a preferred
cloud provider. The cloud providers will frequently provide cost incentives for using their services, but you'll want to trade this against the risk of lock-in to a specific service. Alternatively, you might have an investment in a storage solution in an on-premises data center that you need to leverage.

Overall, cost tends to be the overriding factor in choosing storage solutions, especially over the long term. Make sure your modeling includes not only the cost of the physical storage and any managed services, but also the operational cost involved in managing your chosen solution.

In this section, we’ve discussed how to use volumes to provide storage that can be shared by multiple containers within the same pod. While this is sufficient for some use cases, there are some needs this doesn’t address. A volume does not provide the ability to share storage resources between pods. The definition of a particular storage location is tied to the definition of the pod. Managing storage for individual pods doesn’t scale well as the number of pods deployed in your Kubernetes cluster increases.

Thankfully, Kubernetes provides additional primitives that help simplify the process of provisioning and mounting storage volumes for both individual pods and groups of related pods. We’ll investigate these concepts in the next several sections.

**PersistentVolumes**

The key innovation the Kubernetes developers have introduced for managing storage is the `persistent volume` subsystem. This subsystem consists of three additional Kubernetes resources that work together: PersistentVolumes, PersistentVolumeClaims, and StorageClasses. This allows you to separate the definition and lifecycle of storage from how it is used by pods, as shown in Figure 2-8:

- Cluster administrators define PersistentVolumes, either explicitly or by creating a StorageClass that can dynamically provision new PersistentVolumes.
- Application developers create PersistentVolumeClaims that describe the storage resource needs of their applications, and these PersistentVolumeClaims can be referenced as part of volume definitions in pods.
- The Kubernetes control plane manages the binding of PersistentVolumeClaims to PersistentVolumes.
Figure 2-8. PersistentVolumes, PersistentVolumeClaims, and StorageClasses

Let’s look first at the PersistentVolume resource (often abbreviated PV), which defines access to storage at a specific location. PersistentVolumes are typically defined by cluster administrators for use by application developers. Each PV can represent storage of the same types discussed in the previous section, such as storage offered by cloud providers, networked storage, or storage directly on the worker node, as shown in Figure 2-9. Since they are tied to specific storage locations, PersistentVolumes are not portable between Kubernetes clusters.

Local PersistentVolumes

The figure also introduces a PersistentVolume type called local, which represents storage mounted directly on a Kubernetes worker node such as a disk or partition.
Like `hostPath` volumes, a `local` volume may also represent a directory. A key difference between local and `hostPath` volumes is that when a pod using a `local` volume is restarted, the Kubernetes scheduler ensures the pod is rescheduled on the same node so that it can be attached to the same persistent state. For this reason, `local` volumes are frequently used as the backing store for data infrastructure that manages its own replication, as we’ll see in Chapter 4.

The syntax for defining a PersistentVolume will look familiar, as it is similar to defining a volume within a pod. For example, here is a YAML configuration file that defines a local PersistentVolume (source code):

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-volume
spec:
  capacity:
    storage: 3Gi
  accessModes:
  - ReadWriteOnce
local:
  path: /app/data
nodeAffinity:
  required:
    nodeSelectorTerms:
    - matchExpressions:
      - key: kubernetes.io/hostname
        operator: In
        values:
        - node1
```

As you can see, this code defines a `local` volume named `my-volume` on the worker node `node1`, 3 GB in size, with an access mode of `ReadWriteOnce`. The following access modes are supported for PersistentVolumes:

- **ReadWriteOnce** access allows the volume to be mounted for both reading and writing by a single pod at a time, and may not be mounted by other pods
- **ReadOnlyMany** access means the volume can be mounted by multiple pods simultaneously for reading only
- **ReadWriteMany** access allows the volume to be mounted for both reading and writing by many nodes at the same time
Choosing a volume access mode

The right access mode for a given volume will be driven by the type of workload. For example, many distributed databases will be configured with dedicated storage per pod, making ReadWriteOnce a good choice.

Besides capacity and access mode, other attributes for PersistentVolumes include:

- The `volumeMode`, which defaults to `Filesystem` but may be overridden to `Block`.
- The `reclaimPolicy` defines what happens when a pod releases its claim on this PersistentVolume. The legal values are `Retain`, `Recycle`, and `Delete`.
- A PersistentVolume can have a `nodeAffinity` which designates which worker node or nodes can access this volume. This is optional for most types, but required for the `local` volume type.
- The `class` attribute binds this PV to a particular StorageClass, which is a concept we’ll introduce below.
- Some PersistentVolume types expose `mountOptions` that are specific to that type.

Differences in volume options

Options differ between different volume types. For example, not every access mode or reclaim policy is accessible for every PersistentVolume type, so consult the documentation on your chosen type for more details.

You use the `kubectl describe persistentvolume` command (or `kubectl describe pv` for short) to see the status of the PersistentVolume:

```
$ kubectl describe pv my-volume
Name:              my-volume
Labels:            <none>
Annotations:       <none>
Finalizers:        
StorageClass:
Status:            Available
Claim:             
Reclaim Policy:    Retain
Access Modes:      RWo
VolumeMode:        Filesystem
Capacity:          3Gi
Node Affinity:     
    Required Terms: 
    Term 0:         kubernetes.io/hostname in [node1]
Message:           
```
The PersistentVolume has a status of Available when first created. A PersistentVolume can have multiple different status values:

- **Available** means the PersistentVolume is free, and not yet bound to a claim.
- **Bound** means the PersistentVolume is bound to a PersistentVolumeClaim, which is listed elsewhere in the `describe` output.
- **Released** means that an existing claim on the PersistentVolume has been deleted, but the resource has not yet been reclaimed, so the resource is not yet Available.
- **Failed** means the volume has failed its automatic reclamation.

Now that you've learned how storage resources are defined in Kubernetes, the next step is to learn how to use that storage in your applications.

**PersistentVolumeClaims**

As discussed above, Kubernetes separates the definition of storage from its usage. Often these tasks are performed by different roles: cluster administrators define storage, while application developers use the storage. PersistentVolumes are typically defined by the administrators and reference storage locations which are specific to that cluster. Developers can then specify the storage needs of their applications using PersistentVolumeClaims (PVCs) that Kubernetes uses to associate pods with a PersistentVolume that meets the specified criteria. As shown in Figure 2-10, a PersistentVolumeClaim is used to reference the various volume types we've introduced previously, including local PersistentVolumes, or external storage provided by cloud or networked storage vendors.
Figure 2-10. Accessing PersistentVolumes using PersistentVolumeClaims

Here's what the process looks like from an application developer perspective. First, you’ll create a PVC representing your desired storage criteria. For example, here’s a claim that requests 1GB of storage with exclusive read/write access (source code):

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-claim
spec:
  storageClassName: ""
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

One interesting thing you may have noticed about this claim is that the storageClassName is set to an empty string. We’ll explain the significance of this when we discuss StorageClasses below. You can reference the claim in the definition of a pod like this (source code):

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
```

46 | Chapter 2: Managing Data Storage on Kubernetes
As you can see, the PersistentVolume is represented within the pod as a volume. The volume is given a name and a reference to the claim. This is considered to be a volume of the persistentVolumeClaim type. As with other volumes, the volume is mounted into a container at a specific mount point, in this case into the main application Nginx container at the path `/app/data`.

A PVC also has a state, which you can see if you retrieve the status:

Name:          my-claim
Namespace:     default
StorageClass:  Bound
Volume:        my-volume
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
                pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      3Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Mounted By:    <none>
Events:        <none>

A PVC has one of two Status values: `Bound`, meaning it is bound to a volume (as is the case above), or `Pending`, meaning that it has not yet been bound to a volume. Typically a status of Pending means that no PV matching the claim exists.

Here's what's happening behind the scenes. Kubernetes uses the PVCs referenced as volumes in a pod and takes those into account when scheduling the pod. Kubernetes identifies PersistentVolumes that match properties associated with the claim and binds the smallest available module to the claim. The properties might include a label, or node affinity, as we saw above for local volumes.

When starting up a pod, the Kubernetes control plane makes sure the PersistentVolumes are mounted to the worker node. Then each requested storage volume is mounted into the pod at the specified mount point.
StorageClasses

The example shown above demonstrates how Kubernetes can bind PVCs to PersistentVolumes that already exist. This model in which PersistentVolumes are explicitly created in the Kubernetes cluster is known as static provisioning. The Kubernetes Persistent Volume Subsystem also supports dynamic provisioning of volumes using StorageClasses (often abbreviated SC). The StorageClass is responsible for provisioning (and deprovisioning) PersistentVolumes according to the needs of applications running in the cluster, as shown in Figure 2-11.

Figure 2-11. StorageClasses support dynamic provisioning of volumes

Depending on the Kubernetes cluster you are using, it is likely that there is already at least one StorageClass available. You can verify this using the command `kubectl get sc`. If you’re running a simple Kubernetes distribution on your local machine and don’t see any StorageClasses, you can install an open source local storage provider from Rancher with the following command:

```bash
kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/master/deploy/local-path-storage.yaml
```

This storage provider comes pre-installed in K3s, a desktop distribution also provided by Rancher. If you take a look at the YAML configuration referenced in that statement, you’ll see the following definition of a StorageClass (source code):

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-path
provisioner: rancher.io/local-path
```
As you can see from the definition, a StorageClass is defined by a few key attributes:

- The **provisioner** interfaces with an underlying storage provider such as a public cloud or storage system in order to allocate the actual storage. The provisioner can either be one of the Kubernetes built-in provisioners (referred to as “in-tree” because they are part of the Kubernetes source code), or a provisioner that conforms to the Container Storage Interface (CSI), which we’ll examine below.

- The **parameters** are specific configuration options for the storage provider that are passed to the provisioner. Common options include filesystem type, encryption settings, and throughput in terms of IOPS. Check the documentation for the storage provider for more details.

- The **reclaimPolicy** describes whether storage is reclaimed when the PersistentVolume is deleted. The default is Delete, but can be overridden to Retain, in which case the storage administrator would be responsible for managing the future state of that storage with the storage provider.

- Although it is not shown in the example above, there is also an optional **allowVolumeExpansion** flag. This indicates whether the StorageClass supports the ability for volumes to be expanded. If true, the volume can be expanded by increasing the size of the storage.request field of the PersistentVolumeClaim. This value defaults to false.

- The **volumeBindingMode** controls when the storage is provisioned and bound. If the value is **Immediate**, a PersistentVolume is immediately provisioned as soon as a PersistentVolumeClaim referencing the StorageClass as created, and the claim is bound to the PersistentVolume, regardless of whether the claim is referenced in a pod. Many storage plugins also support a second mode known as **WaitForFirstConsumer**, in which case no PersistentVolume is not provisioned until a pod is created that references the claim. This behavior is considered preferable since it gives the Kubernetes scheduler more flexibility.

### Limits on dynamic provisioning

Local cannot be dynamically provisioned by a StorageClass, so you must create them manually yourself.

Application developers can reference a specific StorageClass when creating a PVC by adding a storageClass property to the definition. For example, here is a YAML configuration for a PVC referencing the local-path StorageClass ([source code]):

```yaml
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Delete
```
apiVersion: v1
type: PersistentVolumeClaim
metadata:
  name: my-local-path-claim
spec:
  storageClassName: local-path
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

If no storageClass is specified in the claim, then the default StorageClass is used. The default StorageClass can be set by the cluster administrator. As we showed above in the Persistent Volumes section, you can opt out of using StorageClasses by using the empty string, which indicates that you are using statically provisioned storage.

StorageClasses provide a useful abstraction that cluster administrators and application developers can use as a contract: administrators define the StorageClasses, and developers reference the StorageClasses by name. The details of the underlying StorageClass implementation can differ across Kubernetes platform providers, promoting portability of applications.

This flexibility allows administrators to create StorageClasses representing a variety of different storage options, for example, to distinguish between different quality of service guarantees in terms of throughput or latency. This concept is known as “profiles” in other storage systems. See How Developers are Driving the Future of Kubernetes Storage (sidebar) for more ideas on how StorageClasses can be leveraged in innovative ways.

**Kubernetes Storage Architecture**

In the preceding sections we’ve discussed the various storage resources that Kubernetes supports via its API. In the remainder of the chapter, we’ll take a look at how these solutions are constructed, as they can give us some valuable insights on how to construct cloud-native data solutions.

**Defining Cloud-native storage**

Most of the storage technologies we discuss in this chapter are captured as part of the “cloud-native storage” solutions listed in Cloud Native Computing Foundation (CNCF) landscape. The CNCF Storage Whitepaper is a helpful resource which defines key terms and concepts for cloud native storage. Both of these resources are updated regularly.
Flexvolume

Originally, the Kubernetes codebase contained multiple “in-tree” storage plugins, that is, included in the same GitHub repo as the rest of the Kubernetes code. The advantage of this was that it helped standardize the code for connecting to different storage platforms, but there were a couple of disadvantages as well. First, many Kubernetes developers had limited expertise across the broad set of included storage providers. More significantly, the ability to upgrade storage plugins was tied to the Kubernetes release cycle, meaning that if you needed a fix or enhancement for a storage plugin, you'd have to wait until it was accepted into a Kubernetes release. This slowed the maturation of storage technology for K8s, and as a result, adoption slowed as well.

The Kubernetes community created the Flexvolume specification to allow development of plugins that could be developed independently, that is, out of the Kubernetes source code tree, without being tied to the Kubernetes release cycle. Around the same time, storage plugin standards were emerging for other container orchestration systems, and developers from these communities began to question the wisdom of developing multiple standards to solve the same basic problem.

Future Flexvolume support

While new feature development has paused on Flexvolume, many deployments still rely on these plugins, and there are no active plans to deprecate the feature as of the Kubernetes 1.21 release.

Container Storage Interface (CSI)

The Container Storage Interface (CSI) initiative was established as an industry standard for storage for containerized applications. CSI is an open standard used to define plugins that will work across container orchestration systems including Kubernetes, Mesos, and Cloud Foundry. As Saad Ali, Google engineer and chair of the Kubernetes Storage Special Interest Group (SIG), noted in The New Stack article The State of State in Kubernetes: “The Container Storage Interface allows Kubernetes to interact directly with an arbitrary storage system.”

The CSI specification is available on GitHub. Support for the CSI in Kubernetes began with the 1.x release and it went GA in the 1.13 release. Kubernetes continues to track updates to the CSI specification.

Once a CSI implementation is deployed on a Kubernetes cluster, its capabilities are accessed through the standard Kubernetes storage resources such as PVCs, PVs, and SCs. On the backend, each CSI implementation must provide two plugins: a node plugin and a controller plugin. The CSI specification defines required interfaces for these plugins using gRPC but does not specify exactly how the plugins are to be deployed.
Let’s briefly look at the role of each of these services, also depicted in Figure 2-12:

- The controller plugin supports operations on volumes such as create, delete, listing, publishing/unpublishing, tracking and expanding volume capacity. It also tracks volume status including what nodes each volume is attached to. The controller plugin is also responsible for taking and managing snapshots, and using snapshots to clone a volume. The controller plugin can run on any node - it is a standard Kubernetes controller.

- The node plugin runs on each Kubernetes worker node where provisioned volumes will be attached. The node plugin is responsible for local storage, as well as mounting and unmounting volumes onto the node. The Kubernetes control plane directs the plugin to mount a volume prior to any pods being scheduled on the node that require the volume.

![Figure 2-12. Container Storage Interface mapped to Kubernetes](image-url)
Additional CSI resources:
The CSI documentation site provides guidance for developers and storage providers who are interested in developing CSI-compliant drivers. The site also provides a very useful list of CSI-compliant drivers. This list is generally more up to date than one provided on the Kubernetes documentation site.

CSI Migration

The Kubernetes community has been very conscious of preserving forward and backward compatibility between versions, and the transition from in-tree storage plugins to the CSI is no exception. Features in Kubernetes are typically introduced as Alpha features, and progress to Beta, before being released as General Availability (GA). The introduction of a new API such as the CSI presents a more complex challenge because it involves the introduction of a new API as well as the deprecation of older APIs.

The CSI migration approach was introduced in order to promote a coherent experience for users of storage plugins. The implementation of each corresponding in-tree plugin is changed to a facade when an equivalent CSI-compliant driver becomes available. Calls on the in-tree plugin are delegated to the underlying CSI-compliant driver. The migration capability is itself a feature that can be enabled on a Kubernetes cluster.

This allows a staged adoption process that can be used as existing clusters are updated to newer Kubernetes versions. Each application can be updated independently to use CSI-compliant drivers instead of in-tree drivers. This approach to maturing and replacing APIs is a helpful pattern for promoting stability of the overall platform and providing administrators control over their migration to the new API.

Container Attached Storage

While the CSI is an important step forward in standardizing storage management across container orchestrators, it does not provide implementation guidance on how or where the storage software runs. Some CSI implementations are basically thin wrappers around legacy storage management software running outside of the Kubernetes cluster. While there are certainly benefits to this reuse of existing storage assets, many developers have expressed a desire for storage management solutions that run entirely in Kubernetes alongside their applications.

Container Attached Storage is a design pattern which provides a more cloud-native approach to managing storage. The business logic to manage storage operations such as attaching volumes to applications is itself composed of microservices running in containers. This allows the storage layer to have the same properties as other applications deployed on Kubernetes and reduces the number of different management
interfaces administrators have to keep track of. The storage layer becomes just another Kubernetes application.

As Evan Powell noted in his article on the CNCF Blog, Container Attached Storage: A primer, “Container Attached Storage reflects a broader trend of solutions that reinvent particular categories or create new ones – by being built on Kubernetes and microservices and that deliver capabilities to Kubernetes-based microservice environments. For example, new projects for security, DNS, networking, network policy management, messaging, tracing, logging and more have emerged in the cloud-native ecosystem.”

There are several examples of projects and products that embody the CAS approach to storage. Let’s examine a few of the open-source options.

**OpenEBS**

OpenEBS is a project created by MayaData and donated to the CNCF, where it became a sandbox project in 2019. The name is a play on Amazon’s Elastic Block Store, and OpenEBS is an attempt to provide an open source equivalent to this popular managed service. OpenEBS provides storage engines for managing both local and NVMe PersistentVolumes.

OpenEBS provides a great example of a CSI-compliant implementation deployed onto Kubernetes, as shown in Figure 2-13. The control plane includes the OpenEBS provisioner, which implements the CSI controller interface, and the OpenEBS API server, which provides a configuration interface for clients and interacts with the rest of the Kubernetes control plane.

The Open EBS data plane consists of the Node Disk Manager (NDM) as well as dedicated pods for each PersistentVolume. The NDM runs on each Kubernetes worker where storage will be accessed. It implements the CSI node interface and provides the helpful functionality of automatically detecting block storage devices attached to a worker node.
OpenEBS creates multiple pods for each volume. A controller pod is created as the primary replica, and additional replica pods are created on other Kubernetes worker nodes for high availability. Each pod includes sidecars that expose interfaces for metrics collection and management, which allows the control plane to monitor and manage the data plane.

Longhorn

Longhorn is an open-source, distributed block storage system for Kubernetes. It was originally developed by Rancher, and became a CNCF sandbox project in 2019. Longhorn focuses on providing an alternative to cloud-vendor storage and expensive external storage arrays. Longhorn supports providing incremental backups to NFS or AWS S3 compatible storage, and live replication to a separate Kubernetes cluster for disaster recovery.

Longhorn uses a similar architecture to that shown for OpenEBS; according to the documentation, “Longhorn creates a dedicated storage controller for each block device volume and synchronously replicates the volume across multiple replicas stored on multiple nodes. The storage controller and replicas are themselves orchestrated using Kubernetes.” Longhorn also provides an integrated user interface to simplify operations.

Rook and Ceph

According to its website, “Rook is an open source cloud-native storage orchestrator, providing the platform, framework, and support for a diverse set of storage
solutions to natively integrate with cloud-native environments.” Rook was originally created as a containerized version of Ceph that could be deployed in Kubernetes. Ceph is an open-source distributed storage framework that provides block, file, and object storage. Rook was the first storage project accepted by the CNCF and is now considered a CNCF Graduated project.

Rook is a truly Kubernetes-native implementation in the sense that it makes use of Kubernetes custom resources (CRDs) and custom controllers called operators. Rook provides operators for Ceph, Apache Cassandra, and Network File System (NFS). We’ll learn more about custom resources and operators in Chapter 4.

There are also commercial solutions for Kubernetes that embody the CAS pattern. These include MayaData (creators of OpenEBS), Portworx by PureStorage, Robin.io, and StorageOS. These companies provide both raw storage in block and file formats, as well as integrations for simplified deployments of additional data infrastructure such as databases and streaming solutions.

Container Object Storage Interface (COSI)

The CSI provides support for file and block storage, but object storage APIs require different semantics and don’t quite fit the CSI paradigm of mounting volumes. In Fall 2020, a group of companies led by MinIO began work on a new API for object storage in container orchestration platforms: the Container Object Storage Interface (COSI). COSI provides a Kubernetes API more suited to provisioning and accessing object storage, defining a bucket custom resource and including operations to create buckets and manage access to buckets. The design of the COSI control plane and data plane is modeled after the CSI. COSI is an emerging standard with a great start and potential for wide adoption in the Kubernetes community and potentially beyond.

How Developers are Driving the Future of Kubernetes Storage

With Kiran Mova, co-founder and CTO of MayaData, member of Kubernetes Storage Special Interest Group (SIG)

Many organizations are just starting their containerization journey. Kubernetes is the shiny object, and everybody wants to run everything in Kubernetes. But not all teams are ready for Kubernetes, much less managing stateful workloads on Kubernetes.

Application developers are the ones driving the push for stateful workloads on Kubernetes. These developers get started with cloud resources that are available to them, even a single node Kubernetes cluster, and assume they’re ready to run that in production. Developers are “Kuberneticizing” their in-house applications, and the demands on storage are quite different from what the platform teams that support them are used to.
Microservices and Kubernetes have changed the way storage volumes are provisioned. Platform teams are used to thinking about data in terms of provisioning volumes with the required throughput or capacity. In the old way, the platform team would meet with the application team, estimate the size of the data, do a month of planning, provision a 2-3 TB volume, and mount it into the VMs or bare metal servers and that would provide enough storage capacity for the next year.

With Kubernetes, provisioning has become much easier and ad-hoc. You can run things in a very cost effective and agile way by adopting Kubernetes. But many platform teams are still working to catch up. Some teams are simply focused on provisioning storage correctly, while others are beginning to focus on “day 2” operations, such as automated provisioning, expanding volumes, or disconnecting and destroying volumes.

Platform teams don’t yet have a foolproof way to run stateful workloads in Kubernetes, so they often offload persistence to public cloud providers. The public clouds make a strong case for their managed services, claiming they have everything that you’ll need to run a storage system, but once you start using managed services for state, you can become dependent on those cloud providers and get stuck.

Meanwhile, there are innovations in storage technology happening in parallel:

The landscape is shifting back and forth between hyperconverged and disaggregated. This re-architecture is happening at all the layers of the stack, and it’s not just the software, it includes processes and the people who consume the data.

Hardware trends are driving toward low-latency solutions including NVMe and DPDK/SPDK, and changes to the Linux kernel like io_uring to take advantage of faster hardware.

Container attached storage will help us manage storage more effectively. For example, being able to reclaim storage space when workloads shrink. This can be a difficult problem with data distributed across multiple nodes. We’ll need better logic for relocating data onto existing nodes.

Technologies that bring more automation for compliance and operations are coming into the picture as well.

With all these innovations, it can be a bit overwhelming to understand the big picture and determine how to leverage this technology for maximum benefit. Platform SREs need to learn about Kubernetes, declarative deployments, GitOps principles, new volume types, and even database concepts like eventual consistency.

We envision a future in which application developers will specify their Kubernetes storage needs in terms of the required quality of service, such as I/O operations per second (IOPS) and throughput. Developers should be able to specify different storage needs for their workloads in more human-relatable terms. For example, platform teams could define StorageClasses for “fast storage” vs “slow storage”, or perhaps “metadata storage” vs “data storage”. These StorageClasses will make different cost/
performance tradeoffs and provide specific service level agreements (SLAs). We may even see some standard definitions start to emerge for these new StorageClasses.

Ideally, application teams should not be picking into what storage solutions are chosen. The only thing an application developer should be concerned with is specifying PersistentVolumeClaims for their application, with the StorageClasses they need. The other details of managing storage should be hidden, although of course the storage subsystem will report errors including status and logs via the standard Kubernetes mechanisms. This capability will make things a lot simpler for application developers, whether they’re deploying a database, or some other stateful workload.

These innovations will guide us to a more optimal place with storage on Kubernetes. Today we’re in a place where deploying infrastructure is easy. Let’s work together to get to a place where deploying the right infrastructure is easy.

As you can see, storage on Kubernetes is an area in which there is a lot of innovation, including multiple open source projects and commercial vendors competing to provide the most usable, cost effective, and performant solutions. The Cloud-Native Storage section of the CNCF Landscape provides a helpful listing of storage providers and related tools, including the technologies referenced in this chapter and many more.

**Summary**

In this chapter, we’ve explored how persistence is managed in container systems like Docker, and container orchestration systems like Kubernetes. You’ve learned about the various Kubernetes resources that can be used to manage stateful workloads, including Volumes, PersistentVolumes, PersistentVolumeClaims, StorageClasses. We’ve seen how the Container Storage Interface and Container Attached Storage pattern point the way toward more cloud-native approaches to managing storage. Now you’re ready to learn how to use these building blocks and design principles to manage stateful workloads including databases, streaming data, and more.
As we discussed in Chapter 1, Kubernetes was designed for stateless workloads. A corollary to this is that stateless workloads are what Kubernetes does best. Because of this, some have argued that you shouldn’t try to run stateful workloads on Kubernetes, and you may hear various recommendations about what you should do instead: “use a managed service”, or “leave data in legacy databases in your on-premises data center”, or perhaps even “run your databases in the cloud, but in traditional VMs instead of containers.”

While these recommendations are still viable options, one of our main goals in this book is to demonstrate that running data infrastructure in Kubernetes has become not only a viable option, but a preferred option. In his article, A Case for Databases on Kubernetes from a Former Skeptic, Chris Bradford describes his journey from being skeptical of running any stateful workload in Kubernetes, to grudging acceptance of running data infrastructure on Kubernetes for development and test workloads, to enthusiastic evangelism around deploying databases on K8s in production.
This journey is typical of many in the Data on Kubernetes community. By the middle of 2020, Boris Kurktchiev was able to cite a growing consensus that managing stateful workloads on Kubernetes had reached a point of viability, and even maturity, in his article *3 Reasons to Bring Stateful Applications to Kubernetes.*

How did this change come about? Over the past several years, the Kubernetes community has shifted focus toward adding features that support the ability to manage state in a cloud-native way on Kubernetes. The storage elements represent a big part of this shift we introduced in the previous chapter, including the Kubernetes PersistentVolume subsystem and the adoption of the Container Storage Interface. In this chapter, we'll complete this part of the story by looking at Kubernetes resources for building stateful applications on top of this storage foundation. We'll focus in particular on a specific type of stateful application: data infrastructure.

**The Hard Way**

The phrase “doing it the hard way” has come to be associated with avoiding the easy option in favor of putting in the detailed work required to accomplish a result that will have lasting significance. Throughout history, pioneers of all persuasions are well known for taking pride in having made the sacrifice of blood, sweat, and tears that make life just that little bit more bearable for the generations that follow. These elders are often heard to lament when their proteges fail to comprehend the depth of what they had to go through.

In the tech world it’s no different. While new innovations such as APIs and “no code” environments have massive potential to grow a new crop of developers worldwide, it is still the case that a deeper understanding of the underlying technology is required in order to manage highly available and secure systems at worldwide scale. It’s when things go wrong that this detailed knowledge proves its worth. This is why many of us who are software developers and never touch a physical server in our day jobs gain so much from building our own PC by wiring chips and boards by hand. It’s also one of the hidden benefits of serving as informal IT consultants for our friends and family.

For the Kubernetes community, of course, “the hard way” has an even more specific connotation. Google engineer Kelsey Hightower's *Kubernetes the Hard Way* has become a sort of rite of passage for those who want a deeper understanding of the elements that make up a Kubernetes cluster. This popular tutorial walks you through downloading, installing, and configuring each of the components that make up the Kubernetes control plane. The result is a working Kubernetes cluster, which, although not suitable for deploying a production workload, is certainly functional enough for development and learning. The appeal of the approach is that all of the instructions are typed by hand instead of downloading a bunch of scripts that do everything for you, so that you understand what is happening at each step.
In this chapter, we'll emulate this approach and walk you through deploying some example data infrastructure the hard way ourselves. Along the way, we'll get more hands-on experience with the storage resources you learned about in Chapter 2, and we'll introduce additional Kubernetes resource types for managing compute and network to complete the “Compute, Network, Storage” triad we introduced in Chapter 1. Are you ready to get your hands dirty? Let's go!

**Examples are Not Production-Grade**

The examples we present in this chapter are primarily for introducing new elements of the Kubernetes API and are not intended to represent deployments we'd recommend running in production. We'll make sure to highlight where there are gaps so that we can demonstrate how to fill them in upcoming chapters.

**Prerequisites for running data infrastructure on Kubernetes**

To follow along with the examples in this chapter, you'll want to have a Kubernetes cluster to work on. If you've never tried it before, perhaps you'll want to build a cluster using the Kubernetes the Hard Way instructions, and then use that same cluster to add data infrastructure the hard way as well. You could also use a simple desktop K8s as well, since we won't be using a large amount of resources. If you're using a shared cluster, you might want to install these examples in their own namespace to isolate them from the work of others.

```
kubectl config set-context --current --namespace=<insert-namespace-name-here>
```

You'll also need to make sure you have a StorageClass in your cluster. If you're starting from a cluster built the hard way, you won't have one. You may want to follow the instructions in the section StorageClasses for installing a simple StorageClass and provisioner that expose local storage (source code).

You'll want to use a StorageClass that supports a `volumeBindingMode` of `WaitForFirstConsumer`. This gives Kubernetes the flexibility to defer provisioning storage until we need it. This behavior is generally preferred for production deployments, so you might as well start getting in the habit.

**Running MySQL on Kubernetes**

First, let's start with a super simple example. MySQL is one of the most widely used relational databases due to its reliability and usability. For this example we'll build on the MySQL tutorial in the official Kubernetes documentation, with a couple of twists. You can find the source code used in this section at Deploying MySQL Example -
Data on Kubernetes the Hard Way. The tutorial includes two Kubernetes deployments: one to run MySQL pod, and another to run a sample client, in this case Wordpress. This configuration is shown in Figure 3-1.

![Figure 3-1. Sample Kubernetes Deployment of MySQL](image)

In this example, we see that there is a PersistentVolumeClaim for each pod. For the purposes of this example, we’ll assume these claims are satisfied by a single volume provided by the default StorageClass. You’ll also notice that each pod is shown as part of a ReplicaSet and that there is a service exposed for the MySQL database. Let’s take a pause and introduce these concepts.

**ReplicaSets**

Production application deployments on Kubernetes do not typically deploy individual pods, because an individual pod could easily be lost when the node disappears. Instead, pods are typically deployed in the context of a Kubernetes resource that manages their lifecycle. ReplicaSet is one of these resources, and the other is StatefulSet, which we’ll look at later in the chapter.

The purpose of a ReplicaSet (RS) is to ensure that a specified number of replicas of a given pod are kept running at any given time. As pods are destroyed, others are created to replace them in order to satisfy the desired number of replicas. A ReplicaSet is defined by a pod template, a number of replicas, and a selector. The pod template defines a specification for pods that will be managed by the ReplicaSet, similar to
what we saw for individual pods created in the examples in Chapter 2. The number of replicas can be 0 or more. The selector identifies pods that are part of the ReplicaSet. 

Let's look at a portion of an example definition of a ReplicaSet for the Wordpress application shown in Figure 3-1:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: wordpress-mysql
labels:
  app: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
tier: mysql
template:
  metadata:
    labels:
      app: wordpress
tier: mysql
spec:
  containers:
  - image: mysql:5.6
    name: mysql
...
```

A ReplicaSet is responsible for creating or deleting pods in order to meet the specified number of replicas. You can scale the size of a RS up or down by changing this value. The pod template is used when creating new pods. Pods that are managed by a ReplicaSet contain a reference to the RS in their `metadata.ownerReferences` field. A ReplicaSet can actually take responsibility for managing a pod that it did not create if the selector matches and the pod does not reference another owner. This behavior of a ReplicaSet is known as *acquiring* a pod.

You might be wondering why we didn't provide a full definition of a ReplicaSet above. As it turns out, most application developers do not end up using ReplicaSets directly, because Kubernetes provides another resource type that manages ReplicaSets declaratively: Deployments.
Define ReplicaSet selectors carefully

If you do create ReplicaSets directly, make sure that the selector you use is unique and does not match any bare pods that you do not intend to be acquired. It is possible that pods that do not match the pod template can be acquired if the selectors match.

For more information about managing the lifecycle of ReplicaSets and the pods they manage, see the Kubernetes documentation.

Deployments

A Kubernetes Deployment is a resource which builds on top of ReplicaSets with additional features for lifecycle management, including the ability to rollout new versions and rollback to previous versions. As shown in Figure 3-2, creating a Deployment results in the creation of a ReplicaSet as well.

![Figure 3-2. Deployments and ReplicaSets](image)

This figure highlights that ReplicaSets (and therefore the Deployments that manage them) operate on cloned replicas of pods, meaning that the definitions of the pods are the same, even down to the level of PersistentVolumeClaims. The definition of a ReplicaSet references a single PVC that is provided to it, and there is no mechanism provided to clone the PVC definition for additional pods. For this reason, Deployments and ReplicaSets are not a good choice if your intent is that each pod have access to its own dedicated storage.

Deployments are a good choice if your application pods do not need access to storage, or if your intent is that they access the same piece of storage. However, the cases
where this would be desirable are pretty rare, since you likely don’t want a situation in which you could have multiple simultaneous writers to the same storage.

Let’s create an example Deployment. First, create a secret that will represent the database password (substitute in whatever string you want for the password):

```
kubectl create secret generic mysql-root-password --from-literal=password=<your password>
```

Next, create a PVC that represents the storage that the database can use (source code). A single PVC is sufficient in this case since you are creating a single node. This should work as long as you have an appropriate storage class as referenced earlier.

```
apiVersion: v1
type: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Next, create a Deployment with a pod spec that runs MySQL (source code). Note that it includes a reference to the PVC you just created as well as the Secret containing the root password for the database.

```
apiVersion: apps/v1
type: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
        - image: mysql:5.7
          name: mysql
```

```
env:
  - name: MYSQL_ROOT_PASSWORD
    valueFrom:
      secretKeyRef:
        name: mysql-root-password
        key: password
ports:
  - containerPort: 3306
    name: mysql
volumeMounts:
  - name: mysql-persistent-storage
    mountPath: /var/lib/mysql
volumes:
  - name: mysql-persistent-storage
    persistentVolumeClaim:
      claimName: mysql-pv-claim

There are a couple of interesting things to note about this Deployment's specification.

- First, note that the Deployment has a Recreate strategy. This refers to how the
  Deployment handles the replacement of pods when the pod template is updated,
  and we'll discuss this shortly.

- Next, note under the pod template that the password is passed to the pod as an
  environment variable extracted from via the secret you created above. Overriding
  the default password is an important aspect of securing any database deployment.

- Note also that a single port is exposed on the MySQL image for database access,
  since this is a relatively simple example. In other samples in this book we'll see
  cases of pods that expose additional ports for administrative operations, metrics
  collection, and more. The fact that access is disabled by default is a great feature
  of Kubernetes.

- The MySQL image mounts a volume for its persistent storage using the PVC
  defined above.

- Finally, note that the number of replicas was not provided in the specification.
  This means that the default value of 1 will be used.

After applying the configuration above, try using a command like kubectl get deploy
ments,rs,pods to check and see the items that Kubernetes created for you. You'll
notice a single ReplicaSet named after the deployment that includes a random string,
for example: wordpress-mysql-655c8d9c54. The pod's name references the name of
the ReplicaSet, adding some additional random characters, for example: wordpress-
mysql-655c8d9c54-tgswd. These names provide a quick way to identify the relation‐
ships between these resources.
Here are a few of the actions that a Deployment takes to manage the lifecycle of ReplicaSets. In keeping with Kubernetes’ emphasis on declarative operations, most of these are triggered by updating the specification of the Deployment:

**Initial rollout**
When you create a Deployment, Kubernetes uses the specification you provide to create a ReplicaSet. The process of creating this ReplicaSet and its pods is known as a *rollout*. A rollout is also performed as part of a rolling update, as described below.

**Scaling up or down**
When you update a Deployment to change the number of replicas, the underlying ReplicaSet is scaled up or down accordingly.

**Rolling update**
When you update the Deployment’s pod template, for example by specifying a different container image for the pod, Kubernetes creates a new ReplicaSet based on the new pod template. The way that Kubernetes manages the transition between the old and new ReplicaSets is described by the Deployment’s `spec.strategy` property, which defaults to a value called `RollingUpdate`. In a rolling update, the new ReplicaSet is slowly scaled up by creating pods conforming to the new template, as the number of pods in the existing ReplicaSet is scaled down. During this transition, the Deployment enforces a maximum and minimum number of pods, expressed as percentages, as set by the `spec.strategy.rollingupdate.maxSurge` and `maxUnavailable` properties. Each of these values default to 25%.

**Recreate update**
The other strategy option for use when you update the pod template is Recreate. This is the option that was set in the Deployment above. With this option, the existing ReplicaSet is terminated immediately before the new ReplicaSet is created. This strategy is useful for development environments since it completes the update more quickly, whereas `RollingUpdate` is more suitable for production environments since it emphasises high availability.

**Rollback update**
It is possible that in creating or updating a Deployment you could introduce an error, for example by updating a container image in a pod with a version that contains a bug. In this case the pods managed by the Deployment might not even initialize fully. You can detect these types of errors using commands such as `kubectl rollout status`. Kubernetes provides a series of operations for managing the history of rollouts of a Deployment. You can access these via `kubectl` commands such as `kubectl rollout history`, which provides a numbered history of rollouts for a deployment, and `kubectl rollout undo`, which reverts a
Deployment to the previous rollout. You can also undo to a specific rollout version with the `--to-version` option. Because kubectl supports rollouts for other resource types we'll cover below (StatefulSets and DaemonSets), you'll need to include the resource type and name when using these commands, for example:

```bash
cubectl rollout history deployment/wordpress-mysql
```

Which produces output such as:

```
deployment.apps/wordpress-mysql
REVISION    CHANGE-CAUSE
 1           <none>
```

As you can see, Kubernetes Deployments provide some sophisticated behaviors for managing the lifecycle of a set of cloned pods. You can test out these lifecycle operations (other than rollback) by changing the Deployment's YAML specification and re-applying it. Try scaling the number of replicas to 2 and back again, or using a different MySQL image. After updating the Deployment, you can use a command like `kubectl describe deployment wordpress-mysql` to observe the events that Kubernetes initiates to bring your Deployment to your desired state.

There are other options available for Deployments which we don't have space to go into here, for example, how to specify what Kubernetes does if you attempt an update that fails. For a more in-depth explanation of the behavior of Deployments, see the Kubernetes documentation.

## Services

In the steps above, you've created a PVC to specify the storage needs of the database, a Secret to provide administrator credentials, and a Deployment to manage the lifecycle of a single MySQL pod. Now that you have a running database, you'll want to make it accessible to applications. In our scheme of compute, network, and storage that we introduced in Chapter 1, this is the networking part.

Kubernetes Services are the primitive that we need to use to expose access to our database as a network service. A Service provides an abstraction for a group of pods running behind it. In the case of a single MySQL node as in this example, you might wonder why we'd bother creating this abstraction. One key feature that a Service supports is to provide a consistently named endpoint that doesn't change. You don't want to be in a situation of having to update your clients whenever the database pod is restarted and gets a new IP address. You can create a Service for accessing MySQL using a YAML configuration like this (source code):

```yaml
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
```
Here are a couple of things to note about this configuration:

- First, this configuration specifies a port that is exposed on the Service: 3306. In defining a service there are actually two ports involved: the port exposed to clients of the Service, and the targetPort exposed by the underlying pods that the Service is fronting. Since you haven’t specified a targetPort, it defaults to the port value.

- Second, the selector defines what pods the Service will direct traffic to. In this configuration, there will only be a single MySQL pod managed by the Deployment, and that’s just fine.

- Finally, if you have worked with Kubernetes Services before, you may note that there is no serviceType defined for this service, which means that it is of the default type, known as ClusterIP. Furthermore, since the clusterIP property is set to None, this is what is known as a headless service, that is, a service where the service’s DNS name is mapped directly to the IP addresses of the selected pods.

Kubernetes supports several types of services to address different use cases, which are shown in Figure 3-3. We’ll introduce them briefly here in order to highlight their applicability to data infrastructure:

**ClusterIP Service**

This type of Service is exposed on an IP address that is only accessible from within the Kubernetes cluster. This is the type of service that you’ll see used most often for data infrastructure such as databases in Kubernetes, especially headless services, since this infrastructure is typically deployed in Kubernetes alongside the application which uses it.

**NodePort Service**

A NodePort Service is exposed externally to the cluster on the IP address of each worker node. A ClusterIP service is also created internally, to which the NodePort routes traffic. You can allow Kubernetes to select what external port is used, or specify the one you desire using the NodePort property. NodePort services are most suitable for development environments, when you need to debug what is happening on a specific instance of a data infrastructure application.
**LoadBalancer**

LoadBalancer services represent a request from the Kubernetes runtime to set up a load balancer provided by the underlying cloud provider. For example, on Amazon's Elastic Kubernetes Service (EKS), requesting a LoadBalancer service causes an instance of an Elastic Load Balancer (ELB) to be created. Usage of LoadBalancers in front of multi-node data infrastructure deployments is typically not required, as these data technologies often have their own approaches for distributing load. For example, Apache Cassandra drivers are aware of the topology of a Cassandra cluster and provide load balancing features to client applications, eliminating the need for a load balancer.

**ExternalName Service**

An ExternalName Service is typically used to represent access to a service that is outside your cluster, for example a database that is running externally to Kubernetes. An ExternalName service does not have a selector as it is not mapping to any pods. Instead, it maps the Service name to a CNAME record. For example, if you create a `my-external-database` service with an `externalName` of `database.mydomain.com`, references in your application pods to `my-external-database` will be mapped to `database.mydomain.com`.

![Figure 3-3. Kubernetes Service Types](image)

Note also the inclusion of Ingress in the figure. While Kubernetes Ingress is not a type of Service, it is related. An Ingress is used to provide access to Kubernetes services
from outside the cluster, typically via HTTP. Multiple Ingress implementations are available, including Nginx, Traefik, Ambassador (based on Envoy) and others. Ingress implementations typically provide features including SSL termination and load balancing, even across multiple different Kubernetes Services. As with LoadBalancer Services, Ingresses are more typically used at the application tier.

**Accessing MySQL**

Now that you have deployed the database, you’re ready to deploy an application that uses it - the Wordpress server.

First, the server will need its own PVC. This helps illustrate that there are cases of applications which leverage storage directly, perhaps for storing files, and applications that use data infrastructure, and applications that do both. You can make a small request since this is just for demonstration purposes (source code):

```yaml
apiVersion: v1
class: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Next, create a Deployment for a single Wordpress node (source code):

```yaml
apiVersion: apps/v1
class: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
```
Notice that the database host and password for accessing MySQL are passed to WordPress as environment variables. The value of the host is the name of the service you created for MySQL above. This is all that is needed for the database connection to be routed to your MySQL instance. The value for the password is extracted from the secret, similar to the configuration of the MySQL deployment above.

You’ll also notice that Wordpress exposes an HTTP interface at port 80, so let’s create a service to expose the Wordpress server (source code):

```yaml
apiVersion: v1
type: LoadBalancer
metadata:
  app: wordpress
spec:
  ports:
  - port: 80
  selector:
    app: wordpress
tier: frontend
type: LoadBalancer

Note that the service is of type LoadBalancer, which should make it fairly simple to access from your local machine. Execute the command `kubectl get services` to get the load balancer’s IP address, then you can open the Wordpress instance in your browser with the URL `http://<ip>`. Try logging in and creating some pages.
Accessing Services from Kubernetes distributions

The exact details of accessing services will depending on the Kubernetes distribution you are using and whether you're deploying apps in production, or just testing something quickly like we're doing here. If you're using a desktop Kubernetes distributions, you may wish to use a NodePort service instead of LoadBalancer for simplicity. You can also consult the documentation for specific instructions on accessing services, such as those provided for Minikube or K3d.

When you're done experimenting with your Wordpress instance, you can clean up the resources specified in the configuration files you've used in the local directory using the command, including the data stored in your PersistentVolumeClaim:

```bash
kubectl delete -k ./
```

At this point, you might be feeling like this was relatively easy, despite our claims of doing things “the hard way”. And in a sense, you'd be right. So far, we've deployed a single node of a simple database with sane defaults that we didn't have to spend much time configuring. Creating a single node is of course fine if your application is only going to store a small amount of data. Is that all there is to deploying databases on Kubernetes? Of course not! Now that we've introduced a few of the basic Kubernetes resources via this simple database deployment, it's time to step up the complexity a bit. Let's get down to business!

Running Apache Cassandra on Kubernetes

In this section we'll look at running a multi-node database on Kubernetes using Apache Cassandra. Cassandra is a NoSQL database first developed at Facebook that became a top-level project of the Apache Software Foundation in 2010. Cassandra is an operational database that provides a tabular data model, and its Cassandra Query Language (CQL) is similar to SQL.

Cassandra is a database designed for the cloud, as it scales horizontally by adding nodes, where each node is a peer. This decentralized design has been proven to have near-linear scalability. Cassandra supports high availability by storing multiple copies of data or replicas, including logic to distribute those replicas across multiple data-centers and cloud regions. Cassandra is built on similar principles to Kubernetes in that it is designed to detect failures and continue operating while the system can recover to its intended state in the background. All of these features make Cassandra an excellent fit for deploying on Kubernetes.

In order to discuss how this deployment works, it's helpful to understand Cassandra's approach to distributing data from two different perspectives: physical and logical.
Borrowing some of the visuals from *Cassandra: The Definitive Guide*, you can see these perspectives in Figure 3-4. From a physical perspective, Cassandra nodes (not to be confused with Kubernetes worker nodes) are organized using concepts called racks and datacenters. While the terms betray Cassandra’s origins when on-premise data centers were the dominant way software was deployed in the mid 2000s, they can be flexibly applied. In cloud deployments, racks often represent an availability zone, while datacenters represent a cloud region. However these are represented, the important part is that they represent physically separate failure domains. Cassandra uses awareness of this topology to make sure that it stores replicas in multiple physical locations to maximize availability of your data in the event of failures, whether those failures are a single machine, a rack of servers, an availability zone, or an entire region.

![Physical and Logical Views of Cassandra's Distributed Architecture](image)

**Figure 3-4. Physical and Logical Views of Cassandra's Distributed Architecture**

The logical view helps us understand how Cassandra determines what data will be placed on each node. Each row of data in Cassandra is identified by a primary key, which consists of one or more partition key columns which are used to allocate data across nodes, as well as optional clustering columns, which can be used to organize multiple rows of data within a partition for efficient access. Each write in Cassandra (and most reads) reference a specific partition by providing the partition key values, which Cassandra hashes together to produce a value known as a *token*, which is a value between $-2^{63}$ and $2^{63} - 1$. Cassandra assigns each of its nodes responsibility for
one or more token ranges (shown as a single range per node in Figure 3-4 for simplicity). The physical topology is taken into account in the assignment of token ranges in order to ensure copies of your data are distributed across racks and datacenters.

Now we're ready to consider how Cassandra maps onto Kubernetes. It's important to consider two implications of Cassandra's architecture:

**Statefulness**

Each Cassandra node has state that it is responsible for maintaining. Cassandra has mechanisms for replacing a node by streaming data from other replicas to a new node, which means that a configuration in which nodes use local ephemeral storage is possible, at the cost of longer startup time. However, it's more common to configure each Cassandra node to use persistent storage. In either case, each Cassandra node needs to have its own unique PersistentVolumeClaim.

**Identity**

Although each Cassandra node is the same in terms of its code, configuration, and functionality in a fully peer-to-peer architecture, the nodes are different in terms of their actual role. Each node has an identity in terms of where it fits in the topology of datacenters and racks, and its assigned token ranges.

These requirements for identity and an association with a specific PersistentVolumeClaim present some challenges for Deployments and ReplicaSets that they weren't designed to handle. Starting early in Kubernetes' existence, there was an awareness that another mechanism was needed to manage stateful workloads like Cassandra.

**StatefulSets**

Kubernetes began providing a resource to manage stateful workloads with the alpha release of PetSets in the 1.3 release. This capability has matured over time and is now known as StatefulSets (see: Sidebar: Are Your Stateful Workloads Pets or Cattle? below). A StatefulSet has some similarities to a ReplicaSet in that it is responsible for managing the lifecycle of a set of pods, but the way in which it goes about this management has some significant differences. In order to address the needs of stateful applications, like those of Cassandra like those listed above, StatefulSets demonstrate the following key properties:

**Stable identity for pods**

First, StatefulSets provide a stable name and network identity for pods. Each pod is assigned a name based on the name of the StatefulSet, plus an ordinal number. For example, a StatefulSet called cassandra would have pods named cassandra-1, cassandra-2, cassandra-3, and so on, as shown in Figure 3-5. These are stable names, so if a pod is lost for some reason and needs replacing, the replacement will have the same name, even if it is started on a different
worker node. Each pod’s name is set as it’s hostname, so if you create a headless service, you can actually address individual pods as needed, for example: cassandra-1.cqlservice.default.svc.cluster.local. We’ll discuss more about running Kubernetes Services for Cassandra later in this chapter in Accessing Cassandra.

![Diagram of Cassandra deployment on Kubernetes with StatefulSets](image)

**Figure 3-5. Sample Deployment of Cassandra on Kubernetes with StatefulSets**

**Ordered lifecycle management**

StatefulSets provide predictable behaviors for managing the lifecycle of pods. When scaling up the number of pods in a StatefulSet, new pods are added according to the next available number. For example, expanding the StatefulSet in Figure 3-5 would cause the creation of pods such as cassandra-4 and cassandra-5. Scaling down has the reverse behavior, as the pods with the highest ordinal numbers are deleted first. This predictability simplifies management, for example by making it obvious which nodes should be backed up before reducing cluster size.

**Persistent disks**

Unlike ReplicaSets, which create a single PersistentVolumeClaim shared across all of their pods, StatefulSets create a PVC associated with each pod. If a pod in a StatefulSet is replaced, the replacement pod is bound to the PVC which has the state it is replacing. Replacement could occur because of a pod failing or the scheduler choosing to run a pod on another node in order to balance the load. For a database like Cassandra, this enables quick recovery when a Cassandra node is lost, as the replacement node can recover its state immediately from the
associated PersistentVolume rather than needing to have data streamed from other replicas.

Managing data replication
When planning your application deployment, make sure you consider whether data is being replicated at the data tier or the storage tier. A distributed database like Cassandra manages replication itself, storing copies of your data on multiple nodes according to the replication factor you request, typically 3 per Cassandra data-center. The storage provider you select may also offer replication. If the Kubernetes volume for each Cassandra pod has 3 replicas, you could end up storing 9 copies of your data. While this certainly promotes high data survivability, this might cost more than you intend.

Are Your Stateful Workloads Pets or Cattle?
PetSet might seem like an odd name for a Kubernetes resource, and has since been replaced, but it provides some interesting insights into the thought process of the Kubernetes community in supporting stateful workloads. The name PetSets is a reference to a discussion that has been active in the DevOps world since at least 2012. The original concept has been attributed to Bill Baker, formerly of Microsoft.

The basic idea is that there are two ways of handling servers: to treat them as pets that require care, feeding, and nurture, or to treat them as cattle, to which you don’t develop an attachment or provide a lot of individual attention. If you’re logging into a server regularly to perform maintenance activities, you’re treating it as a pet.

The implication is that the life of the operations engineer can be greatly improved by being able to treat more and more elements as cattle than as pets. With the move to modern cloud-native architectures, this concept has extended from servers, to virtual machines and containers, and even to individual microservices. It’s also helped promote the use of architectural approaches for high availability and surviving the loss of individual components that have made technologies like Kubernetes and Cassandra successful.

As you can see, the naming of a Kubernetes resource “PetSets” carried a lot of freight and perhaps even a bit of skepticism to running stateful workloads on Kubernetes at all. In the end, however, PetSets helped take the care and feeding out of managing state on Kubernetes, and the name change to StatefulSets was very appropriate. Taken together, capabilities like StatefulSets, the PersistentVolume subsystem introduced in Chapter 2, and operators (coming in Chapter 4) are bringing a level of automation that promises a day in the near future when we will manage data on Kubernetes like cattle.
Defining StatefulSets

Now that you’ve learned a bit about StatefulSets, let’s examine how they can be used to run Cassandra. You’ll configure a simple 3-node cluster the “hard way” using a Kubernetes StatefulSet to represent a single Cassandra datacenter containing a single rack. While this example was inspired by the Cassandra tutorial in the Kubernetes documentation, it does differ from the tutorial in a few respects. The source code used in this section is located at Deploying Cassandra Example - Data on Kubernetes the Hard Way. This approximates the configuration shown in Figure 3-5.

To set up a Cassandra cluster in Kubernetes, you’ll first need a headless service. This service represents the “CQL Service” shown in Figure 3-5, providing an endpoint that clients can use to obtain addresses of all the Cassandra nodes in the StatefulSet (source code):

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

You’ll reference this service in the definition of a StatefulSet which will manage your Cassandra nodes (source code). Rather than applying this configuration immediately, you may want to wait until after we do some quick explanations below. The configuration looks like this:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  podManagementPolicy: OrderedReady
  updateStrategy: RollingUpdate
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
```

78  |  Chapter 3: Databases on Kubernetes the Hard Way
app: cassandra
spec:
  containers:
    - name: cassandra
      image: cassandra
      ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
      lifecycle:
        preStop:
          exec:
            command:
            - /bin/sh
            - -c
            - nodetool drain
      env:
        - name: CASSANDRA_CLUSTER_NAME
          value: "cluster1"
        - name: CASSANDRA_DC
          value: "dc1"
        - name: CASSANDRA_RACK
          value: "rack1"
        - name: CASSANDRA_SEEDS
          value: "cassandra-0.cassandra.default.svc.cluster.local"
      volumeMounts:
        - name: cassandra-data
          mountPath: /var/lib/cassandra
  volumeClaimTemplates:
    - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: standard-rwo
      resources:
        requests:
          storage: 1Gi

This is the most complex configuration we’ve looked at together so far, so let’s simplify it by looking at one portion at a time.

**StatefulSet metadata**

We’ve named and labeled this StatefulSet cassandra, and that same string will be used as the selector for pods belonging to the StatefulSet.
Exposing `StatefulSet` pods via a Service

The spec of the `StatefulSet` starts with a reference to the headless service you created above. While `serviceName` is not a required field according to the Kubernetes specification, some Kubernetes distributions and tools such as Helm expect it to be populated and will generate warnings or errors if you fail to provide a value.

Number of replicas

The `replicas` field identifies the number of pods that should be available in this `StatefulSet`. The value provided of 3 reflects the smallest Cassandra cluster that one might see in an actual production deployment, and most deployments are significantly larger, which is when Cassandra's ability to deliver high performance and availability at scale really begin to shine through.

Lifecycle management options

The `podManagementPolicy` and `updateStrategy` describe how Kubernetes should manage the rollout of pods when the cluster is scaling up or down, and how updates to the pods in the `StatefulSet` should be managed, respectively. We'll examine the significance of these values in *Managing the lifecycle of a StatefulSet*.

Pod specification

The next section of the `StatefulSet` specification is the template used to create each pod that is managed by the `StatefulSet`. The template has several subsections. First, under `metadata`, each pod includes a label `cassandra` that identifies it as being part of the set.

This template includes a single item in the `containers` field, a specification for a Cassandra container. The `image` field selects the latest version of the official Cassandra Docker image, which at the time of writing is Cassandra 4.0. This is where we diverge with the Kubernetes `StatefulSet` tutorial referenced above, which uses a custom Cassandra 3.11 image created specifically for that tutorial. Because the image we've chosen to use here is an official Docker image, you do not need to include registry or account information to reference it, and the name `cassandra` by itself is sufficient to identify the image that will be used.

Each pod will expose ports for various interfaces: a `cql` port for client use, intra-node and tls-intra-node ports for communication between nodes in the Cassandra cluster, and a `jmx` port for management via the Java Management Extensions (JMX).

The pod specification also includes instructions that help Kubernetes manage pod lifecycles, including a `readinessProbe`, a `livenessProbe`, and a `preStop` command. We'll learn how each of these are used below.

According to its documentation, the image we're using has been constructed to provide two different ways to customize Cassandra's configuration, which is...
stored in the cassandra.yaml file within the image. One way is to override the entire contents of the cassandra.yaml with a file that you provide. The second is to make use of environment variables that the image exposes to override a subset of Cassandra configuration options that are used most frequently. Setting these values in the env field causes the corresponding settings in the cassandra.yaml file to be updated:

- **CASSANDRA_CLUSTER_NAME** is used to distinguish which nodes belong to a cluster. Should a Cassandra node come into contact with nodes that don’t match its cluster name, it will ignore them.

- **CASSANDRA_DC** and **CASSANDRA_RACK** identify the datacenter and rack that each node will be a part of. This serves to highlight one interesting wrinkle of the way that StatefulSets expose a pod specification. Since the template is applied to each pod and container, there is no way to vary the configured datacenter and rack names between Cassandra pods. For this reason, it is typical to deploy Cassandra in Kubernetes using a StatefulSet per rack.

- **CASSANDRA_SEEDS** define well known locations of nodes in a Cassandra cluster that new nodes can use to bootstrap themselves into the cluster. The best practice is to specify multiple seeds in case one of them happens to be down or offline when a new node is joining. However, for this initial example, it’s enough to specify the initial Cassandra replica as a seed via the DNS name cassandra-0.cassandra.default.svc.cluster.local. We’ll look at a more robust way of specifying seeds in Chapter 4 using a service, as implied by the “Seed Service” shown in Figure 3-5.

The last item in the container specification is a volumeMount which requests that a PersistentVolume be mounted at the /var/lib/cassandra directory, which is where the Cassandra image is configured to store its data files. Since each pod will need its own PersistentVolumeClaim, the name cassandra-data is a reference to a PersistentVolumeClaim template which is defined below.

**Volume claim templates**

The final piece of the StatefulSet specification is the volumeClaimTemplates. The specification must include a template definition for each name referenced in one of the container specifications above. In this case, the cassandra-data template references the standard storage class we’ve been using in these examples. Kubernetes will use this template to create a PersistentVolumeClaim of the requested size of 1GB whenever it spins up a new pod within this StatefulSet.

**StatefulSet lifecycle management**

Now that we’ve had a chance to discuss the components of a StatefulSet specification, you can go ahead and apply the source:

```bash
kubectl apply -f cassandra-statefulset.yaml
```
As this gets applied, you can execute the following to watch as the StatefulSet spins up Cassandra pods:

```
kubectl get pods -w
```

Let's describe some of the behavior you can observe from the output of this command. First, you'll see a single pod cassandra-0. Once that pod has progressed to Ready status, then you'll see the cassandra-1 pod, followed by cassandra-2 after cassandra-1 is ready. This behavior is specified by the selection of podManagementPolicy for the StatefulSet. Let’s explore the available options and some of the other settings that help define how pods in a StatefulSet are managed.

The `podManagementPolicy` determines the timing of addition or removal of pods from a StatefulSet. The `OrderedReady` policy applied in our Cassandra example is the default. When this policy is in place and pods are added, whether on initial creation or scaling up, Kubernetes expands the StatefulSet one pod at a time. As each pod is added, Kubernetes waits until the pod reports a status of `Ready` before adding subsequent pods. If the pod specification contains a `readinessProbe` as we have done in this example, Kubernetes executes the provided command iteratively to determine when the pod is ready to receive traffic. When the probe completes successfully (i.e. with a zero return code), it moves on to creating the next pod. For Cassandra, readiness is typically measured by the availability of the CQL port (9042), which means the node is able to respond to CQL queries.

Similarly, when a StatefulSet is removed or scaled down, pods are removed one at a time. As a pod is being removed, any provided `preStop` commands for its containers are executed to give them a chance to shutdown gracefully. In our current example, the `nodetool drain` command is executed to help the Cassandra node exit the cluster cleanly, assigning responsibilities for its token range(s) to other nodes. as Kubernetes waits until a pod has been completely terminated before removing the next pod. The command specified in the `livenessProbe` is used to determine when the pod is alive, and when it no longer completes without error, Kubernetes can proceed to removing the next pod. See the Kubernetes documentation for more information on configuring readiness and liveness probes.

The other pod management policy is `Parallel`. When this policy is in effect, Kubernetes launches or terminates multiple pods at the same time in order to scale up or down. This has the effect of bringing your StatefulSet to the desired number of replicas more quickly, but it may also result in some stateful workloads taking longer to stabilize. For example, a database like Cassandra shuffles data between nodes when the cluster size changes in order to balance the load, and will tend to stabilize more quickly when nodes are added or removed one at a time.
With either policy, Kubernetes manages pods according to the ordinal numbers, always adding pods with the next unused ordinal numbers when scaling up, and deleting the pods with the highest ordinal numbers when scaling down.

**Pod Management Policies**

**Update Strategies**

The `updateStrategy` describes how pods in the StatefulSet will be updated if a change is made in the pod template specification, such as changing a container image. The default strategy is `RollingUpdate`, as selected in this example. With the other option, `OnDelete`, you must manually delete pods in order for the new pod template to be applied.

In a rolling update, Kubernetes will delete and recreate each pod in the StatefulSet, starting with the pod with the largest ordinal number and working toward the smallest. Pods are updated one at a time, and you can specify a number of pods called a partition in order to perform a phased rollout or canary. Note that if you discover a bad pod configuration during a rollout, you’ll need to update the pod template specification to a known good state and then manually delete any pods that were created using the bad specification. Since these pods will not ever reach a `Ready` state, Kubernetes will not decide they are ready to replace with the good configuration.

Note that Kubernetes offers similar lifecycle management options for Deployments, ReplicaSets and DaemonSets including revision history.

---

**More sophisticated lifecycle management for StatefulSets**

One interesting set of opinions on additional lifecycle options for StatefulSets comes from OpenKruise, a CNCF Sandbox project, which provides an Advanced StatefulSet. The Advanced StatefulSet adds capabilities including:

- Parallel updates with a maximum number of unavailable pods
- Rolling updates with an alternate order for replacement, based on a provided prioritization policy
- Updating pods “in-place” by restarting their containers according to an updated pod template specification

This Kubernetes resource is also named StatefulSet to facilitate its use with minimal impact to your existing configurations. You just need to change the `apiVersion` from `apps/v1` to `apps.kruise.io/v1beta1`.
We recommend getting more hands-on experience with managing StatefulSets in order to reinforce your knowledge. For example, you can monitor the creation of PersistentVolumeClaims as a StatefulSet scales up. Another thing to try: delete a StatefulSet and recreate it, verifying that the new pods recover previously stored data from the original StatefulSet. For more ideas, you may find these guided tutorials helpful: StatefulSet Basics from the Kubernetes documentation, and StatefulSet: Run and Scale Stateful Applications Easily in Kubernetes from the Kubernetes blog.

StatefulSets are extremely useful for managing stateful workloads on Kubernetes, and that’s not even counting some capabilities we didn’t address, such as pod affinity, anti-node affinity, managing resource requests for memory and CPU, and availability constraints such as PodDisruptionBudgets. On the other hand, there are capabilities you might desire that StatefulSets don’t provide, such as backup/restore of persistent volumes, or secure provisioning of access credentials. We’ll discuss how to leverage or build these capabilities on top of Kubernetes in Chapter 4 and beyond.
issues where a pod is stuck in a pending state, or cannot run, and just restart these pods. However, the thing to remember with StatefulSets is that protecting the underlying data is the most important priority. We could end up making the suggested change in order to allow faster updates in parallel for development environments where data protection is less of a concern, but require opting in with a feature flag.

Another frequently requested feature is the ability to auto-delete the PersistentVolumeClaims of a StatefulSet when the StatefulSet is deleted. The original behavior is to preserve the PVCs, again as a data protection mechanism, but there is a Kubernetes Enhancement Proposal (KEP) for auto-deletion that is under consideration for the Kubernetes 1.23 release.

Even though there are some significant differences in the way StatefulSets manage pods versus other controllers, we are working to make the behaviors more similar across the different controllers as much as possible. One example is the addition of a `minReadySeconds` setting in the pod template, which allows you to say, I'd like this application to be unavailable for a little bit of extra time before sending traffic to it. This is helpful for some stateful workloads that need a bit more time to initialize themselves, for example to warm up caches, and brings StatefulSets in line with other controllers.

Another example is the work that is in progress to unify status reporting across all of the application controllers. Currently, if you’re building any kind of higher level orchestration or management tools, you need to have different behavior to handle the status of StatefulSets, Deployments, DaemonSets, and so on, because each of them was written by a different author. Each author had a different requirement for what should be in the status, how the resource should express information about whether it’s available, or whether it’s in a rolling update, or it’s unavailable, or whatever is happening with it. DaemonSets are especially different in how they report status.

There is also a feature in progress that allows you to set a `maxUnavailable` number of pods for a StatefulSet. This number would be applied during the initial rollout of a StatefulSet and allow the number of replicas to be scaled up more quickly. This is another feature that brings StatefulSets into greater alignment with how the other controllers work. If you want to understand the work that is in progress from the SIG Apps team, the best way is to look at Kubernetes open issues that are labeled `sig/apps`.

It can be difficult to build StatefulSets as a capability that will meet the needs of all stateful workloads; we’ve tried to build them in such a way as to handle the most common requirements in a consistent way. We could obviously add support for more and more edge cases, but this tends to make the functionality significantly more complicated for users to grasp. There will always be users who are dissatisfied because their use case is not covered, and there’s always a balance of how much we can put in without affecting both functionality and performance.
In most cases where users need more specific behaviors, for example to handle edge cases, it’s because they’re trying to manage a complex application like Postgres or Cassandra. That’s where there’s a great argument for creating your own controllers and even operators to deal with those specific cases. Even though it might sound super scary, it’s really not that difficult to write your own controller. You can start reasonably quickly and get a basic controller up and running in a couple of days using some simple examples including the sample controller, which is part of the Kubernetes code base and maintained by the project. The O’Reilly book Programming Kubernetes also has a chapter on writing controllers. Don’t just assume you’re stuck with the behavior that comes out of the box. Kubernetes is meant to be open and extensible, whether it’s networking, controllers, CSI, plugins, and more. If you need to customize Kubernetes, you should go for it!

### Accessing Cassandra

Once you have applied the configurations listed above, you can use Cassandra’s CQL shell cqlsh to execute CQL commands. If you happen to be a Cassandra user and have a copy of cqlsh installed on your local machine, you could access Cassandra as a client application would, using the CQL Service associated with the StatefulSet. However, since each Cassandra node contains cqlsh as well, this gives us a chance to demonstrate a different way to interact with infrastructure in Kubernetes, by connecting directly to an individual pod in a StatefulSet:

```bash
kubectl exec -it cassandra-0 -- cqlsh
```

This should bring up the cqlsh prompt and you can then explore the contents of Cassandra’s built in tables using `DESCRIBE KEYSPACES` and then `USE` to select a particular keyspace and run `DESCRIBE TABLES`. There are many Cassandra tutorials available online that can guide you through more examples of creating your own tables, inserting and querying data, and more. When you’re done experimenting with cqlsh, you can type `exit` to exit the shell.

Removing a StatefulSet is the same as any other Kubernetes resource - you can delete it by name, for example:

```bash
kubectl delete sts cassandra
```

You could also delete the StatefulSet referencing the file used to create it:

```bash
kubectl delete sts cassandra
```

When you delete a StatefulSet with a policy of Retain as in this example, the PersistentVolumeClaims it creates are not deleted. If you recreate the StatefulSet, it will bind to the same PVCs and reuse the existing data. When you no longer need the claims, you’ll need to delete them manually. The final cleanup from this exercise...
What about DaemonSets?

If you’re familiar with the resources Kubernetes offers for managing workloads, you may have noticed that we haven’t yet mentioned DaemonSets. DaemonSets allow you to request that a pod be run on each worker node in a Kubernetes cluster, as shown in Figure 3-6. Instead of specifying a number of replicas, a DaemonSet scales up or down as worker nodes are added or removed from the cluster. By default, a DaemonSet will run your pod on each worker node, but you can use taints and tolerations to override this behavior, for example, limiting some worker nodes, or selecting to run pods on Kubernetes master nodes as well. DaemonSets support rolling updates in a similar way to StatefulSets.

Figure 3-6. Daemon Sets run a single pod on selected worker nodes

On the surface, DaemonSets might sound useful for running databases or other data infrastructure, but this does not seem to be a widespread practice. Instead, DaemonSets are most frequently used for functionality related to worker nodes and their relationship to the underlying Kubernetes provider. For example, many of the Container Storage Interface (CSI) implementations that we saw in Chapter 2 use DaemonSets to run a storage driver on each worker node. Another common usage is to run pods that perform monitoring tasks on worker nodes, such as log and metrics collectors.
Summary

In this chapter we’ve learned how to deploy both single node and multi-node distributed databases on Kubernetes with hands-on examples. Along the way you’ve gained familiarity with Kubernetes resources such as Deployments, ReplicaSets, StatefulSets, and DaemonSets, and learned about the best use cases for each:

- Use Deployments/ReplicaSets to manage stateless workloads or simple stateful workloads like single-node databases or caches that can rely on ephemeral storage
- Use StatefulSets to manage stateful workloads that involve multiple nodes and require association with specific storage locations
- Use DaemonSets to manage workloads that leverage specific worker node functionality

You’ve also learned the limits of what each of these resources can provide. Now that you’ve gained experience in deploying stateful workloads on Kubernetes, the next step is to learn how to automate the so-called “day 2” operations involved in keeping this data infrastructure running.
In the previous chapter, you learned how to deploy both single node and multi-node databases on Kubernetes by hand, creating one element at a time. We took the “hard way” route on purpose to help maximize your understanding of how to leverage Kubernetes primitives in order to set up the compute, network and storage resources that a database requires. Of course, this doesn't represent the experience of running databases in production on Kubernetes, for a couple of reasons.

First, teams typically don't deploy databases by hand, one yaml file at a time. That can get pretty tedious. And even combining the configurations into a single file could start to get pretty complicated, especially for more sophisticated deployments. Consider the increase in the amount of configuration required in Chapter 3 for Cassandra as a multi-node database compared with the single-node MySQL deployment. In this
chapter, we’ll look at tools that help standardize the deployment of databases and other applications, reducing the amount of configuration code you have to write.

Second, while deploying a database is great, what about keeping it running over time? You need our data infrastructure to remain reliable and performant over the long haul, and data infrastructure is known for requiring a lot of care and feeding. Put another way, the task of running a system is often divided into “day 1” - the joyous day when you deploy an application to production, and “day 2” - which represents every day after the first, where you need to operate and evolve your application while maintaining high availability. We’ll start to address data infrastructure operations in these next two chapters and carry that theme throughout the remainder of the book.

**Deploying Applications with Helm charts**

First, let’s take a look at a tool that helps you manage the complexity of managing configurations: Helm. Helm is a package manager for Kubernetes which is open source and a CNCF graduated project. The concept of a package manager is a common one across multiple programming languages, such as pip for Python, the Node Package Manager (NPM) for JavaScript, and Ruby’s Gems feature. There are also package managers for specific operating systems, such as Apt for Linux, or Homebrew for MacOS. As shown in Figure 4-1, the essential elements of a package manager system are the packages, the registries where the packages are stored, and the package manager application (or “client”) which helps the user locate, install, and update packages on their local systems.

![Figure 4-1. Helm, a Package Manager for Kubernetes](image)

Helm extends the package management concept to Kubernetes, with some interesting differences. If you’ve worked with one of the package managers listed above, you’ll be familiar with the idea that a package consists of a binary (executable code) as well as metadata describing the binary, such as its functionality, API, and installation instructions. In Helm, the packages are called *charts*. However, instead of containing
binaries directly, charts provide a description for how to build a Kubernetes application piece by piece using the Kubernetes resources for compute, networking, and storage introduced in previous chapters. The binaries are container images referenced in compute workload definitions within the charts that reside in public or private container registries.

Helm allows charts to reference other charts as dependencies, which provides a great way to compose applications by creating assemblies of charts. For example, you could define an application such as the Wordpress / MySQL example from the previous chapter by defining a chart for your Wordpress deployment that referenced a chart defining a MySQL deployment that you wish to reuse. Or, you might even find a Helm chart that defines an entire Wordpress application including the database.

Kubernetes environment prerequisites

The examples in this chapter assume you have access to a Kubernetes cluster with couple of characteristics:

- The cluster should have at least 3 worker nodes, in order to demonstrate mechanisms Kubernetes provides to allow you to request pods to be spread across a cluster. You can create a simple cluster on your desktop using an open source distribution called Kind. See the Kind quick start guide for instructions on installing Kind and creating a multi-node cluster. The code for this example also contains a configuration file you may find useful to create a simple three-node Kind cluster (source code).

- You will also need a StorageClass that supports dynamic provisioning. You may wish to follow the instructions in the StorageClasses classes section for installing a simple StorageClass and provisioner that expose local storage (source code).

Using Helm to deploy MySQL

To make things a bit more concrete, let's use Helm to deploy the databases you worked with in Chapter 3. First, you'll need to install Helm on your system using the documentation on the Helm website. Next, add the Bitnami Helm repository:

```bash
helm repo add bitnami https://charts.bitnami.com/bitnami
```

The Bitnami Helm repository contains a variety of Helm charts to help you deploy infrastructure such as databases, analytics engines, and log management systems, as well as applications including e-commerce, customer relationship management (CRM), and you guessed it: Wordpress. You can find the source code for the charts in the Bitnami Charts repository on GitHub. The README for this repo provides helpful instructions for using the charts in various Kubernetes distributions.
Now, let's use the Helm chart provided in the bitnami repository to deploy MySQL. In Helm’s terminology, each deployment is known as a *release*. The simplest possible release that you could create using this chart would look something like this:

```bash
# don’t execute me yet!
helm install mysql bitnami/mysql
```

If you execute this command, it will create a release called *mysql* using the Bitnami MySQL Helm chart with its default settings. As a result you’d have a single MySQL node. Since you’ve already deployed a single node of MySQL manually in Chapter 3, let’s do something a bit more interesting this time and create a MySQL cluster. To do this you’ll create a `values.yaml` file with contents like this (or reuse the sample provided in the source code):

```yaml
architecture: replication
secondary:
  replicaCount: 2
```

The settings in this `values.yaml` file let Helm know that you want to use options in the Bitnami MySQL Helm chart to deploy MySQL in a replicated architecture in which there is a primary node and two secondary nodes.

### MySQL Helm chart configuration options

If you examine the default `values.yaml` file provided with the Bitnami MySQL Helm chart, you’ll see that there are quite a few options available beyond the simple selections shown here. The configurable values include the following:

- Images to pull and their locations
- The Kubernetes StorageClass that will be used to generate PersistentVolumes
- Security credentials for user and administrator accounts
- MySQL configuration settings for primary and secondary replicas
- Number of secondary replicas to create
- Details of liveness, readiness probes
- Affinity and anti-affinity settings
- Managing high availability of the database using pod disruption budgets

Many of these concepts you’ll be familiar with already, and others like affinity and pod disruption budgets will be covered later in the chapter.
Once you’ve created the `values.yaml` file, you can start the cluster using this command:

```
helm install mysql bitnami/mysql -f values.yaml
```

After running the command you’ll see the status of the install from Helm, plus instructions that are provided with the chart under NOTES:

```
NAME: mysql
LAST DEPLOYED: Thu Oct 21 20:39:19 2021
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
...
```

We’ve omitted the notes here since they are a bit lengthy. They describe suggested commands for monitoring the status as MySQL initializes, how clients and administrators can connect to the database, how to upgrade the database, and more.

**Use Namespaces to help isolate resources**

One thing you’ll notice from the output above is that the Helm release has been installed in the default Kubernetes namespace. If you want to install a Helm release in its own namespace in order to work with its resources more effectively, you could run something like the following:

```
helm install mysql bitnami/mysql --namespace mysql --create-namespace
```

This creates a namespace called `mysql` and installs the `mysql` release inside of it.

In order to obtain information about the Helm releases you’ve created, use the `helm list` command, which produces output such as this (formatted for readability):

```
NAME   NAMESPACE  REVISION  UPDATED
mysql  default    1         2021-10-21 20:39:19

STATUS    CHART        APP VERSION
deployed  mysql-8.8.8  8.0.26
```

If you haven’t installed the release in its own namespace, it’s still simple to see the compute resources that Helm has created on your behalf, because they have all been labeled with the name of your release:

```
NAME                   READY   STATUS    RESTARTS   AGE
pod/mysql-primary-0    1/1     Running   0          3h40m
```
As you can see, Helm has created two StatefulSets, one for primary replicas and one for secondary replicas. The `mysql-primary` StatefulSet is managing a single MySQL pod containing a primary replica, while the `mysql-secondary` StatefulSet is managing two MySQL pods containing secondary replicas. See if you can determine which Kubernetes worker node each MySQL replica is running on using the `kubectl describe pod` command.

From the output above, you’ll also notice two services created for each StatefulSet, one a headless service and another that has a dedicated IP address. Since `kubectl get all` only tells you about compute resources and services, you might also be wondering about the storage resources. To check on these, run the `kubectl get pv` command. Assuming you have a StorageClass installed that supports dynamic provisioning, you should see PersistentVolumes that are bound to PersistentVolumeClaims named `data-mysql-primary-0`, `data-mysql-secondary-0`, and `data-mysql-secondary-1`.

In addition to the resources we’ve discussed above, installing the chart has also resulted in the creation of a few additional resources which we’ll explore below.
How Helm Works

Did you wonder what happened when you executed the `helm install` command with a provided values file? To understand what’s going on, let’s take a look at the contents of a Helm chart, as shown in Figure 4-2. As we discuss these contents, it will also be helpful to look at the source code of the MySQL Helm chart you just installed.

Looking at the contents of a Helm chart, you’ll notice the following:

- A `README` file explaining how to use the chart. These instructions are provided along with the chart in registries.
- A `Chart.yaml` file containing metadata about the chart such as its name, publisher, version, keywords, and any dependencies on other charts. These properties are useful when searching Helm registries to find charts.
- A `values.yaml` file listing out the configurable values supported by the chart and their default values. These files typically contain a good amount of comments that explain the available options. For the Bitnami MySQL Helm chart, there are a lot of available options, as we noted above.
- A `templates` directory containing Go templates that define the chart. The templates include a `Notes.txt` file which is used to generate the output you saw above after executing the `helm install` command, and one or more `yaml` files that describe a pattern for a Kubernetes resource. These yaml files may be organized

---

**Figure 4-2. Customizing a Helm release using a values.yaml file**

Using Helm to deploy MySQL | 95
in subdirectories, for example, the `template` that defines a StatefulSet for MySQL primary replicas. Finally, there is a `_helpers.tpl` file that describes how to use the templates. Some of the templates may be used multiple times, or not at all, depending on the selected configuration values.

When you execute the `helm install` command, the Helm client makes sure it has an up-to-date copy of the chart you've named by checking with the source repository. Then it systematically applies the template to generate yaml configuration code, in the process overriding default values from the chart's `values.yaml` file with any values you've provided. It then uses the `kubectl` command to apply this configuration to your currently configured Kubernetes cluster.

If you'd like to see the configuration that a Helm chart will produce before applying it, there's a handy `template` command you can use. It supports the same syntax as the `install` command:

```bash
helm template mysql bitnami/mysql -f values.yaml
```

Running this command will produce quite a bit of output, so you may want to pipe it to a file (append “> values-template.yaml” to the command) so you can take a longer look. Alternatively, you can look at the copy we have saved in the source code repository.

You'll notice that there are several different types of resources created, as summarized in Figure 4-3. Many of the resources shown have been discussed above, including the StatefulSets for managing the primary and secondary replicas, each with its own service (the chart also creates headless services which are not shown in the figure). Each pod has its own PersistentVolumeClaim which is mapped to a unique Persistent Volume.

Figure 4-3 also includes resource types we haven't discussed previously. Notice first that each StatefulSet has an associated ConfigMap that is used to provide a common set of configuration settings to its pods. Next, notice that there is a Secret named `mysql`, which stores passwords needed for accessing various interfaces exposed by the database nodes. Finally, there is a ServiceAccount resource, which is applied to every pod created by this Helm release.
Let’s focus on some interesting aspects of this deployment, including the usage of Labels, ServiceAccounts, Secrets, and ConfigMaps.

**Labels**

If you look through the output from the `helm template`, you’ll notice that the resources have a common set of labels:

```yaml
labels:
  app.kubernetes.io/name: mysql
  helm.sh/chart: mysql-8.8.8
  app.kubernetes.io/instance: mysql
  app.kubernetes.io/managed-by: Helm
```

These labels help identify the resources as being part of the `mysql` application and indicate that they are managed by Helm using a specific chart version. The labels are
useful for selecting resources, which is often useful in defining configurations for other resources.

**ServiceAccounts**

Kubernetes clusters make a distinction between human users and applications for access control purposes. A ServiceAccount is a Kubernetes resource that represents an application and what it is allowed to access. For example, a ServiceAccount may be given access to some portions of the Kubernetes API, or access to one or more secrets containing privileged information such as login credentials. This latter capability is used in your Helm installation of MySQL to share credentials between pods.

Every pod that is created in Kubernetes has a ServiceAccount assigned. If you do not specify one then the default ServiceAccount is used. Installing the MySQL Helm chart creates a ServiceAccount called `mysql`. You can see the specification for this resource in the generated template:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mysql
  namespace: default
  labels: ...
  annotations:
    secrets:
      - name: mysql
```

As you can see, this ServiceAccount has access to a secret called `mysql` which we'll discuss shortly, but does not make use of other capabilities of ServiceAccounts:

- A ServiceAccount can also have an additional type of secret known as an `imagePullSecret`. These secrets are used when an application needs to use images from a private registry.
- By default a ServiceAccount does not have any access to the Kubernetes API. To add this you would create a RoleBinding.

See the Kubernetes documentation for more details on managing ServiceAccounts and configuring them for role-based access.

**Secrets**

As you learned in Chapter 2, a Secret provides secure access to information you need to keep private. Your `mysql` Helm release contains a Secret called `mysql` containing login credentials for the MySQL instances themselves:

```yaml
apiVersion: v1
kind: Secret
```
The three different passwords represent different types of access: the `mysql-root-password` provides administrative access to the MySQL node, while the `mysql-replication-password` is used for nodes to communicate for the purposes of data replication between nodes. The `mysql-password` is used by client applications to access the database to write and read data.

**ConfigMaps**

The Bitnami MySQL Helm chart creates Kubernetes ConfigMap resources to represent the configuration settings used for pods that run the MySQL primary and secondary replica nodes. ConfigMaps store configuration data as key-value pairs. For example, the ConfigMap created by the Helm chart for the primary replicas looks like this:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-primary
  namespace: default
  labels: ...
data:
  my.cnf: |-

  [mysqld]
  default_authentication_plugin=mysql_native_password

...
```

In this case, the key is the name `my.cnf`, which represents a filename, and the value is a multi-line set of configuration settings which represent the contents of a configuration file (which we've abbreviated here). Next, look at the definition of the StatefulSet for the primary replicas. Notice how the contents of the ConfigMap are mounted as a read-only file inside each template, according to the pod specification for the StatefulSet (again, we've omitted some detail to focus on key areas):

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-primary
  namespace: default
  labels: ...
```
Mounting the ConfigMap as a volume in a container results in the creation of a read-only file in the mount directory which is named according to the key and has the value as its content. For our example, mounting the ConfigMap in the pod’s mysql container results in the creation of the file `/opt/bitnami/mysql/conf/my.cnf`.

This is just one of several ways that ConfigMaps can be used in Kubernetes applications:

- As described in the Kubernetes documentation, you could choose to store configuration data in more granular key-value pairs, which also makes it easier to access individual values in your application
- You can also reference individual key-value pairs as environment variables you pass to a container
- Finally, applications can access ConfigMap contents via the Kubernetes API
More configuration options

Now that you have a Helm release with a working MySQL cluster, you can point an application to it, such as Wordpress. Why not try seeing if you can adapt the Wordpress deployment from Chapter 3 to point to the MySQL cluster you’ve created here.

For further learning, you could also compare your resulting configuration with that produced by the Bitnami Wordpress Helm Chart, which uses MariaDB instead of MySQL but is otherwise quite similar.

Updating Helm Charts

If you’re running a Helm release in a production environment, chances are you’re going to need to maintain it over time. There are several reasons why you might want to update a Helm release:

- A new version of a chart is available
- A new version of an image used by your application is available
- You want to change the selected options

To check for a new version of a chart, execute the `helm repo update` command. Running this command with no options looks for updates in all of the chart repositories you have configured for your helm client:

```
helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "bitnami" chart repository
Update Complete. ⎈Happy Helming!⎈
```

Next, you’ll want to make any desired updates to your configured values. If you’re upgrading to a new version of a chart, make sure to check the release notes and documentation of the configurable values. It’s a good idea to test out an upgrade before applying it. The `--dry-run` option allows you to do this, producing similar values to the `helm template` command:

```
helm upgrade mysql bitnami/mysql -f values.yaml --dry-run
```
Using an overlay configuration file

One useful option you could use for the upgrade is to specify values you wish to override in a new configuration file, and apply both the new and old, something like this:

```
helm upgrade mysql bitnami/mysql -f values.yaml -f new-values.yaml
```

Note that configuration files are applied in the order they appear on the command line, so if you use this approach, make sure your overridden values file appears after your original values file.

Once you’ve applied the upgrade, Helm sets about its work, only updating resources in the release that are affected by your configuration changes. If you’ve specified changes to the pod template for a StatefulSet, the pods will be restarted according to the update policy specified for the StatefulSet, as we discussed in Managing the lifecycle of a StatefulSet.

Uninstalling Helm charts

When you are finished using your Helm release, you can uninstall it by name like this:

```
helm uninstall mysql
```

Note that Helm does not remove any of the PersistentVolumeClaims or PersistentVolumes that were created for this Helm chart, following the behavior of StatefulSets discussed in Chapter 3.

Additional Deployment Tools: Kustomize and Skaffold

In addition to Helm, other tools in the Kubernetes ecosystem are available to help you manage the configuration and deployment of applications, such as Kustomize and Skaffold.

Kustomize is a configuration management tool for Kubernetes. Unlike a package manager, Kustomize does not provide a registry; instead its focus is helping you manage Kubernetes configuration yaml files for different environments. Kustomize uses a template based approach in which you create snippets of configuration code called overlays which are intended to override sections of a base yaml file. These overlays are typically intended for different environments such as development, test, and production, or for isolating configuration specific to different Kubernetes providers, with a similar effect to a Helm values.yaml file. The sections to be overridden are identified by selectors such as Kubernetes labels or annotations. You provide a kustomization.yaml file to describe the mapping of templates to their selectors. Kustomize
works best when the yaml file you want to customize is well structured and makes use of labels or annotations.

Skaffold is a tool that automates application deployment in your development environment. You can execute Skaffold imperatively from the command line, or as a daemon that watches for code changes to build artifacts such as container images. When it detects a relevant change, the daemon automatically performs actions according to the workflow you define in a skaffold.yaml file. The workflow can include actions such as building and tagging images, updating Helm charts or regular Kubernetes configuration files, and deploying your app using kubectl, helm, or Kustomize.

Using Helm to deploy Apache Cassandra

Now let's switch gears and look at deploying Apache Cassandra using Helm. In this section, you'll use another chart provided by Bitnami, so there's no need to add another repository. You can find the implementation of this chart on GitHub. Helm provides a quick way to see the metadata about this chart:

```
helm show chart bitnami/cassandra
```

After reviewing the metadata, you'll also want to learn about the configurable values. You can examine the values.yaml file in the GitHub repo, or use another option on the show command:

```
helm show values bitnami/cassandra
```

The list of options for this chart is shorter than the list for the MySQL chart, because Cassandra doesn't have the concept of primary and secondary replicas. However, you'll certainly see similar options for images, storage classes, security, liveness and readiness probes, and so on. There are also configuration options that are unique to Cassandra, such as those having to do with JVM settings and seed nodes (as discussed in Chapter 3).

One interesting feature of this chart is the ability to export metrics from Cassandra nodes. If you set `metrics.enabled=true`, the chart will inject a sidecar container into each Cassandra pod that exposes a port which can be scraped by Prometheus. Other values under `metrics` configure what metrics are exported, the collection frequency, and more. While we won’t use this feature here, metrics reporting is a key part of managing data infrastructure we'll cover in Chapter 6.

For a simple three-node Cassandra configuration, you could set the replica count to three and let other configuration values to their defaults. However, since you're only overriding a single configuration value, this is a good time to take advantage of
Helm’s support for setting values on the command line, instead of providing a `values.yaml` file:

```
helm install cassandra bitnami/cassandra --set replicaCount=3
```

As discussed above, you can use the `helm template` command to check the configuration before installing it, or look at the file we’ve saved on GitHub. However, since you’ve already created the release, you can also use this command:

```
helm get manifest cassandra
```

Looking through the resources in the yaml, you’ll see a similar set of infrastructure has been established, as shown in Figure 4-4:

![Diagram of deploying Apache Cassandra using the Bitnami Helm Chart](image)

*Figure 4-4. Deploying Apache Cassandra using the Bitnami Helm Chart*

The configuration includes:

- A ServiceAccount referencing a Secret, which contains the password for the Cassandra administrator account.
- A single StatefulSet, with a headless Service used to reference its Pods. The Pods are spread evenly across the available Kubernetes worker nodes, which we'll discuss momentarily under Affinity and Anti-Affinity. The Service exposes Cassandra ports used for intra-node communication (7000, with 7001 used for secure communication via TLS), administration via JMX (7199), and client access via CQL (9042).

This configuration represents a very simple Cassandra topology, with all three nodes in a single datacenter and rack. This simple topology reflects one of the limitations of this chart - it does not provide the ability to create a Cassandra cluster consisting of multiple datacenters and racks. To create a more complex deployment, you'd have to install multiple Helm releases, using the same `clusterName` (in this case you're using the default name `cassandra`), but a different datacenter and rack per deployment. You'd also need to obtain the IP address of a couple of nodes in the first datacenter to use as `additionalSeeds` when configuring the releases for the other racks.

**Affinity and Anti-Affinity**

As shown in Figure 4-4, the Cassandra nodes are spread evenly across the worker nodes in your cluster. To verify this in your own Cassandra release, you could run something like the following:

```bash
cubectl describe pods | grep "^Name:" -A 3
Name:         cassandra-0
Namespace:    default
Priority:     0
Node:         kind-worker/172.20.0.7
--
Name:         cassandra-1
Namespace:    default
Priority:     0
Node:         kind-worker2/172.20.0.6
--
Name:         cassandra-2
Namespace:    default
Priority:     0
Node:         kind-worker3/172.20.0.5
```

As you can see in this output, each Cassandra node is running on a different worker node. If your Kubernetes cluster has at least 3 worker nodes and no other workloads, you'll likely observe similar behavior. While it is true that this even allocation could happen naturally in a cluster that has an even load across worker nodes, this is probably not the case in your production environment. However, in order to promote maximum availability of your data, we want to try to honor the intent of Cassandra’s architecture to run nodes on different machines in order to promote high availability.
In order to help guarantee this isolation the Bitnami Helm chart makes use of Kubernetes's affinity capabilities, specifically anti-affinity. If you examine the generated configuration for the cassandra StatefulSet, you'll see the following:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  namespace: default
labels: ...
spec:
  ...
  template:
    metadata:
      labels: ...
    spec:
      ...
      affinity:
        podAffinity:
          podAntiAffinity:
            preferredDuringSchedulingIgnoredDuringExecution:
              - podAffinityTerm:
                  labelSelector:
                    matchLabels:
                    app.kubernetes.io/name: cassandra
                    app.kubernetes.io/instance: cassandra
                  namespaces:
                    - "default"
                  topologyKey: kubernetes.io/hostname
                  weight: 1
    nodeAffinity:
```

As shown here, the pod template specification lists three possible types of affinity, with only the `podAntiAffinity` being defined. What do these concepts mean?

**Pod Affinity**
Pod affinity refers to a preference that a Pod is scheduled onto a node where another specific Pod is running. For example, pod affinity could be used to colocate a web server with its cache.

**Pod Anti-Affinity**
Pod anti-affinity is the opposite of Pod affinity; that is, a preference that a pod not be scheduled on a node where another identified Pod is running. This is the constraint used in this example, as we'll discuss shortly.

**Node Affinity**
Node affinity is a preference that a Pod be run on a node with specific characteristics.
Each of these types of affinity can be expressed as either hard or soft constraints, known as

requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. The first constraint specifies rules that must be met before a Pod is scheduled on a node, while the second specifies a preference that the scheduler will attempt to meet, but may relax if necessary in order to schedule the Pod. The IgnoredDuringExecution reference in these names implies that the constraints only apply when the Pods are first scheduled. In the future, new RequiredDuringExecution options will be added called requiredDuringSchedulingRequiredDuringExecution and requiredDuringSchedulingIgnoredDuringExecution. These will ask Kubernetes to evict pods (that is, move them to another node) that no longer meet the criteria, for example by a change in their labels.

Looking at the example above, the Pod template specification for the Cassandra StatefulSet specifies an anti-affinity rule using the labels that are applied to each Cassandra pod. The net effect of this is that Kubernetes will try to spread the Pods across the available worker nodes.

More Kubernetes scheduling constraints

Kubernetes supports additional mechanisms for providing hints to its scheduler about Pod placement. One of the simplest is NodeSelectors, which is very similar to node affinity, but with a less expressive syntax that can match on one or more labels using AND logic. Since you may or may not have the required privileges to attach labels to worker nodes in your cluster, pod affinity is often a better option. Taints and tolerations are another mechanism that can be used to configure worker nodes to repel specific pods from being scheduled on those nodes.

In general, you want to be careful to understand all of the constraints you’re putting on the Kubernetes scheduler from various workloads so as not to overly constrain its ability to place Pods. See the Kubernetes documentation for more information on scheduling constraints.

Those are the highlights of looking at the Bitnami Helm chart for Cassandra. To clean things up, uninstall the Cassandra release:

```
helm uninstall cassandra
```

If you don’t want to work with Bitnami Helm charts any longer, you can also remove the repository from your Helm client:
Helm Limitations

While Helm is a powerful tool for deploying of complex applications to Kubernetes clusters, it has some limitations, especially when it comes to managing the operations of those applications. To get a good picture of the challenges involved, we spoke to a practitioner who has built assemblies of Helm charts to manage a complex database deployment. This discussion begins to introduce concepts like Kubernetes Custom Resource Definitions (CRDs) and the operator pattern, both of which we'll cover in depth in Chapter 5.

### Pushing Helm to the limit

With John Sanda, Software Engineer, DataStax

K8ssandra is a distribution of Apache Cassandra on Kubernetes built from multiple open source components, including a Cassandra operator (cass-operator), and operational tools for managing anti-entropy repair (Reaper) and backups (Medusa). K8ssandra also includes the Prometheus-Grafana stack for metrics collection and reporting.

From the start, we used Helm to help manage the installation and configuration of these components. Helm enabled us to quickly bootstrap the project and attract developers in the Cassandra community who didn't necessarily have much Kubernetes expertise and experience. Many of these folks found it easy to grasp a package management tool and installer like Helm.

As the project grew, we began to run into some limitations with Helm. While it was pretty straightforward to get the installation of K8ssandra clusters working correctly, we encountered more issues when it came to upgrading and managing clusters.

**Writing complex logic**

Helm has good support for control flow, with loops and if statements. However, when you start getting multiple levels deep, it's harder to read and reason through the code, and indentation becomes an issue. In particular, we found that peer-reviewing changes to Helm charts became quite difficult.

**Reuse and extensibility**

Helm variables are limited to the scope of the template where you declare them, which meant we had to recreate the same variables in multiple templates. This prevented us from keeping our code DRY, which we found to be a source of defects.

Similarly, Helm has a big library of helper template functions, but that library doesn't cover every use case, and there is no interface to define your own func-
tions. You can define your own custom templates, which allow for a lot of reuse, but those are not a replacement for functions.

**Project structure and inheritance**

We also ran into difficulties as we tried to implement an umbrella chart design pattern, which is a best practice for Helm. We were able to create a top-level K8ssandra Helm chart with sub-charts for Cassandra and Prometheus but ran into problems with variable scoping when attempting to create additional sub-charts. Our intent was to define authentication settings in the top-level chart and push them down to sub-charts, but this functionality is not supported by the Helm inheritance model.

**Custom resource management**

Helm can create Kubernetes custom resources, but it doesn't manage them. This was a deliberate design choice that the Helm developers made for Helm 3. Because the definition of a custom resource is cluster-wide, it can get confusing if multiple Helm installs are trying to work off of different versions of a CRD. This presented us with some difficulties in managing updates to resources like a Cassandra datacenter within Helm. The workaround we implemented was to implement custom Kubernetes jobs labeled as pre-upgrade hooks that Helm would execute on an upgrade. At some point, writing these jobs began to feel like we were writing an operator.

**Multi-cluster deployments**

While we've been able to work around these Helm challenges in many cases, the next major feature on our roadmap was implementing Cassandra clusters that spanned multiple Kubernetes clusters. We realized that even without the intricacies of the network configuration, this was going to be a step beyond what we could implement effectively using Helm.

In the end, we realized that we were trying to make Helm do too much. It's easy to get into a situation where you learn how to use the hammer and everything looks like a nail, but what you really need is a screwdriver. However, we don't see Helm and operators as mutually exclusive. These are complementary approaches and we need to use each one in terms of its strengths. We continue to use Helm to perform basic installation actions including installing operators and setting up the administrator service account used by Cassandra and other components; these are the sort of actions that package managers like Helm do best.

Note: this section was adapted from the post We Pushed Helm to the Limit, then Built a Kubernetes Operator.

---

**Summary**

In this chapter, you’ve learned how a package management tool like Helm can help you to manage the deployment of applications on Kubernetes, including your data-
base infrastructure. Along the way you’ve also learned how to use some additional
Kubernetes resources like ServiceAccounts, Secrets, and ConfigMaps. Now it’s time to
round out our discussion of running databases on Kubernetes. In the next chapter,
we’ll take a deeper dive into managing database operations on Kubernetes using the
operator pattern.
In this chapter we’ll continue our exploration of running databases on Kubernetes, but shift our focus from installation to operations. It’s not enough just to know how the elements of a database application map onto the primitives provided by Kubernetes for an initial deployment. You also need to know how to maintain that infrastructure over time in order to support your business-critical applications. In this chapter, we’ll take a look at the Kubernetes approach to operations so that you can keep databases running effectively.

Operations for databases and other data infrastructure consist of a common list of “day 2” tasks such as:

- scaling capacity up and down, including reallocating workload across resized clusters
• monitoring database health and replacing failed (or failing) instances
• performing routine maintenance tasks, such as repair operations in Apache Cassandra
• updating and patching software
• maintaining secure access keys and other credentials which may expire over time
• performing backups, and using backups to restore data in disaster recovery scenarios

While the details of how these tasks are performed may vary between technologies, the common concern is how we can use automation to reduce the workload on human operators and enable us to operate infrastructure at larger and larger scales. How can we incorporate the knowledge that human operators have built up around these tasks? While traditional cloud operations have used scripting tools that run externally to your cloud infrastructure, a more cloud-native approach is to have this database control logic running directly within your Kubernetes clusters. The question we'll explore in this chapter is: what is the Kubernetes-friendly way to represent this control logic?

**Extending the Kubernetes Control Plane**

The good news is that the designers of Kubernetes aren't surprised at all by this question. In fact, the Kubernetes control plane and API are designed to be extensible. Kelsey Hightower and others have referred to Kubernetes as “A platform for building platforms.”

Kubernetes provides multiple extension points, primarily related to its control plane. Figure 5-1 includes the Kubernetes core components along with indications of the extension points they support.
Extending Kubernetes Clients

Let’s review key elements of the control plane and their extension points, starting first with those that reside outside of a Kubernetes cluster.

Kubectl

The kubectl command line tool is the primary interface for many users for interacting with Kubernetes. You can extend kubectl with plugins that you download and make available on your system’s PATH, or use Krew, a package manager for kubectl plugins which maintains a list of plugins. Plugins perform tasks such as bulk actions across multiple resources or even multiple clusters, or assessing the state of a cluster and making security or cost recommendations. More particularly to our focus in this chapter, several plugins are available to manage operators and custom resources.

Extending Kubernetes Master Node Components

The following control plane components run on the Kubernetes master node:
API Server

The **API Server** is the primary interface for external and internal clients of a Kubernetes cluster. It exposes REST-ful interfaces via an HTTP API. The API Server performs a coordination role, routing requests from clients to other components to implement imperative and declarative instructions. The API Server supports two types of extensions: custom resources and API aggregation. Custom resources allow you to add new types of resources, and are managed through kubectl without further extension. API aggregation allows you to extend the Kubernetes API with additional REST endpoints, which the API Server will delegate to a separate API server provided as a plugin. Custom resources are the more commonly used extension mechanism and will be a major focus throughout the remainder of the book.

Scheduler

The **Scheduler** determines the assignment of pods to worker nodes, considering factors including the load on each worker node, as well as affinity rules, taints and tolerations (as discussed previously in Chapter 4). The Scheduler can be extended with plugins that override default behavior at multiple points in its decision making process. For example, a **scheduling plugin** could filter out nodes for a specific type of pod, or set the relative priority of nodes by assigning a score. **Binding plugins** can customize the logic that prepares a node for running a scheduled pod, such as mounting a network volume the pod needs. Data infrastructure such as Apache Spark that relies on running a lot of short-lived tasks may benefit from this ability to exercise more fine-grained control over scheduling decisions, as we'll discuss in Chapter 8.

etcd

**etcd** is a distributed key-value store used by the API Server to persist information about the cluster’s configuration and status. As resources are added, removed and updated, the API server updates the metadata in etcd accordingly, so that if the API Server crashes or needs to be restarted, it can easily recover its state. As a strongly consistent data store that supports high availability, etcd is frequently used by other data infrastructure that runs on Kubernetes, as we’ll see frequently throughout the book.

Controller Manager and Cloud Controller Manager

The **Controller Manager** and **Cloud Controller Manager** incorporate multiple control loops called controllers. These managers contain multiple logically separate controllers compiled into a single executable to simplify Kubernetes’ ability to manage itself. The Controller Manager includes controllers which manage built in resource types such as Pods and StatefulSets, and more. The Cloud Controller Manager includes controllers that differ between Kubernetes providers to enable the management of platform-specific resources such as load balancers or virtual machines.
Extending Kubernetes Worker Node Components

There are also elements of the Kubernetes control plane that run on the worker nodes:

**Kubelet**

The Kubelet manages the pods running on a node assigned by the Scheduler, including the containers that run within a pod. The Kubelet restarts containers when needed, provides access to container logs, and more.

**Compute, Network, and Storage Plugins**

The Kubelet can be extended with plugins that take advantage of unique compute, networking, and storage capabilities provided by the underlying environment on which it is running. Compute plugins include container runtimes, and device plugins which expose specialized hardware capabilities such as GPU or FPGA. Network plugins, including those that comply with the Container Network Interface (CNI), can provide features beyond Kubernetes built-in networking, such as bandwidth management or network policy management. We've previously discussed storage plugins in Chapter 2, including those that conform to the Container Storage Interface (CSI).

**Kube-proxy**

The kube-proxy maintains network routing for the pods running on a worker node so that they can communicate with other pods running inside your Kubernetes cluster, or clients and services running outside of the cluster. Kube-proxy is part of the implementation of Kubernetes Services, providing the mapping of virtual IPs to individual pods on a worker node.

**Container runtime**

The container runtime (not shown in Figure 5-1) is used to execute containers on the worker’s operating system. Container runtimes for Linux include containerd, CRI-O, and Docker.

**Custom controllers and operators**

These controllers are responsible for managing applications installed on a Kubernetes cluster using custom resources. Notice that while these controllers are extensions to the control plane, they run on worker nodes rather than the Kubernetes master node.

The Operator Pattern

With this context, we're ready to examine one of the most common patterns for extending Kubernetes: the operator pattern. The operator pattern combines custom resources with controllers that operate on those resources. Let's examine each of these
concepts in more detail in order to see how they apply to data infrastructure, and then you'll be ready to dig into an example operator for MySQL.

Controllers

The concept of a controller originates from the domain of electronics and electrical engineering, in which a controller is a device that operates in a continuous loop. On each iteration through the loop, the device receives an input signal, compares that with a set point value, and generates an output signal intended to produce a change in the environment that can be detected in future inputs. A simple example of this is a thermostat, which powers up your air conditioner or heater when the temperature in a space is too high or low.

A Kubernetes controller implements a similar control loop, consisting of the following steps:

1. Reading the current state of resources
2. Making changes to the state of resources
3. Updating the status of resources
4. Repeat

These steps are embodied both by Kubernetes built-in controllers that run in the Controller Manager and Cloud Controller Manager, as well as custom controllers that are provided to run applications on top of Kubernetes. Let's look at some examples of what these steps might entail for controllers that manage data infrastructure.

Reading the current state of resources

A controller tracks the state of one or more resource types, including built-in resources like Pods, PersistentVolumes, and Services, as well as custom resources we'll discuss below. Controllers are driven asynchronously, that is, by notification from the API Server. The API Server sends watch events to controllers to notify them of changes in state for resource types for which they have registered interest, such as the creation or deletion of a resource, or an event occurring on the resource.

For data infrastructure these changes could include a change in the number of requested replicas for a cluster, or a notification that a pod containing a database replica has died. Because there could be many such updates occurring in a large cluster, controllers frequently make use of caching.

Making changes to the state of resources

This is the core business logic of a controller - comparing the state of resources to their desired state and executing actions to change the state to the desired state. In the Kubernetes API, the current state is captured in .status fields of resources,
and the desired state is expressed in terms of the `.spec` field. The changes could include invocations of the Kubernetes API to modify other resources, administrative actions on the application being managed, or even interactions outside of the Kubernetes cluster.

For example, consider a controller managing a distributed database with multiple replicas. When the database controller receives a notification that the desired number of replicas has increased, the controller could scale an underlying Deployment or StatefulSet that it is using to manage replicas. Later, when receiving a notification that a pod has been created to host a new replica, the controller could initiate an action on one or more replicas in order to rebalance the workload across those replicas.

**Updating the status of resources**
In the final step of the control loop, the controller updates the `.status` fields of the resource using the API server, which in turn updates that state in etcd. You’ve viewed the status of resources like Pods and Persistent Volumes in previous chapters using the `kubectl get` and `kubectl describe` commands. For example, the status of a Pod includes its overall state (Pending, Running, Succeeded, Failed, etc.), the most recent time at which various conditions were noted (PodScheduled, ContainersReady, Initialized, Ready), as well as the state of each of its containers (Waiting, Running, Terminated). Custom resources can define their own status fields as well. For example a custom resource representing a cluster might have status values reflecting the overall availability of the cluster and its current topology.

**Events**
A controller can also produce *Events* via the Kubernetes API for consumption by human operators. These are distinct from the watcher events described above that the Kubernetes API uses to notify controllers of changes, which are not exposed to other clients. If you’ve ever misconfigured a Pod specification and observed a CrashLoop-BackOff status, you may have encountered Events. Using the `kubectl describe pod` command you can observe Events such as a container being started, failing, and a backoff period followed by the container restarting. Events expire from the API server in an hour, but common Kubernetes monitoring tools provide capabilities to track them. Controllers can also create events for custom resources.
Writing a custom controller

While you may not ever need to write your own controller, it’s helpful to be familiar with the concepts involved. The book *Programming Kubernetes* by Michael Hausenblas and Stefan Schimanski is a great resource for those who are interested in digging deeper.

The controller-runtime project provides a common set of libraries to help aid the process of writing controllers, including registering for notifications from the API Server, caching resource status, implementing reconciliation loops, and more. Controller-runtime libraries are implemented in the Go programming language, so it’s no surprise that most controllers are implemented in Go.

Go was first developed at Google beginning in 2007 and used there in many cloud-native applications including Borg, the predecessor to Kubernetes, and of course Kubernetes itself. Go is a strongly typed, compiled language (as opposed to interpreted languages like Java and JavaScript) with a high value on usability and developer productivity (in reaction to the higher learning curve of C/C++).

Custom Resources

As we’ve discussed above, controllers can operate on built-in Kubernetes resources as well as custom resources. We’ve briefly mentioned this concept above, but let’s take this opportunity to define what custom resources are and how they extend the Kubernetes API.

Fundamentally, a custom resource is a piece of configuration data that Kubernetes recognizes as part of its API. While a custom resource is similar to a ConfigMap, it has a structure similar to built-in resources: metadata, specification, and status. The specific attributes of a particular custom resource type are defined in a Custom Resource Definition, or CRD. A CRD is itself a Kubernetes resource that is used to describe a custom resource.

In this book, we’ve been discussing how Kubernetes enables you to move beyond managing virtual machines and containers to managing virtual data centers. Custom resources provide the flexibility that helps make this a practical reality. Instead of being limited to the resources that Kubernetes provides off the shelf, you can create additional abstractions to extend Kubernetes for your own purposes. This is a critical component in a fast-moving ecosystem.

Let’s see what you can learn about custom resources from the command line. Use the kubectl api-resources command to get a listing of all of the resources defined in your cluster:
As you look through the output, you'll see many resource types introduced in previous chapters, along with their short names: StorageClass (sc), PersistentVolumes (pv), Pods (po), StatefulSets (sts), and so on. The API versions provide some clues as to the origins of each resource type. For example, resources with version v1 are core Kubernetes resources. Other versions such as apps/v1, networking.k8s.io/v1, or storage.k8s.io/v1 indicate resources that are defined by various Kubernetes Special Interest Groups (SIGs).

Depending on the configuration of the Kubernetes cluster you are using, you may have some custom resources defined already. If any are present, they will appear in the output of the kubectl api-resources command. They'll stand out by their API version, which will typically include a path other than k8s.io.

Since a CRD is itself a Kubernetes resource, you can also use the command kubectl get crd to list custom resources installed in your Kubernetes cluster. For example, after installing the Vitess operator referenced in the section below, you would see several CRDs:

```
$ kubectl get crd
NAME                                   CREATED AT
ectdlockservers.planetscale.com        2021-11-21T22:06:04Z
vitessbackups.planetscale.com          2021-11-21T22:06:04Z
vitessbackupstorages.planetscale.com   2021-11-21T22:06:04Z
vitesscells.planetscale.com            2021-11-21T22:06:04Z
vitessclusters.planetscale.com         2021-11-21T22:06:04Z
vitesskeyspaces.planetscale.com        2021-11-21T22:06:04Z
vitessshards.planetscale.com           2021-11-21T22:06:04Z
```

We'll introduce the usage of these custom resources later on, but for now let's focus on the mechanics of a specific CRD to see how it extends Kubernetes. You use the kubectl describe crd or kubectl get crd commands to see the definition of a CRD. For example, to get yaml-formatted description for the vitesskeyspace custom resource, you could run:

```
$ kubectl get crd vitesskeyspaces.planetscale.com -o yaml
```

Looking at the original yaml configuration for this CRD, you'll see something like this:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
```
From this part of the definition, you can see the declaration of the custom resource’s name or Kind and shortName. The scope designation of Namespaced means that custom resources of this type are confined to a single namespace.

The longest part of the definition is the validation section, which we’ve omitted here because of its considerable size. Kubernetes supports the definition of attributes within custom resource types, as well as the ability to define legal values for these types using the OpenAPI v3 schema which is used to document RESTful APIs, which in turn uses JSON schema to describe rules used to validate JSON objects. Validation rules ensure that when you create or update custom resources, the definitions of the objects are valid and can be understood by the Kubernetes control plane. The validation rules are used to generate the documentation you use as you define instances of these custom resources in your application.

Once a CRD has been installed in your Kubernetes cluster, you can then create and interact with the resources using kubectl. For example, the command kubectl get vitesskeyspaces will return a list of Vitess keyspaces. You create an instance of a Vitess keyspace by providing a compliant yaml definition to the kubectl apply command.

**Operators**

Now that you’ve learned about custom controllers and custom resources, let’s tie these threads back together. An operator is a combination of custom resources and custom controllers that maintain the state of those resources and manage an application (or operand) in Kubernetes.
As we'll see in examples throughout the rest of the book, this simple definition can cover a pretty wide range of implementations. The recommended pattern is to provide a custom controller for each custom resource, but beyond that the details may vary. A simple operator might consist of a single resource and controller, while a more complex operator might have multiple resources and controllers. Those multiple controllers might run in the same process space or be broken out into separate pods.

**Controllers vs. operators**

While technically operators and controllers are distinct concepts in Kubernetes, the terms are frequently used interchangeably. It's common to refer to a deployed controller or collection of controllers as “the operator”, and you'll see this usage reflected both in this book and the community in general.

To unpack this pattern and see how the different elements of an operator and the Kubernetes control plane work together, let's consider the interactions of a notional operator, the DbCluster operator, as shown in Figure 5-2.

After an administrator installs the DbCluster Operator and db-cluster custom resource in the cluster, users can then create instances of the db-cluster resource in the cluster, users can then create instances of the db-cluster resource
using kubectl (1), which registers the resource with the API Server (2), which in turns stores the state in etcd (3) to ensure high availability (other interactions with etcd are omitted from this sequence for brevity).

The DbCluster controller (part of the operator) is notified of the new db-cluster resource (4) and creates additional Kubernetes resources using the API Server (5), which could include StatefulSets, Services, PersistentVolumes, PersistentVolume-Claims, and more, as we've seen in previous examples of deploying databases on Kubernetes.

Focusing on the StatefulSet path, the StatefulSet Controller running as part of the Kubernetes Controller Manager is notified of a new StatefulSet (6) and creates new Pod resources (7). The API Server asks the Scheduler to assign each Pod to a worker node (8) and communicates with the Kubelet on the chosen worker nodes (9) to start each of the required Pods (10).

As you see, creating a db-cluster resource sets off a chain of interactions as various controllers are notified of changes to Kubernetes resources and initiate changes to bring the state of the cluster in line with the desired state. The sequence of interactions appears complex from a user perspective, but the design demonstrates strong encapsulation: the responsibilities of each controller are well-bounded and independent of other controllers. This separation of concerns is what makes the Kubernetes control plane so extensible.

### Managing MySQL in Kubernetes using the Vitess Operator

Now that you understand how operators, custom controllers and custom resources work, it's time to get some hands-on experience with an operator for the database we've been using as our primary relational database example: MySQL.

MySQL examples in previous chapters were confined to simple deployments of a single primary replica and a couple of secondary replicas. While this could provide a sufficient amount of storage for many cloud applications, managing a larger cluster can quickly become quite complex, whether it runs on bare-metal servers or as a containerized application in Kubernetes.

### Vitess Overview

**Vitess** is an open source project started at YouTube in 2010. Before the company was acquired by Google, YouTube was running on MySQL, and as they scaled up they reached a point of daily outages. Vitess was created as a layer to abstract application access to databases by making multiple instances appear to be a single database, routing application requests between the instances using a sharding approach. Vitess has some complexity to it, so before we explore deploying Vitess on Kubernetes, let’s take
some time to explore its architecture. We’ll start with the high level concepts shown in Figure 5-3: cells, keyspaces, shards, leaders, and replicas.

At a high level, a Vitess cluster consists of multiple MySQL instances which may be spread across multiple data centers or **cells**. Each MySQL instance takes on a role as either a leader or replica, and may be dedicated to a specific slice of a database known as a shard. Let’s consider the implications of each of these concepts for reading and writing data in Vitess.

**Cell**

A typical production deployment of Vitess is spread across multiple failure domains in order to provide high availability. Vitess refers to each of these failure domains as a **cell**. The recommended topology is a cell per data center or cloud provider zone. While writes and replication involve communication across cell boundaries, Vitess reads are confined to the local cell to optimize performance.

**Keyspace**

A Vitess **keyspace** is a logical database consisting of one or more tables. Each keyspace in a cluster can be **sharded** or **unsharded**. An unsharded keyspace has a primary cell where a MySQL instance designated as the **leader** will reside, while other cells will contain **replicas**. In the unsharded keyspace shown on the left side of Figure 5-3, writes from client applications are routed to the leader and replicated to the replica nodes in the background. Reads can be served from the leader or replica nodes.
**Shard**

The real power of Vitess comes from its ability to scale by spreading the contents of a keyspace across multiple replicated MySQL databases known as *shards*, while providing the abstraction of a single database to client applications. The client on the right side of Figure 5-3 is not aware of how data is sharded. On writes, Vitess determines what shards are involved, and routes the data to the appropriate leader instances. On reads, Vitess gathers data from leader or replica nodes in the local cell.

The sharding rules for a keyspace are specified in a *VSchema*, an object which contains the sharding key (known in Vitess as the *KeyspaceID*) used for each table. To provide maximum flexibility over how data is sharded, Vitess allows you to specify which columns in a table are used to calculate the KeyspaceID, as well as the algorithm (or *VIndex*) used to make the calculation. Tables can also have secondary VIndexes to support more efficient queries across multiple KeyspaceIDs.

In order to understand how Vitess manages shards and how it routes queries to the various MySQL instances, you’ll want to get to know the components of a Vitess cluster shown in Figure 5-4, including VTGate, VTTablet, and the Topology Service.

*Figure 5-4. Vitess architecture including VTGate, VTTablets, Topology Service*

Let’s walk through each of these components to learn what they do and how they interact.
**VTGate**

A Vitess gateway or VTGate is a proxy server that provides the SQL binary endpoint used by client applications, making the Vitess cluster appear as a single database. Vitess clients generally connect to a VTGate running in the same cell (data center). The VTGate parses each incoming read or write query and uses its knowledge of the VSchema and cluster topology to create a query execution plan. The VTGate executes queries for each shard, assembles the result set, and returns it to the client. The VTGate can detect and limit queries that will impact memory or CPU utilization, providing high reliability and helping to ensure consistent performance. Although VTGate instances do cache cluster metadata, they are stateless, so you can increase the reliability and scalability of your cluster by running multiple VTGate instances per cell.

**VTTablet**

A Vitess tablet or VTTablet is an agent that runs on the same compute instance as a single MySQL database, managing access to it and monitoring its health. Each VTTablet takes on a specific role in the cluster, such as the leader of a shard, or one of its replicas. There are two types of replica, those that can be promoted to replace a leader, and those that cannot. The latter are typically used to provide additional capacity for read-intensive use cases such as analytics. The VTTablet exposes a gRPC interface which the VTGate uses to send queries and control commands, which the VTTablet then turns into SQL commands on the MySQL instance. VTTablets maintain a pool of long lived connections to the MySQL node, leading to improved throughput, reduced latency and reduced memory pressure.

**Topology Service**

Vitess requires a strongly consistent data store to maintain a small amount of metadata describing the cluster topology, including the definition of keyspaces and their VSchema, what VTTables exist for each shard, and which VTTablet is the leader. Vitess uses a pluggable interface called the topology service, with three implementations provided by the project: etcd (the default), ZooKeeper, and Consul. VTGates and VTTablets interface with the Topology Service in the background in order to maintain awareness of the topology, and do not interact with the Topology Service on the query path to avoid performance impact. For multi-cell deployments, Vitess incorporates both cell-local Topology Services, and a global Topology Service with instances in multiple cells that maintains knowledge of the entire cluster. This design provides high availability of topology information across the cluster.

**Vtctl and Vtctlclient**

The Vitess control daemon vtctl and its client vtctlclient provide the control plane used to configure and manage Vitess clusters. (There is also a command-line version called vtctl that combines the client and server as a single executable,
but it is not frequently used in cloud deployments for security reasons.) *Vtctl* is deployed one or more of the cells in the cluster, while *vtctlclient* is deployed on the client machine of the user administering the cluster. *Vtctl* uses a declarative approach similar to Kubernetes to perform its work: it updates the cluster metadata in the Topology Service, and the VTGates and VTTablets pick up changes and respond accordingly.

Now that you understand the Vitess architecture and basic concepts, let’s discuss how they are mapped into a Kubernetes environment. This is an important consideration for any application, but especially for a complex piece of data infrastructure like Vitess.

**PlanetScale Vitess Operator**

Over time, Vitess has evolved in a couple of key aspects: first, it can now run additional MySQL-compatible database engines such as Percona and MariaDB. Second, and more important for our investigations, PlanetScale has packaged Vitess as a containerized application that can be deployed to Kubernetes.

---

**Evolving options for running Vitess in Kubernetes**

The state of the art for running Vitess in Kubernetes has evolved over time. While Vitess once included a Helm chart, this was deprecated in the 7.0 release in mid 2020. The Vitess project also hosted an operator which was deprecated around the same time. Both of these options were retired in favor of the PlanetScale operator we examine in this section.

Let’s see how easy it is to deploy a multi-node MySQL cluster using the PlanetScale Vitess Operator. Since the Vitess project has adopted the PlanetScale operator as it’s officially supported operator, you can reference the getting started guide in the Vitess project documentation. We’ll walk through a portion of this guide here in order to get an understanding of the operator’s contents and how it works.

**Examples require Kubernetes clusters with more resources**

The examples in previous chapters have not required a large amount of compute resources, and we encouraged you to run them on local distributions such as Kind or K3s. Starting with this chapter, the examples become more complex and may require more resources than you have available on your desktop or laptop. For these cases we will provide references to documentation or scripts for creating Kubernetes clusters with sufficient resources.
Installing the Vitess Operator

You can find the source code used in this section at Vitess Operator Example. The files are copied for convenience from their original source in the Vitess GitHub repo. First, install the operator using the provided configuration file:

```bash
kubectl apply -f operator.yaml
customresourcedefinition.apiextensions.k8s.io/
etcdlockservers.planetscale.com created
...
```

As you'll see in the output of the `kubectl apply` command, this configuration creates several CRDs, as well as a deployment managing a single instance of the operator. Figure 5-5 shows many of the elements you've just installed, in order to highlight a few interesting details will not be obvious at first glance:

- The operator contains a controller corresponding to each CRD. If you're interested in seeing what this looks like in the operator source code in Go, compare the controller implementations with the custom resource specifications that are used to generate the CRD configurations introduced in Custom Resources. See more about building operators below.

- The figure depicts a hierarchy of CRDs representing their relationships and intended usage, as described in the operator's API reference. To use the Vitess operator, you define a VitessCluster resource which contains the definitions of VitessCells and VitessKeyspaces. VitessKeyspaces, in turn, contain definitions of VitessShards. While you can view the status of each VitessCell, VitessKeyspace, and VitessShard independently, you must update them in the context of the parent VitessCluster resource.

- Currently the Vitess operator only supports etcd as the Topology Service implementation. The EtcdLockserver CRD is used to configure these etcd clusters.
**Figure 5-5. Vitess Operator and Custom Resource Definitions**

**Roles and RoleBindings**

As shown toward the bottom of Figure 5-4, installing the operator caused the creation of a ServiceAccount, along with two new resources we have not discussed previously: a Role and a RoleBinding. These additional resources allow the ServiceAccount to access specific resources on the Kubernetes API. First, let’s examine the configuration of the vitess-operator Role:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: vitess-operator
rules:
  - apiGroups:
      - ""
    resources:
    - pods
    - services
```
This first portion of the Role definition identifies resources that are part of the core Kubernetes distribution, which may be designated by passing the empty string as the apiGroup instead of k8s.io. The verbs correspond to operations the Kubernetes API provides on resources, including get, list, watch, create, update, patch, and delete. This Role is given access to all operations using the wildcard ". The Role is also given access to other resources, including partial access to Deployments and StatefulSets, and full access to resources in the apiGroup planetscale.com.

The RoleBinding associates the ServiceAccount with the Role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: vitess-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: vitess-operator
subjects:
- kind: ServiceAccount
  name: vitess-operator
```

**Least privilege for operators**

As a creator or consumer of operators, exercise care in which permissions are granted to operators, and be conscious of the implications for what an operator is allowed to do.

**PriorityClasses**

There is another detail not depicted in Figure 5-4: installing the operator created two PriorityClass resources. PriorityClasses provide input to the Kubernetes scheduler to indicate the relative priority of Pods. The priority is an integer value, where higher values indicate higher priority. Whenever a Pod resource is created and is ready to be assigned to a worker node, the Scheduler takes the Pod's priority into account as part of its decisions. When multiple Pods are awaiting scheduling, higher priority Pods are assigned before lower priority Pods. When a cluster’s nodes are running low on compute resources, lower priority Pods may be stopped or evicted in order to make room for higher priority Pods, a process known as preemption.
A PriorityClass is a convenient way to set a priority value referenced by multiple Pods or other workload resources such as Deployments and StatefulSets. The Vitess operator creates two PriorityClasses: vitess-operator-control-plane defines a higher priority used for the operator and vtctld Deployments, while the vitess class is used for the data plane components such as the VTGate and VTTablet Deployments.

**Kubernetes scheduling complexity**

Kubernetes provides multiple constraints that influence Pod scheduling, including prioritization and preemption, affinity and anti-affinity, and scheduler extensions. The interaction of these constraints may not be predictable, especially in large clusters shared across multiple teams. As resources in a cluster become scarce, pods can be preempted or fail to be scheduled in ways you don’t expect. It’s a best practice to maintain awareness of the various scheduling needs and constraints across the workloads in your cluster to avoid surprises.

**Creating a Vitess Cluster**

Now let’s create a VitessCluster and put the operator to work. The code sample contains a configuration file defining a very simple cluster named example, with a VitessCell zone1, keyspace commerce, and single shard, which the operator gives the name x-x:

```bash
kubectl apply -f 101_initial_cluster.yaml
vitesscluster.planetscale.com/example created
secret/example-cluster-config created
```

The output of the command indicates a couple of items that are created directly, but there is more going on behind the scenes, as the operator detects the creation of the VitessCluster and begins provisioning other resources, as summarized in Figure 5-6.
By comparing the configuration script with Figure 5-6, you can make several observations about this simple VitessCluster. First, the top level configuration allows you to...
specify the name of the cluster and the container images that will be used for the various components:

```yaml
apiVersion: planetscale.com/v2
kind: VitessCluster
metadata:
  name: example
spec:
  images:
    vtctld: vitess/lite:v12.0.0
  ...
```

Next, the VitessCluster configuration provides a definition of the VitessCell zone1. The values provided for gateway specify a single VTGate instance to be allocated for this cell, with specific compute resource limits.

```yaml
cells:
  - name: zone1
gateway:
    authentication:
      static:
        secret:
          name: example-cluster-config
          key: users.json
          replicas: 1
          resources:
            ...
```

The Vitess Operator uses this information to create a VTGate Deployment prefixed with example-zone1-vtgate containing a single replica, and a Service that provides access. The access credentials for the VTGate instance are provided in the example-cluster-config Secret. This Secret is used to secure other configuration values, as you'll see below.

The next section of the VitessCluster configuration specifies the creation of a single vtctld instance (aka “dashboard”) with permission to control zone1. The Vitess Operator uses this information to create a Deployment to manage the dashboard using the specified resource limits, and a Service to provide access to the VTGate.

```yaml
vitessDashboard:
  cells:
    - zone1
  extraFlags:
    security_policy: read-only
    replicas: 1
    resources:
      ...
```
The VitessCluster also defines the commerce keyspace, which contains a single shard (essentially, an unsharded keyspace). This single shard has a pool of two VTTablets in the cell zone1, each of which will be allocated 10GB of storage.

```
keyspaces:
  - name: commerce
turndownPolicy: Immediate
  partitionings:
    - equal:
      parts: 1
      shardTemplate:
        databaseInitScriptSecret:
          name: example-cluster-config
          key: init_db.sql
        replication:
          enforceSemiSync: false
      tabletPools:
        - cell: zone1
type: replica
replicas: 2
vttablet:
  ...
mysqld:
  ...
dataVolumeClaimTemplate:
  accessModes: ["ReadWriteOnce"]
resources:
  requests:
    storage: 10Gi
```

As shown in Figure 5-6, the Vitess operator manages a Pod for each VTTablet and creates a Service to manage access across the tablets. The operator does not use a StatefulSet because the VTTablets have distinct roles with one as the primary and the other as a replica. Each VTTablet Pod contains multiple containers, including the vttablet sidecar which configures and controls the mysql container. The vttablet sidecar initializes the mysql instance using a script contained in the example-cluster-config Secret.

While this configuration doesn’t specifically include details about etcd, the Vitess Operator uses its default settings to create a 3-node etcd cluster to serve as the Topology Service for the VitessCluster. Because of the shortcomings of the StatefulSets, the operator manages each pod and PersistentVolumeClaim individually. This points to the possibility for future improvements as Kubernetes and the operator mature; perhaps the Kubernetes API server can one day serve the role of the Topology Service in the Vitess architecture.
Visualizing larger Kubernetes applications

While it’s a good exercise to use the `kubectl get` and `kubectl describe` commands to explore all of the resources that were created when you installed the operator and created a cluster, you may find it easier to use a tool such as Lens, which offers a friendly graphical interface that allows you to click through the resources more quickly.

At this point, you have a VitessCluster with all of its infrastructure provisioned in Kubernetes. The next steps are to create the database schema and configure your applications to access the cluster using the VTGate Service. You can follow the steps in the blog post *Vitess Operator for Kubernetes*, which also describes other use cases for managing a Vitess installation on Kubernetes, including schema migration, backup, and restore.

The backup/restore capabilities leverage VitessBackupStorage and VitessBackup CRDs which you may have noticed during installation. VitessBackupStorage resources represent locations where backups can be stored. After you configure the backup section of a VitessCluster and point to a backup location, the operator creates VitessBackup resources as a record of each backup it performs.

Resharding is another interesting use case, which you might need to perform when a cluster becomes unbalanced and one or more shards run out of capacity more quickly than others. You'll need to modify the schema using `vtctlclient`, and then update the VitessCluster resource with additional VitessShards so that the operator provisions the required infrastructure. This highlights the division of responsibility: the Vitess operator manages Kubernetes resources, while the Vitess control daemon (vtctld) provides more application-specific behavior.

What We Learned Building the Vitess Operator

With Deepthi Sigireddi, Software Engineer, PlanetScale

Vitess can be described simply as a scaling infrastructure for MySQL. Vitess started at YouTube in 2010 when the team was struggling with daily outages with MySQL. A few people got together and decided that rather than fighting fires every day, they would solve their problem from the ground up. Initially, Vitess was very customized to YouTube's environment. Applications were segmented into groups to run against one database or another, with a layer in between to route queries to the right backing database. Over time, the internal architecture became more complex but simpler from the application's point of view. Vitess started with custom sharding which required the application to know which database to query against. Now the application doesn't need to know whether there are 10 MySQL databases or 100, or 1000. As far as the application layer is concerned, it looks like a single database.
The move toward Kubernetes started when YouTube was acquired by Google. The mandate to use Google infrastructure included adapting Vitess to run on Borg, the precursor to Kubernetes. With Borg, the applications had to be tolerant to being restarted anytime, but that wasn't something supported by MySQL. If Borg doesn't want a Vitess component running on a machine, that component is rescheduled to run on some other machine. The team built tolerations for this type of automation as features in Vitess, and that's how Vitess became cloud native. All this sounds familiar to us now because that's how Kubernetes operates. When we decided to make Vitess run on Kubernetes, the team at YouTube was able to do the work without a lot of changes.

Before Vitess was donated to CNCF in January of 2018, there was already a project called Metacontroller, which predated the Operator SDK. This was used to get Vitess working on Kubernetes, independent of Google's infrastructure. It seemed intuitive that you would want to run Vitess using an operator, since there was already a community-contributed Helm chart and we saw the movement in the community toward operators.

There was an early community effort by an individual Vitess contributor to write a Kubernetes operator, but it was a pretty complex undertaking to take on alone and so it didn't go far. Other Vitess users such as HubSpot have built their own custom operators which are private since they are quite specific to their own deployments. PlanetScale started building a Kubernetes operator for Vitess to run as a cloud service and once it matured, we released 90% of that code as an open source Vitess operator.

In order to write an operator for an application, you need to understand both Kubernetes and the application really well. Kubernetes moves fast, with new releases every 4 months. Many features that were in alpha when we first started building our operator and are now a part of Kubernetes. Meanwhile, MySQL continues to evolve and add new query constructs. Recently in MySQL 8.0, there was a lot of new syntax added and we have to keep up with those changes.

To run a service in Kubernetes, you have to know the important lifecycle events and how those disrupt availability. Vitess achieves automatic failure detection and failover through a mixture of approaches. If your primary MySQL node is running with a persistent volume that goes down, Kubernetes will restart it with a downtime of 20 or 30 seconds. This is pretty fast, but maybe more than what some applications can tolerate. We are building into Vitess the ability to detect and failover much faster than a Kubernetes hot restart. Vitess will detect that the primary has gone down and will fail over to a replica that has kept up with the primary within 5 or 10 seconds. This capability has been available previously with third-party solutions built on top of Vitess, but by integrating this directly into Vitess, the reliability will be greatly improved.

Another area of focus is speeding up startup and shutdown. Network constraints like TCP/IP timeouts limit how quickly you can detect failure, but MySQL startup and shutdown are not yet at the point of hitting that lower bound. The first operator we built at PlanetScale took 10 or 20 minutes to bring up a cluster. This was partly due to
inefficiencies in the Operator SDK, and partly because we had written a single controller with a gigantic reconcile loop. We rewrote the operator to use a newer version of the Operator SDK and to have a separate controller for each resource. This made our startup and shutdown times 20 times faster, which was a hard requirement for providing a cloud service. Clients expect those operations to take 10 or 15 seconds, not two or three minutes. We have to be more clever to achieve that.

We also need more primitives from Kubernetes in order to continue to mature database operators. While Kubernetes provides Deployments, ReplicaSets and StatefulSets, it doesn’t yet support the concept of primary and replicas as MySQL needs. Imagine if you could configure Kubernetes to designate a primary, and specify an action to perform if the primary is restarted. A lot of the error handling code included in Vitess would actually not be required. While Kubernetes has a leader election module, there’s no clear way to leverage this for an operator that already has the concept of primaries and replicas. This leads to more duplicated code.

Application developers are looking for more control over where their data is stored, and easy ways to ingest or load data. Every organization that provides a database solution on Kubernetes should consider providing it as a service to make it easier for developers to consume. Developers should not have to worry about data locality. Today if a developer is running an application in AWS and a particular data service is not available there, they have to consider using another cloud or building the capability themselves. It should be really easy to create and populate a data source for an application no matter where you run it.

Infrastructure provisioning is getting easier and easier, and long may that trend continue. Even so, there is a lot more work to do. Those of us who get paid to work on open source are very fortunate because there are many developers who aren’t compensated for their open-source contributions. Hopefully, employers will continue to see the benefits of working on open source software and we can continue to grow as a community.

A Growing Ecosystem of Operators

The operator pattern has become quite popular in the Kubernetes community, resulting in the development of the Operator Framework, an ecosystem for creating and distributing operators. In this section we’ll examine the Operator Framework and related open source projects.

Choosing Operators

While we’ve focused in this chapter on Vitess as an example database operator, operators are clearly relevant to all of the elements of your data stack. In all aspects of cloud native data, we see a growing number of maturing, open-source operators to use in
your deployments, and we’ll be looking at additional operators as we examine how to run different types of data infrastructure on Kubernetes in upcoming chapters.

You should consider multiple aspects in choosing an operator - what are its features? How much does it automate? How well supported is it? Is it proprietary or open source? The Operator Framework provides a great resource you should consider your first stop when looking for an operator: Operator Hub. Operator Hub is a well-organized list of various operators that cover every aspect of cloud native software. It does rely on maintainers to submit their operators for listing, which means that many existing operators may not be listed.

The Operator Framework also contains the Operator Lifecycle Manager, an operator for installing and managing other operators in your cluster. You can curate your own custom catalog of operators that are permitted in your environment, or use catalogs provided by others. For example, Operator Hub can itself be treated as a catalog.

Part of the curation the Operator Hub provides is rating the capability of each operator according to the Operator Capability Model. The levels in this capability model are summarized in Table 5-1, using text from the Operator Framework site with additional commentary we’ve added to highlight considerations for database operators. The examples are not prescriptive but indicate the type of capabilities expected at each level.

Table 5-1. Operator Capability Levels applied to databases

<table>
<thead>
<tr>
<th>Capability Level</th>
<th>Characteristics</th>
<th>Database Operator Examples</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 - Basic Install</td>
<td>Installation and configuration of Kubernetes and workloads</td>
<td>The operator uses custom resources to provide a central point of configuration for a database cluster.</td>
<td>Helm, Ansible, Go</td>
</tr>
<tr>
<td>Level 2 - Seamless Upgrades</td>
<td>Upgrade of the managed workload and operator</td>
<td>The operator can update an existing database to a newer version without data loss (or hopefully, downtime). The operator can be replaced with a newer version of itself.</td>
<td></td>
</tr>
</tbody>
</table>

The examples are not prescriptive but indicate the type of capabilities expected at each level.
These levels are useful both for evaluating databases and for providing targets for operator developers to aim for. They also provide an opinionated view on what Helm-based operators can accomplish, limiting them to Level 2. For full lifecycle management and automation, more direct involvement with the Kubernetes control plane is needed. For a Level 5 operator, the goal is a complete hands-off deployment.

Let’s take a quick look at a few of the available operators for popular open-source databases:

**Cass-operator for Apache Cassandra**

In 2021, several companies in the Cassandra community who had developed their own operators came together in support of cass-operator. Cass-operator was inspired by the best features of the community operators and DataStax experience running Astra, a Cassandra-based DBaaS. The operator has been donated to the K8ssandra project, where it is part of a larger ecosystem for deploying Cassandra on Kubernetes. We’ll take a deeper look at K8ssandra and cass-operator in Chapter 7.

**PostgreSQL Operators**

There are several operators available for PostgreSQL, which is not surprising given that it is the second most popular open source database after MySQL. Two of the most popular operators are the Zalando Postgres Operator, and PGO (which also stands for Postgres Operator) from CrunchyData. Read the blog...
Comparing Kubernetes operators for PostgreSQL for a helpful comparison of these and other operators.

**MongoDB Kubernetes Operator**

MongoDB is the most popular document database, beloved by developers for its ease of use. The MongoDB Community Operator provides basic support for creating and managing MongoDB Replica Sets, scaling up and down, and upgrades. This operator is available on GitHub but not yet listed on Operator Hub, possibly because MongoDB also offers a separate operator for its enterprise version.

**Redis Operator**

Redis is an in-memory key-value store that has a broad set of use cases. Application developers typically use Redis as an adjunct to other data infrastructure when ultra-low latency is required. It excels at things such as caching, counting and shared data structures. The Redis Operator covers the basic install and upgrade but also manages harder operations such as cluster failover and recovery.

As you can see, operators are available for many popular open source databases, although it’s unfortunate that some vendors have tended to think of Kubernetes operators primarily as a feature differentiator for paid enterprise versions.

**Building Operators**

While there is broad consensus in the Kubernetes community that you should use operators for distributed data infrastructure whenever possible, there are a variety of opinions about who exactly should be building operators. If you don’t happen to work for a data infrastructure vendor, this can be a challenging question. The article “When to Use, and When to Avoid, the Operator Pattern” provides some excellent questions to consider, which we’ll summarize here:

- What is the scale of the deployment? If you’re only deploying a single instance of the database application, building and maintaining an operator might not be cost effective.
- Do you have the expertise in the database? The best operators tend to be built by companies that are running databases at scale in production, including vendors that are providing DBaaS solutions.
- Do you need higher levels of application awareness and automation, or would deployment with a Helm chart and standard Kubernetes resources be sufficient?
- Are you trying to make the operator manage resources that are external to Kubernetes? Consider a solution that runs closer to the resources being managed with an API you can access from your Kubernetes application.
• Have you considered security implications? Since operators are extensions of the Kubernetes control plane, you’ll want to carefully manage what resources your operator can access.

If you decide to write an operator, there are several great tools and resources available:

**OperatorSDK**

The Operator Framework includes Operator SDK, a software development kit containing tools to build, test, and package operators. Operator SDK uses templates to auto-generate new operator projects and provides APIs and abstractions to simplify common aspects of building operators, especially interactions with the Kubernetes API. The SDK supports the creation of operators using Go, Ansible or Helm.

**Kubebuilder**

Kubebuilder is a toolkit for building operators managed by the Kubernetes API Machinery SIG. Similar to Operator SDK, Kubebuilder provides tools for project generation, testing, and publishing controllers and operators. Both Kubebuilder and OperatorSDK are built on The Kubernetes controller-runtime, a set of Go libraries for building controllers. The blog post Kubebuilder vs Operator SDK provides a concise summary of the differences between these toolkits.

**Kudo**

The Kubernetes Universal Declarative Operator, or Kudo for short, takes a declarative approach. Kudo is an operator that allows you to create operators declaratively using yaml files. This is an attractive approach for some developers as it eliminates the need to write Go. The blog post How to deploy your first app with Kudo operator on K8S provides a helpful introduction to using Kudo and discusses some of the pros and cons of the declarative approach.

Finally, the O’Reilly books Kubernetes Operators and Programming Kubernetes are great resources for understanding the operator ecosystem and getting into the details of writing operators and controllers in Go.

---

**Can one operator rule them all?**

With Umair Mufti, Product Manager, PortWorx

The number of Kubernetes operators for databases has been continually growing. Database developers want their databases to run on Kubernetes, so to their credit, they are stepping up and developing operators to make it easy for others. This initiative is great, but the problem is that everyone is building operators their own way and solving similar problems with different implementations. As a result, there is no uniformity between operators for stateful workloads.
The question we have to reckon with is how specialized we expect end-users to be. Because of the popularity of cloud-native, microservice architectures, application developers now expect polyglot persistence: to run a relational database right next to a graph database or a key-value store. This gives cluster administrators the requirement to provide different types of databases while maintaining the operational simplicity of a single platform.

No Kubernetes admin wants to be maintaining 10 or 15 different operators on their cluster. The point of Kubernetes is the ease of operations when deploying applications, monitoring them on day two, and making lifecycle management simpler. As soon as you have the maintenance overhead of managing operator lifecycles, you are already broken. Multiply that 10 or 15 times over, and you are completely at odds with what Kubernetes is trying to provide. The only way out of this situation is to reduce the number of operators. Could there be a single operator for all our databases or stateful workloads? Let’s explore.

The operator pattern is simply a design pattern for running stateful workloads in Kubernetes, just as the Model View Controller framework is a pattern for user-facing applications. Various web frameworks such as Angular, Vue and React use the MVC pattern, but they all implement the pattern in different ways, and your code will vary based on the implementation you use. This is a familiar experience for developers using operators today. Each operator solves the problem of running a stateful workload in Kubernetes in a unique way, and it requires specialization to become proficient with each operator. The irony is that if you’re running Cassandra, Redis, or Postgres, a lot of the problems being addressed are very much the same: cluster membership, failure detection, backup and restore, and more.

Could we actually build “one operator to rule them all”? Well, not necessarily. Perhaps what we need is not literally one single operator, but a collection of higher-level interfaces that operators should adhere to, so they work with multiple data service types. This would enable administrators to choose an operator based on factors other than the vendor or project that created it. What if you could use an operator that would manage your Cassandra, ElasticSearch, and Kafka clusters? This is what we need to reduce the burden on operations teams and fully realize the benefits of managing stateful workloads on Kubernetes.

We need to build another layer of abstraction on top of the operator pattern. As a community, we can develop a common set of custom resources, and each controller can manage them in their own way. For example, we might define a “topology-aware StatefulSet” as a new CRD, or a cluster membership CRD that describes how a node joins a cluster. Instead of ElasticSearch developers and Cassandra developers creating separate definitions of a server group or topology, we could all agree that a distributed database has a concept of topology, agree on a CRD, and controllers can implement the specified behavior as needed.

The ideal end-state is a world with multiple implementations that adhere to a common standard. Kubernetes itself has a specification and each Kubernetes distribution...
has to provide certain APIs to be considered a valid distribution. Users can choose which operators to use in the same way, knowing that they can expect a baseline standard while applying other criteria.

Kubernetes still shows signs that it was born of a stateless world but I’m excited about the future of stateful workloads on Kubernetes. We are very much in that “Crossing the Chasm” moment and still just hitting the inflection point with stateful workloads. With more advanced operators, we’ll no longer be working in silos, solving the same problems over and over again. Then we can use our collective talents and skills to solve bigger and higher level problems.

**Summary**

In this chapter, you’ve learned about several ways of extending the Kubernetes control plane, especially operators and custom resources. The operator pattern provides the critical breakthrough that enables us to simplify database operations in Kubernetes through automation. While you should definitely be using operators to run distributed databases in Kubernetes, think carefully before starting to write your own operator. If building an operator is the right course for you, there are plenty of resources and frameworks to help you along the way. There are certainly ways in which Kubernetes itself could improve to make writing operators easier, as you’ve learned from the experts we spoke to in this chapter.

While we’ve spent the past couple of chapters focusing primarily on running databases on Kubernetes, let’s see how you can apply what you’ve learned so far to a different type of data infrastructure: streaming and messaging.
A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that this book’s code examples are available at https://github.com/data-on-k8s-book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at jleonard@oreilly.com.

When asking application developers about data infrastructure, persistence is what immediately comes to mind—storing the state of running applications. While our focus up to this point has been on databases and storage, it’s now time to branch out and consider the other aspects of the cloud native data stack, and in this chapter, we’ll spend some time looking at streaming technologies. If you manage data pipelines, you may even start here.

Introduction to Streaming

In Chapter 1, we defined streaming as the function of moving data from one point to another, optionally with processing in transit. The history of streaming is almost as long as persistence. As data was pooling in various isolated stores, it became evident that moving data reliably was just as important as the reliability of storing data. In those days, it was called messaging, and data was transferred in a slower but deliberate way that resembled something closer to postal mail. Messaging infrastructure put
data in a place where it could be read asynchronously, in order, with delivery guarantees.

Modern application requirements have evolved what was known as messaging into today’s definition of streaming. Typically this means managing large volumes of data that require more immediate processing, which we call “near real-time.” Ordering and delivery guarantees become a critically important feature in the distributed applications deployed in Kubernetes and in many cases, are a key enabler of the scale required. How can adding more infrastructure complexity help scale? By providing an orderly way to manage the flow from where data is created to where it can be used and stored. There is a lot of software and terminology around streaming that can confuse first-time users. As with any complex topic, decomposing the parts can be helpful as we build understanding. There are three areas to evaluate when choosing a streaming system for your use case:

- Types of delivery
- Delivery guarantees
- Feature scope for streaming

**Types of delivery**

To use streaming in your application, you will need to understand the delivery methods available to you from the long choice list of streaming systems. You will need to understand your application requirements in order to plan how data flows from producer to consumer efficiently. For example, “Does my consumer need exclusive access?” The answer will drive which system fits the requirements. Figure 6-1 shows the choices most common in streaming systems: Point to point, and publish / subscribe.
**Point to point**

Data created by the producer is passed through the broker and then to a single consumer in a one-to-one relationship. This is primarily used as a way to decouple direct connections from producer to consumer. An excellent feature for resilience as consumers can be removed and added with no data loss while the broker maintains order and last message read, addressable by the consumer using an offset.

**Publish / Subscribe (pub/sub)**

The broker serves as a distribution hub for a single producer and one or more consumers in a one-to-many relationship. Consumers subscribe to a topic and receive notifications for any new messages created by the producer—a critical component for reactive or event-driven architectures.

**Delivery Guarantees**

In conjunction with the delivery types, the broker maintains delivery guarantees from producer to consumer per message type in an agreement called a *contract*. The common delivery types are shown in Figure 6-2: at-most-once, at least-once, and exactly once.

![Figure 6-2. Delivery Guarantees](image)

**At-most-once**

The lowest guarantee allows data created by the producer to be skipped by the consumer. For example, this might be used when a consumer only needs to react to the most current information. If a consumer is taken offline for any period, the correct action once back online is to pick up processing on the latest data and
ignore anything previously delivered. The critical downside to understand is that data loss is possible by design.

**At-least-once**

This guarantee is the opposite side of at-most-once. Data created by the producer is guaranteed to be picked up by a consumer. The added aspect allows for re-delivery any number of times after the first. For example, this might be used with a unique key such as a date stamp or ID number that is considered idempotent on the consumer side that multiple processing won’t impact. The consumer will always see data delivered by the producer but could see it numerous times. Your application will need to account for this possibility.

**Exactly-once**

The strictest of the three guarantees, this means that data created by a producer will only be delivered one time to a producer. Example: Exact transactions such as money movement where subtractions or additions must be delivered and processed one time to avoid problems. This guarantee puts a greater burden on the broker to maintain, so you will need to adjust the resources allocated to the broker as well as your expected throughput.

You should exercise care in the selection of delivery guarantees for each type of message. Delivery guarantees are ones to carefully evaluate as they can have unexpected downstream effects on the consumer if not wholly understood. Questions like “Can my application handle duplicate messages?” need a good answer. “Maybe” is not good enough.

**Feature scope**

There are many streaming technologies available, some of which have been around for quite a few years. On the surface, the technologies may appear similar, but they each solve a different problem due to new requirements. The majority are open source projects, so each found a community of like-minded individuals who join in and advance the project. Similar to how many different persistent data stores fit under the large umbrella of “database”, the combination and difference of features can vary significantly under the heading of data streaming.

Feature scope is likely the most important selection criteria when evaluating which streaming technology to use, but you should also challenge yourself to add the suitability for Kubernetes as a criteria, and consider whether more complex features are worth the added resource cost? Fortunately, the cost for getting your decision wrong the first time is relatively low. Streaming data systems tend to be some of the easiest to migrate due to their ephemeral nature. The deeper into your feature stack the streaming technology goes, the harder it is to move. The scope of streaming features can be broken into the two large buckets shown in Figure 6-3.
Message broker
The simplest form of streaming technology that facilitates the moving of data from one point to another with one or more of the delivery methods and guarantees listed above. It’s easy to discount this feature's simplistic appearance, but it's the backbone of modern cloud native applications. It's like saying FedEx is just a package delivery company, but imagine what would happen to the world economy if it stopped for even one day? Example OSS message brokers include: Apache Kafka, Apache Pulsar, RabbitMQ, and Apache ActiveMQ.

Stream analytics
In some cases, the best or only time to analyze data is while it is moving. Waiting for data to persist and then begin the analysis could be far too late, and the insight’s value is almost useless. Consider fraud detection. The only opportunity to stop the fraudulent activity is when it's happening, waiting for a report to run the next day just doesn’t work. Example OSS stream analytics systems include: Apache Spark, Apache Flink, Apache Storm, Apache Kafka Streams, and Apache Pulsar.

The Role of Streaming in Kubernetes
Now that we have covered the basic terminology, how does streaming fit into a cloud native application running on Kubernetes? Database applications follow the pattern of create, read, update and delete (CRUD). For a developer, the database provides a single location for data. The addition of streaming assumes some sort of motion in the data from one place to another. Data may be short-lived if used to create new data. Some data may be transformed in transit, and some may eventually be persisted. Streaming assumes a distributed architecture, and the way to scale a streaming system is to manage its resource allocation of compute, network, and storage. This is landing right into the sweet spot of cloud native architecture. In the case of stream-
driven applications in Kubernetes, you’re managing the reliable flow of data in an environment that can change over time. Allocate what you need, when you need it.

**Streaming and Data Engineering**

Data Engineering is a relatively new discipline so we want to be sure to define it as this is a fast-growing field. This is especially applicable to the practice of data streaming. Data Engineers are concerned with the efficient movement of data in complex environments. The two T’s are important in this case. Transport and Transformation. The role of the Data Scientist is to derive meaning and insights from data, where the Data Engineer is building the pipeline that collects data from various locations, organizes it, and in most cases, persists to something like a data lake. Data Engineers work with Application Developers and Data Scientists to make sure application requirements are met in the increasingly distributed nature of data.

The most critical aspect of your speed and agility is how well your tools work together. When developers dream up new applications, how fast can that idea turn into a production deployment? Deploying and managing separate infrastructure (streaming, persistence, microservices) for one application is burdensome and prone to error. When asking why you would want to add streaming into your cloud native stack, you should consider the cost of not integrating your entire stack in terms of technical debt.

Cloud native streaming is game-changing but remember the fundamentals

With Jesse Anderson, Managing Director, Big Data Institute

What makes streaming a good fit for Kubernetes? If you think about which component in your system that is the most dynamic, it’s probably streaming. Your database won’t have as much need to scale up and down in the course of a day. The typical demand curve in a 24 hour period is going to require more scaling for streaming, especially the processing. If you’re moving to Kubernetes from virtual machines, you will be tempted to copy your exact environment into pods and forget about it. By doing this, you are missing the primary value of cloud native for to streaming workloads. In my experience, teams pre-provisioning for expected loads typically end up wasting over 50% of resources by over-provisioning. The best way to manage cost is to add resources when needed and release them when you are finished. The real measurement of success is when end users have no idea that infrastructure is coming and going. They get a smooth experience and a consistent service level. On the other hand, artificially constraining your streaming capacity due to costs can reduce
response times and degrade service levels. In the worst case, a situation where the real-time processing window falls behind without any way to catch up.

The challenge in deploying streaming workloads in Kubernetes is one of matching system architectures to balance provisioning and service levels. If the technology wasn't designed with the idea of dynamic workload matching it could take a lot of effort to force it to do something it wasn't designed to accomplish. Kafka is a highly scalable distributed system, but the idea of scaling down wasn't part of the initial design. A Kafka cluster is designed to maintain the declared operational state. If ten brokers have been deployed and one is lost, Kafka tries to return to the state of ten brokers. While this is a critically important feature for resiliency, it takes a different approach to achieve elasticity. Pulsar is an example of a streaming system that has been designed with cloud native thinking to handle dynamic workloads from day one. Flink is a stream processing system designed with the same considerations. Used in combination, a deployment will consume compute and storage at different times and in different volumes. That is a closer match to the Kubernetes architecture.

Storage has been an area of rapid change for the Kubernetes project but one that you should avoid making assumptions about in your streaming deployments. When the data you are streaming needs to be persisted, where is it going? A great resilience question to ask is “What happens if I mistakenly delete my Kubernetes cluster?” I have worked with teams deploying streaming on Kubernetes who were unknowingly using ephemeral storage by mistake. You have to make sure you are thinking about the durability of your storage from the earliest stages of your move to Kubernetes. Streaming requires a higher level of operational excellence. Having five nines or better isn’t optional. In contrast to a batch system where downtime isn’t a high impact, you can just rerun the job if there is a failure. With streaming, if you are down, you’ve potentially lost data. Having an operational outage due to losing a StatefulSet can be a big deal.

The final thing to consider is your disaster recovery plan. Do not assume that cloud native deployments eliminate potentially devastating failures. You can mitigate many of them but in my experience, some amount of failure is inevitable which is why planning is so important. At a minimum, be ready for the various failures that can happen with infrastructure, such as loss of a Pod, a StatefulSet, or an entire Kubernetes cluster. The most common and impactful failures are due to human error, like purposefully deleting data thinking you are working in a QA environment, or getting a configuration wrong. It happens to everyone, and we just need to plan for it.

For Data Engineers and Site Reliability Engineers (SREs), your planning and implementation of streaming in Kubernetes can make a huge impact on your organization. Cloud native data should allow for more agility and speed while squeezing out all the efficiency you can get. As a reader of this book, you are already on your way to thinking differently about your infrastructure. Taking some advice from Jesse Anderson, there are two areas you should be thinking about as you begin your journey into streaming data on Kubernetes.
Resource Allocation

Are you planning for peaks as well as the valleys? As you’ll recall from Chapter 1, elasticity is one of the more challenging aspects of cloud native data to get right. Scaling up is a commonly solved problem in large-scale systems, but scaling down can potentially result in data loss, especially with streaming systems. Traffic to resources needs to be redirected before they are decommissioned, and any data they are managing locally will need to be accounted for in other parts of the system. The risk involved with elasticity is what keeps it from being widely used and the result is a lot of unused capacity. Commit yourself to the idea that resources should never be idle and build streaming systems that use what they need and no more.

Disaster Recovery Planning

Moving data efficiently is an important problem to solve but just as important is how to manage inevitable failure. You can’t just rely on Kubernetes to handle recovery without understanding your data flows and durability requirements. This is about more than backing up data. How are Pods scheduled so that physical server failure has a reduced impact? Can you benefit from geographic redundancy? Are you clear on where data is persisted and understand the durability of those storage systems? And finally, do you have a clear plan to restore systems after a failure? In all of these cases, writing down the procedure is the first step, but testing those procedures is the difference between success and failure.

We’ve covered the what and why of streaming data on Kubernetes, and it’s time we start looking at the how with a particular focus on cloud native deployments. We’ll give a quick overview of how to install these technologies on Kubernetes and highlight some important details that will aid your planning. Since you’ve already learned how to use many of the Kubernetes resources we’ll need in previous chapters, we’ll speed up the pace a bit. Let’s get started on the first cloud native streaming technology.

Streaming on Kubernetes with Apache Pulsar™

Apache Pulsar™ is an exciting project to watch for cloud native streaming applications. Streaming software has mostly been built in an era before Kubernetes and cloud native architectures. Pulsar was originally developed at Yahoo! which is no stranger to high scale cloud native workloads. Donated to the Apache Software Foundation, it was accepted as a top level project in 2018. That’s not to say that projects like Apache Kafka™ or RabbitMQ are ill-suited for use in Kubernetes; they will just require more planning and well-written operators to function at the level of efficiency of Pulsar. In terms of the streaming definitions we covered previously, Pulsar supports the following characteristics
• Types of delivery: one-to-one and pub/sub
• Delivery guarantees: at-least-once, at-most-once, exactly-once
• Feature scope for streaming: Message broker, analytics (through functions)

So what makes Pulsar a good fit for Kubernetes?

We use Kubernetes to create virtual data centers to efficiently use compute, network and storage. Pulsar was designed from the beginning with a separation of compute and storage resource types linked by the network, similar to a microservices architecture. These resources can even span multiple Kubernetes clusters or physical data centers, as shown in Figure 6-4. This gives operators flexibility on how to install and scale a running Pulsar cluster based on use case and workload. It was also designed with multi-tenancy in mind which can make a big efficiency difference in large deployments. Instead of installing a separate Pulsar instance per application, many applications (tenants) can use one Pulsar instance with guardrails to prevent resource contention between each other. Finally, built-in storage tiering creates automated alternatives for storage persistence as data ages, and lower cost storage can be utilized.

Pulsar’s highest level of abstraction is an instance that consists of one or more clusters. We call the local logical administration domain a cluster and deploy in a Kubernetes cluster and where we’ll concentrate our attention. Clusters can share meta-data and configuration, allowing producers and consumers to see a single system regardless of location. Each cluster is made of several parts acting in concert that primarily consume either compute or storage which we will note.
The Bookkeeper project provides infrastructure for managing distributed write-ahead logs. In Pulsar the individual instances are called bookies and use Bookkeeper as the durable message store when persistence is enabled. The unit of storage is called a ledger and each topic can have one or more ledgers. Multiple bookie instances are created to provide load-balancing and failure protection. They also offer storage tiering functionality, allowing operators to provide a mix of fast and long-term storage options based on use case. When brokers interact with bookies, they read and write to a topic ledger which is an append-only data structure. Bookies provide a single reference to the ledger but manage the repli-
cation and load balancing behind the primary interface. In a Kubernetes environment, knowing where data is stored is critical for maintaining resilience.

**Apache Zookeeper™ (Compute)**

Zookeeper is a stand-alone project used in many distributed systems for coordination, leader election, and metadata management. Pulsar uses Zookeeper for service coordination similarly to the way Etcd is used in a Kubernetes cluster, storing important metadata such as tenants, topics, and cluster configuration state so that the brokers can remain stateless. Bookies use Zookeeper for ledger metadata and coordination between multiple storage nodes.

**Proxy (Network)**

The proxy is a solution for dynamic environments such as Kubernetes. Instead of exposing every broker to HTTP traffic, the proxy serves as a gateway and creates an ingress route to the Pulsar cluster. As brokers are added and removed, the proxy uses service discovery to keep the connections flowing to and from the cluster. When using Pulsar in Kubernetes the proxy service IP should be the single access for your applications to a running Pulsar cluster.

**Functions (Compute)**

Separate from message-passing functionality of the brokers and not included in Figure 6-4. Pulsar Functions operate independently and consume their own compute resources. The component added to a Pulsar cluster is the worker which accepts function runtimes on an ad-hoc basis. Pulsar Functions work in conjunction with the message broker. When deployed, they take data from a topic, alter it with user code, and then return it to a different topic. Operators have the option to deploy Functions as a part of a larger cluster or as a stand-alone for more fine-grained resource management.

**Preparing Your Environment**

When preparing to do your first installation, you need to make some choices. For the most complete and up-to-date information on installing Pulsar in Kubernetes check the official documentation. For this book, we will look closer at why you make certain choices and how they pertain to different cloud native application use cases.

To begin, create a local clone directory of the Pulsar Helm chart repository:

```bash
git clone https://github.com/apache/pulsar-helm-chart
```

This subproject of Pulsar is well documented with several helpful examples to follow. When using Helm to deploy Pulsar you will need a values.yaml file that contains all of the options to customize your deployment. You can include as many parameters as you want to change. The Pulsar Helm chart has a complete set of defaults for a typical cluster that might work for you, but with even the best defaults, you will have nuance.
in your environment that requires specific values. The examples directory has various deployment scenarios. If you choose the default installation as described in the TBD file, you’ll end up with a set of resources like that shown in Figure 6-5. As you can see, the installation wraps the proxy and brokers in Deployments and provides a service...

Affinity is a mechanism built into Kubernetes to create rules for which pods can and cannot be co-located on the same physical node (See more in-depth discussion in chapter 4). Pulsar, being a distributed system, has deployment requirements for maximum resilience. An example is brokers. When multiple brokers are deployed, each pod should run on a different physical node in case of failure. If all broker pods were grouped on the same node and the node went down, the Pulsar cluster would be unavailable. Kubernetes would still recover the runtime state and restart the pods. However, there would be downtime as they came back online. The easiest thing is to not allow pods of the same type to group together onto the same nodes. When enabled, anti-affinity will keep this from happening. If you are running on a single node system such as a desktop, disabling it will allow your cluster to start without blocking based on affinity.
affinity:
  - anti_affinity: true

Fine-grained control over Pulsar component replica counts lets you tailor your deployment based on the use case. Each replica pod consumes resources and should be considered in the application's lifecycle. For example, starting with a low number of brokers and bookkeeper pods can manage some level of traffic, but as it increases, more replicas can be added and configuration updated via Helm.

zookeeper:
  - replicaCount: 1
bookkeeper:
  - replicaCount: 1
broker:
  - replicaCount: 1
proxy:
  - replicaCount: 1

Securing Communications by Default with Cert-manager

It's a toss-up on what gets left to the end of product development: security or documentation. Unfortunately, Kubernetes doesn't have much in the way of building documentation but when it comes to security there has been some great progress!

As you can see, installing Pulsar has created a lot of infrastructure, with a lot of communication between the elements. This is a typical situation. When we build out virtual data centers in Kubernetes, it will create a lot of internode and external network traffic. All of that traffic should be encrypted with Transport Layer Security (TLS) and Secure Socket Layer (SSL) using x.509 certificates. The most important part of this system is the Certificate Authority (CA) which in a Public Key Infrastructure (PKI) arrangement acts as a trusted third party that digitally signs certificates used to create a chain of trust between two entities. Going through the procedure to have a certificate issued by CA historically has been a manual and arduous process, which unfortunately has led to a lack of secure communications in cloud-based applications.

Cert-manager is a tool that uses the Automated Certificate Management Environment (ACME) protocol to add certificate management seamlessly to your Kubernetes infrastructure. We should always use TLS to secure the data moving from one service to another for our streaming application. The cert-manager project is arguably one of the most critical pieces of your Kubernetes infrastructure that you will eventually forget about. That’s the hallmark of a project that fits the moniker of “it just works.”
What is ACME?

When working with x.509 certificates, you'll frequently see references to the Automated Certificate Management Environment (ACME). ACME allows for automated deployment of certificates between user infrastructure and certificate authorities. It was designed by the Internet Security Research Group when they were building their free certificate authority, Let's Encrypt. It would be putting it lightly to say this fantastic free service has been a game-changer for cloud native infrastructure.

Adding TLS to your Pulsar deployment has been made incredibly easy with just a few configuration steps. Before installing Pulsar, you'll need to set up the Cert-manager service inside the target Kubernetes cluster. First, add the Cert-manager repo to your local Helm installation.

```
helm repo add jetstack https://charts.jetstack.io
```

The installation process takes a few parameters which you should make sure to use. First is declaring a separate namespace to keep the Cert-manager neatly organized in your virtual datacenter. The second is installing the Custom Resource Description (CRD) assets. This allows you to create the services that automate your certificate management.

```
helm install 
  cert-manager jetstack/cert-manager 
  --namespace cert-manager 
  --create-namespace 
  --set installCRDs=true
```

After the Cert-manager is installed, you'll then need to configure the certificate issuer that will be called when new certificates are needed. There are many different options based on the environment you are operating in and these are covered quite extensively in the documentation. One of the custom resources created when installing cert-manager is Issuer. The most basic Issuer is the selfsigned-issuer that can create a certificate with a user-supplied private key. You can create a basic Issuer by applying the following yaml configuration.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
  namespace: cert-manager
spec:
  selfSigned: {}
...
```

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
```

156 | Chapter 6: Streaming Data on Kubernetes
name: selfsigned-cluster-issuer
spec:
  selfSigned: {}

When installing Pulsar with Helm, you can secure inter-service communication with a few lines of yaml configuration. You can pick which services are secured by setting the TLS enabled to true or false for each service in the yaml that defines your Pulsar cluster.

tls:
  # settings for generating certs for proxy
  proxy:
    enabled: true
    cert_name: tls-proxy
  # settings for generating certs for broker
  broker:
    enabled: true
    cert_name: tls-broker
  # settings for generating certs for bookies
  bookie:
    enabled: false
    cert_name: tls-bookie
  # settings for generating certs for zookeeper
  zookeeper:
    enabled: false
    cert_name: tls-zookeeper

Or you can secure the entire cluster with just one command.

tls:
  enabled: true

Later in your configuration file, you can use self signing certificates to create TLS connections between components.

# issue selfsigning certs
certs:
  internal_issuer:
    enabled: true
    type: selfsigning

If you have been involved in securing infrastructure communication any time in the past, you know the toil in working through all the steps and applying TLS. Inside a Kubernetes virtual data center, you no longer have an excuse to leave network communication unencrypted. With a few lines of configuration, everything is secured and maintained.

Cert-manager: Making security easy so you’ll use it
With Josh van Leeuwen, Software Engineer, Jetstack
Cert-manager is a project born of necessity as our cloud native world grows. Previously, you might have a bunch of virtual machines or bare metal running somewhere, running in a ringed fence. You could get away with sticking an SSL certificate in the front gateway and moving on. All of that has changed now with the thousands or even hundreds of thousands of machines that need to be secured in our cloud native systems. With all of these small containers running microservices continually coming and going, automation is the only way to manage the volume of changes. There is no way a human can do that alone. Of course, this opens a new challenge of reliable automation—one which Kubernetes has taken head-on.

Soon after the ACME protocol was created, custom resources and CRDs became a feature in Kubernetes. Cert-manager is a project that joins those two concepts, providing a declarative way to represent what an X.509 certificate should look like inside a Kubernetes deployment. ACME happened at just the right time for the Kubernetes ingress use case, and the first use case for Cert-manager was for ACME SSL certificates. However, it quickly became apparent that this would not be the only secure networking problem that needed solving in Kubernetes. Those growing numbers of machines all need to talk to each other, and they all need some kind of security in place, which is generally done with Transport Level Security (TLS). TLS certificates require the concept of an issuer, and Cert-manager was expanded to allow for different types of issuers to automate the complete lifecycle further of those certificates.

Because it emerged so early in the Kubernetes project, Cert-manager has become the de-facto X.509 provider and certificate manager. With this comes a responsibility to make securing communications in Kubernetes easy. Security is only as good as it is easy. If security is challenging to implement, then it's practically useless. Many people don't like GPG, for these reasons; not because it's necessarily flawed security-wise, but because it's challenging to use. Cert-manager should continue to see wide adoption in cloud native applications. It makes everything secure by default, with little intervention or minimal knowledge of how RSA or TLS works. It's a project which is easy to use and solves people's problems by default.

One thing that has made Cert-manager easy for end-users is having a well-defined API to describe their application requirements in a simple way. It is a way of abstracting the more complicated questions, such as what does it mean to have a certificate signed or an issuer? These APIs provide the guardrails to make sure you do the right thing as much as possible. There are still some things that require planning and thoughtfulness, such as not reusing private key passwords, which is allowed but discouraged.

Guardrails and standardization are topics that need to become more prevalent in other parts of Kubernetes. The declarative nature and extensibility of Kubernetes are powerful tools, but with great power comes great responsibility. Different people within an organization can make extension points in a Kubernetes cluster. With a single command, you can have an endpoint exposed on the internet without even realizing it. There is no single pane of glass available to security professionals for those
extensions. Nor are there guardrails to prevent unexpected behaviors. Without proper guardrails in place, it’s too easy to self-own quite badly. As Kubernetes matures, we’ll need more ways to avoid unhappy accidents.

The Cert-manager project is in an excellent state, being vendor-neutral and mature in its current form. If you grep the project changelog for the word “feature,” you’ll see a decrease in occurrence in each successive release. This means we have a core API that is useful and stable, which is an excellent place to be for a core security-based project. The bulk of changes happening in the project are focused on taking advantage of this stable core API to add new issuers. This stability ensures the project stays up-to-date with the latest requirements without a disruptive breaking change.

As for the future, the Cert-manager project will continue to work with the Kubernetes community to continue the path of “default secure” and make security so easy that it’s used universally. There are still some challenges to overcome, like how secrets are stored and how to manage trust chains, and the momentum of Kubernetes practically assures that those are problems that will be solved shortly. If these are interesting problems, I urge you to get involved in one of the many ways security professionals can impact the future of Kubernetes.

Cert-manager should be one of the first things you install in a new Kubernetes cluster. The combination of project maturity and simplicity makes security the easy first thing to add to your project instead of the last. This is true not only for Pulsar but for every service you deploy in Kubernetes that requires network communication.

### Using Helm to Deploy Apache Pulsar™

Now that we covered how to design a Pulsar cluster to maximize resources, you can use Helm to carry out the deployment into Kubernetes. First, add the Pulsar Helm repository.

```
helm repo add apache https://pulsar.apache.org/charts
```

One of the special requirements for a Helm install of Pulsar is preparing Kubernetes. In the git repository you cloned earlier, there is a script that will run through all the preparations such as creating the destination namespace. The more complicated setup is the roles with associated keys and tokens. These are important for inter-service communication inside the Pulsar cluster. From the docs you can invoke the prep script using this example.

```
./scripts/pulsar/prepare_helm_release.sh -n <k8s-namespace> -k <helm-release-name>
```

Once the Kubernetes cluster has been prepared for Pulsar, the final installation can be run. At this point you should have a yaml configuration file with the settings you need for your Pulsar use case like we described earlier. The helm install command will take that config file and direct Kubernetes to meet the desired state you have
specified. When creating a new cluster, use the initialize=true to create the base metadata configuration in Zookeeper.

```
helm install \
  --values <config yaml file> \
  --set initialize=true \
  --namespace <namespace from prepare script> \
  <pulsar cluster name> apache/pulsar
```

In a typical production deployment, you should expect the setup time to take 10 minutes or more. There are a lot of dependencies to walk through as Zookeeper, Bookies, Brokers, and finally Proxies are brought online and in order.

**Stream Analytics with Apache Flink™**

Now let's look at a different type of streaming project that is quickly gaining popularity in cloud native deployments: Apache Flink™. Flink is a system primarily designed to focus on stream analytics at incredible scale. As we discussed at the beginning of the chapter, streaming systems come in a lot of different flavors and this is a perfect example. Flink has its own competencies that overlap very little with other systems, in fact, it's very common to see Pulsar and Flink deployed together to complement each other's strengths in a cloud native application. As a streaming system, the following are available in Flink:

- Type of delivery: one-to-one
- Delivery guarantee: exactly-once
- Feature scope for streaming: analytics

The two main components of the Flink architecture are shown in Figure 6-6: the Job Manager and Task Manager.

![Figure 6-6. Apache Flink Architecture](image-url)
The Flink project is designed for managing stateful computations, which should cause you to immediately think of storage requirements. Every transaction in Flink is guaranteed to be strongly consistent with no single point of failure. These are features when you are trying to build the kind of highly scalable systems that Flink was designed to accomplish. There are two different types of streaming: bounded and unbounded.

**Unbounded streaming**

These streaming systems react to new data whenever the data arrives. There is no endpoint where you can stop and analyze the data gathered, every data received is independent. The use cases for this can be alerting on values or counting when exactness is important. Reactive processing can be very resource-efficient.

**Bounded streaming**

This is also known as batch processing in other systems but is a specific case within Flink. Bounded windows can be marked by time or certain values. In the case of time windows, they can also slide forward giving the ability to do rolling updates on values. Resource considerations should be given based on the size of the data window to be processed. The limit of the boundary size is mostly constrained by memory.

One of the foundational tenets of Flink is a strong focus on operations. At the scale required for cloud native applications, easy to use and deploy can be the difference between using it or not. This includes core support for running workloads in Kubernetes and feature parity with cloud native applications. Built-in support for savepoints makes updates easier by pausing and resuming jobs before and after system updates. They can also be used for fast recovery if a processing pod fails mid-job. Tighter integration with Kubernetes gives Flink the ability to self-heal on failure by restoring pods and restarting jobs with savepoints. Observability within the entire application is enabled with the support for the output of Prometheus metrics.

The fundamental unit of work for Flink is called a job. Jobs are Java or Scala programs that define how the data is read, analyzed, and output. Jobs are chained together into a Flink application which is compiled into a jar file. The Flink project provides a Docker image that allows the encapsulation of the application code in a deployable form. Holding up to the commitment to operations, having a docker image makes the orchestration on Kubernetes an easy task. The advantage of using Kubernetes with streaming applications can now be enhanced by allowing teams to use continuous deployment.

Put all together, we have a way for data teams to participate in the overall cloud native stack while giving operators everything needed to manage the entire deployment. Application developers building microservices can share a CI/CD pipeline with developers building the stream analytics of data generated from the application. As changes occur in any part of the stack, they can be integration tested entirely and
deployed as a single unit. Teams can move faster with more confidence knowing there aren't disconnected requirements that may show up in production. This sort of outcome is a strong argument to employ cloud native methodologies in your entire stack so time to see how this is done.

**Deploying Apache Flink™ on Kubernetes**

When deploying a Flink cluster into a running Kubernetes cluster there are a few things to consider. The Flink project has gone the route of offering what they call “Kubernetes Native” which programmatically installs the required Flink components without the need for kubectl or Helm. This may change in the future and there are already side-projects in the Flink ecosystem that bring a more typical experience Kubernetes operators might expect with operators and Helm charts. For now, we will discuss the official method endorsed by the project.

![Diagram](image)

**Figure 6-7. Deploying Flink on Kubernetes**

A running Flink cluster has two main components we'll deploy in pods.
**JobManager**

This is the control plane for any running Flink application code deployed. They consume CPU resources but only to maintain job control, no actual processing is done on the JobManager. In High Availability (HA) mode, which is exclusive to Flink running on Kubernetes, storage resources need to be allocated for metadata storage. When in HA mode, multiple standby JobManagers will be provisioned but remain idle until the primary is no longer available.

**TaskManager**

Where the work gets done on a running Flink job. TaskManagers are used by the JobManager to satisfy the chain of tasks needed in the application. A chain is the order of operation. In some cases, these operations can be run in parallel, and some need to be run in series. The TaskManager will only run one discrete task and pass it on. Resource management can be controlled through the number of TaskManagers in a cluster and execution slots per TaskManager. The current guidance says that you should allocate one CPU to each TaskManager or slot.

Those are the basic units but which deployment mode to use is the key consideration for your use case. They dictate how compute and network resources are utilized. Another thing of note is how to deploy on Kubernetes. As mentioned before, there are no official project operators or Helm charts. The Flink distribution contains command-line tools that will deploy into a running Kubernetes cluster based on the mode for your application. **Figure 6-8** shows the modes available for deploying Flink clusters in Kubernetes: Application Mode and Session Mode. Flink also supports a third mode called Per-Job mode, but this is not available for Kubernetes deployments, which leaves us with Application Mode and Session Mode.
Figure 6-8. Apache Flink™ Modes

The selection of either Application Mode or Session Mode comes down to resource management inside your Kubernetes cluster so let’s look at both so you can make an informed decision.

Application Mode isolates each Flink application into its own cluster. As a reminder, a Flink application jar can consist of multiple jobs chained together. The startup cost of the cluster can be minimized with a single application initialization and job graph. Once deployed, resources are consumed for client traffic and execution of the jobs in the application. Network traffic is much more efficient since there is only one Job-Manager and client traffic can be multiplexed.

To start in Application Mode, the flink command line is invoked with the target of kubernetes-application. You will need the name of the running Kubernetes cluster accessible via kubectl. The application to be run is contained in a docker image and the path to the jar file supplied in the command line. Once started, the Flink cluster is created, application code is initialized, and will then be ready for client connections.

$ ./bin/flink run-application \
  --target kubernetes-application \

$ ./bin/flink run-application \
  --target kubernetes-application \

Figure 6-8. Apache Flink™ Modes
Session Mode changes resource management by creating a single Flink cluster that can accept any number of applications on an ad-hoc basis. Instead of having multiple independent clusters running and consuming resources, you may find it more efficient to have a single cluster that can grow and shrink when new applications are submitted. The downside for operators is that you now have a single cluster that if fails, will take several applications with it. Kubernetes will restart the downed pods, but you will have a recovery time to manage as resources are re-allocated. To start in Session Mode, use the kubernetes-session shell file and give it the name of your running Kubernetes cluster. The default is for the command to execute and detach from the cluster. To re-attach or remain in an interactive mode with the running cluster use the execution.attached=true switch.

```bash
$ ./bin/kubernetes-session.sh \
  -Dkubernetes.cluster-id=<kubernetes cluster name> \ 
  -Dexecution.attached=true
```

This was a quick fly-by of a very massive topic but hopefully, it inspires you to look further. Adding Flink to your application isn’t just about choosing a platform to perform stream processing. In cloud native applications we should be thinking holistically about the entire application stack we are attempting to deploy in Kubernetes. The way Flink uses containers as encapsulation lends itself to working with other development workflows.

## Conclusion

In this chapter, we have branched out from persistence-oriented data infrastructure into the world of streaming. We defined what streaming is, how to navigate all the terminology, and how it fits into Kubernetes. From there we took a deeper look into Apache Pulsar and learned how to deploy it into your Kubernetes cluster according to your environment and application needs. As a part of deploying streaming, we took a side trip into default secure communications with Cert-manager to see how it works and how to create self-managed encrypted communication. Finally, we looked at Kubernetes deployments of Apache Flink, which is used primarily for high scale stream analytics.

As you saw in this chapter with Pulsar and Cert-manager, it’s frequently the case that running cloud-native data infrastructure on Kubernetes involves the composition of multiple components as part of an integrated stack. We’ll discuss more examples of this in the next chapter and beyond.
About the Authors

**Jeff Carpenter** works in Developer Relations at DataStax, where he uses his background in system architecture, microservices and Apache Cassandra to help empower developers and operations engineers to build distributed systems that are scalable, reliable, and secure. Jeff has worked on large-scale systems in the defense and hospitality industries and is co-author of Cassandra: The Definitive Guide.

**Patrick McFadin** works at DataStax in Developer Relations and as a contributor to the Apache Cassandra project. He has worked as Chief Evangelist for Apache Cassandra and as a consultant for DataStax, where he had a great time building some of the largest deployments in production. Previous to DataStax, he held positions as Chief Architect, Engineering Lead and Database DBA/Developer.