O'REILLY®

Managing Cloud Native Data on Kubernetes

Architecting Cloud Native Data Services Using Open Source Technology

Early Release
Raw & Unedited

Compliments of

Jeff Carpenter & Patrick McFadin

portworx
by Pure Storage

Uncomplicate Data on Kubernetes

Make your data services scalable, available and secure on Kubernetes

Get Started Today

www.dbooks.org

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

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Managing Cloud Native Data on Kubernetes

by Jeff Carpenter and Patrick McFadin

Copyright © 2023 Jeff Carpenter and Patrick McFadin. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Jill Leonard
Production Editor: Christopher Faucher
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

January 2023: First Edition

Revision History for the Early Release

2021-10-01: First release
2021-12-08: Second release
2022-02-09: Third release
2022-04-26: Fourth release
2022-05-20: Fifth release
2022-07-08: Sixth release
2022-08-30: Seventh release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098111397* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Managing Cloud Native Data on Kubernetes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Portworx. See our statement of editorial independence.

978-1-098-11132-8

www.dbooks.org

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 15
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 23
Summary 26

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

Kubernetes Storage Architecture 50
Flexvolume 50
Container Storage Interface (CSI) 51
Container Attached Storage 53
Container Object Storage Interface (COSI) 55
Summary 58

3. Databases on Kubernetes the Hard way. 59
The Hard Way 60
Prerequisites for running data infrastructure on Kubernetes 61
Running MySQL on Kubernetes 61
ReplicaSets 62
Deployments 64
Services 68
Accessing MySQL 71
Running Apache Cassandra on Kubernetes 73
StatefulSets 75
Accessing Cassandra 86
Summary 88

4. Automating Database Deployment on Kubernetes with Helm. 89
Deploying Applications with Helm charts 90
Using Helm to deploy MySQL 91
Using Helm to deploy Apache Cassandra 103
Helm Limitations 107
Summary 109

5. Automating Database Management on Kubernetes with Operators. 111
Extending the Kubernetes Control Plane 112
Extending Kubernetes Clients 113
Extending Kubernetes Control Plane Components 113
Extending Kubernetes Worker Node Components 115
The Operator Pattern 116
Controllers 116
Events 117
Custom Resources 118
Operators 120
Managing MySQL in Kubernetes using the Vitess Operator 122
Vitess Overview 122
PlanetScale Vitess Operator 126
A Growing Ecosystem of Operators 136
Choosing Operators 136

www.dbooks.org

Summary 142

6. Integrating Data Infrastructure in a Kubernetes Stack. 143
K8ssandra: Production-ready Cassandra on Kubernetes 144
K8ssandra Architecture 144
Installing the K8ssandra Operator 145
Creating a K8ssandraCluster 149
Managing Cassandra in Kubernetes with Cass Operator 151
Enabling Developer Productivity with Stargate APIs 155
Unified Monitoring Infrastructure with Prometheus and Grafana 158
Performing Repairs with Cassandra-Reaper 162
Backing up and Restoring Data with Cassandra Medusa 163
Deploying Multi-cluster applications in Kubernetes 167
Summary 173

7. The Kubernetes Native Database. 175
Why a Kubernetes native approach is needed 176
Hybrid data access at scale with TiDB 178
TiDB Architecture 179
Serverless Cassandra with DataStax Astra DB 191
What to look for in a Kubernetes Native Database 199
Basic requirements 199
The Future of Kubernetes Native 201
Summary 203

8. Streaming Data on Kubernetes. 205
Introduction to Streaming 205
Types of delivery 206
Delivery Guarantees 207
Feature scope 209
The Role of Streaming in Kubernetes 210
Streaming on Kubernetes with Apache Pulsar™ 213
Preparing Your Environment 215
Securing Communications by Default with Cert-manager 217
Using Helm to Deploy Apache Pulsar™ 222
Stream Analytics with Apache Flink™ 222
Deploying Apache Flink™ on Kubernetes 225
Summary 227

9. Data Analytics on Kubernetes. 229
Introduction to Analytics 230
Deploying Analytic Workloads in Kubernetes 232

Introduction to Apache Spark™ 234
Deploying Apache Spark™ in Kubernetes 236
Kubernetes Operator for Apache Spark 239
Alternative Schedulers for Kubernetes 243
Apache YuniKorn™ 245
Volcano 247
Analytic Engines for Kubernetes 250
Dask 252
Ray 253
Summary 255

10. Machine Learning and Other Emerging Use Cases for Data on Kubernetes. 257
The Cloud Native AI/ML Stack 258
AI/ML Definitions 258
Defining an AI/ML Stack 261
Real-time Model Serving with KServe 262
Full-Lifecycle Feature Management with Feast 265
Efficient Data Movement with Apache Arrow 271
Versioned Object Storage with LakeFS 274
Summary 277

www.dbooks.org

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. The GitHub repo is 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: 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, 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 do 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.

www.dbooks.org

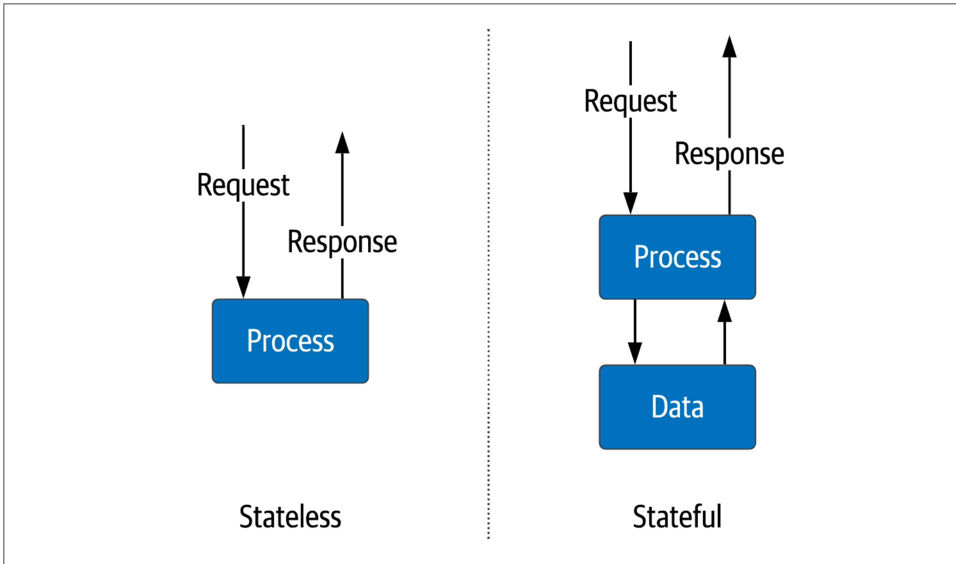


Figure 1-1. Stateless vs. Stateful Services

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. This creates the need for consistency and mechanisms to keep data in sync between replicas. 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 have given way to virtual machines that are easy to deploy and maintain
- Virtual machines have been 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.

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. [1]

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.

1 From What is Kubernetes (https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/)

www.dbooks.org

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. [2]

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 appli-

2 From CNCF git repository (https://github.com/cncf/toc/blob/main/DEFINITION.md)

cations 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.

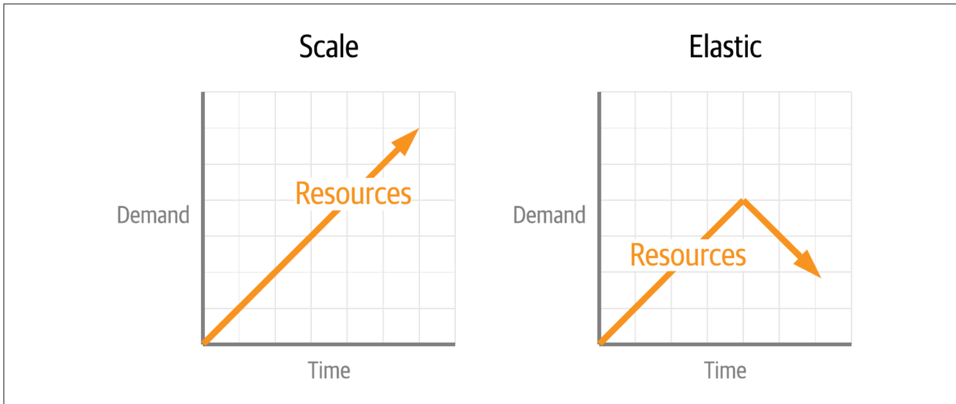


Figure 1-2. Comparing Scalability and Elasticity

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.

www.dbooks.org

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 needs 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.

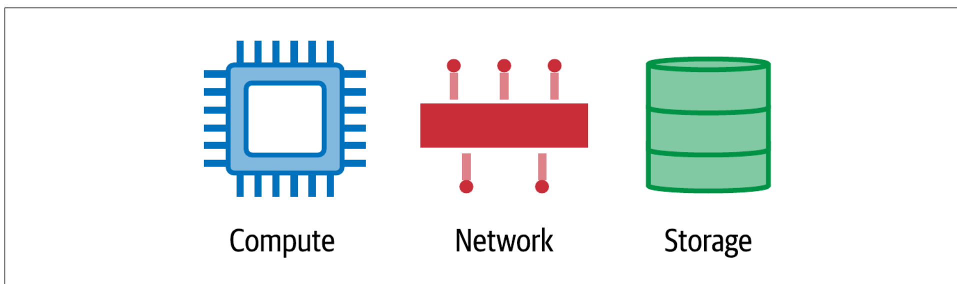


Figure 1-3. Fundamental resources of cloud applications: compute, network, and storage

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 data centers as a complete definition as shown in Figure 1-4.

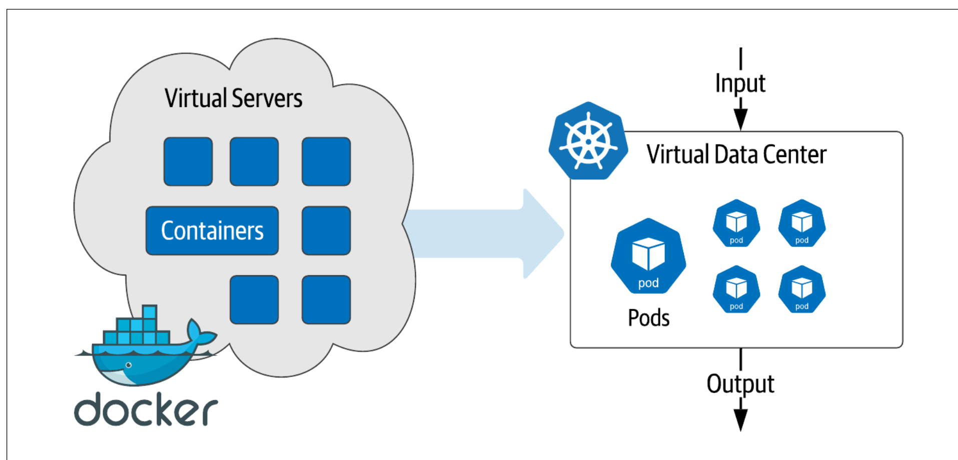


Figure 1-4. Moving from virtual servers to virtual data centers

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 par-

www.dbooks.org

allel 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 sizable 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

www.dbooks.org

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 steaming systems that move data are Apache Flink™ and Apache Kafka™, whereas 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 is 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 offloading 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

www.dbooks.org

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.[3]

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

3 SRE definition from Wikipedia (https://en.wikipedia.org/wiki/Site_reliability_engineering)

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.

www.dbooks.org

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:

8 Fallacies of Distributed Computing[4]

- The network is reliable;
- Latency is zero;
- Bandwidth is infinite;
- The network is secure;
- Topology doesn't change;
- There is one administrator;
- Transport cost is zero;
- 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, consider 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

4 https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

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 compute, 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 compute, 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.

www.dbooks.org

Principle 2: Separate the control and data planes

Kubernetes promotes the separation of control and data planes. The Kubernetes API server is the front door of the control plane, providing the interface used by the data plane 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 Kuberentes 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!

www.dbooks.org

CHAPTER 2

Managing Data Storage on Kubernetes

> ### 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. The GitHub repo is 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.

www.dbooks.org

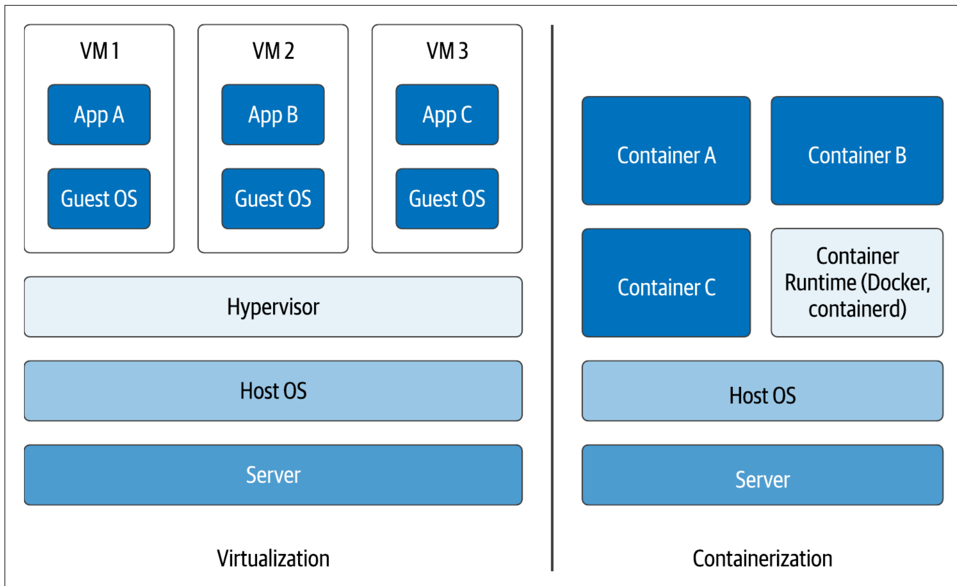


Figure 2-1. Comparing containerization to virtualization

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.

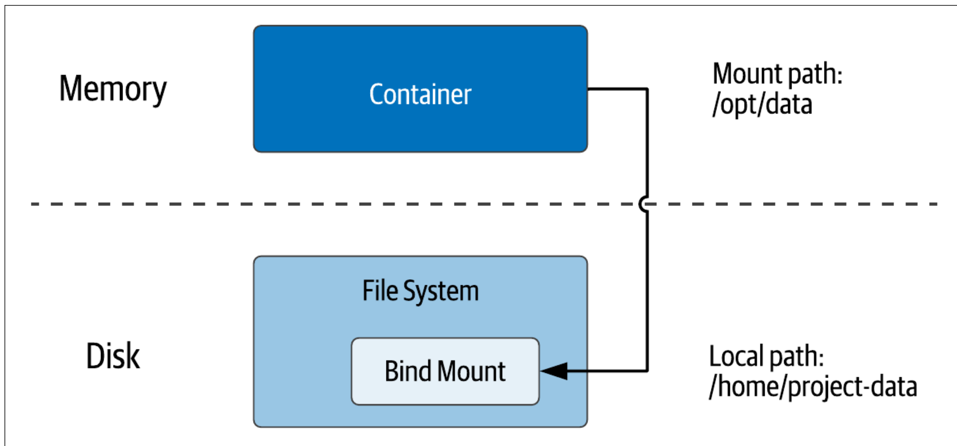


Figure 2-2. Using Docker Bind Mounts to access the host filesystem

When starting a container within Docker, you specify a bind mount with the --volume or -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 -d --name web -p 8080:80 -v ~/site-content:/usr/share/nginx/
html nginx
```

This exposes the webserver on port 8080 on your local host. 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

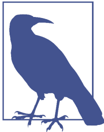
The preferred option within Docker is to use volumes. Docker volumes are created and managed by Docker under a specific directory on the host filesystem. The Docker volume create command is used to create a volume. For example, you might create a volume called site-content to store files for a website:

www.dbooks.org

```
docker volume create site-content
```

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 ro option:

```
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.

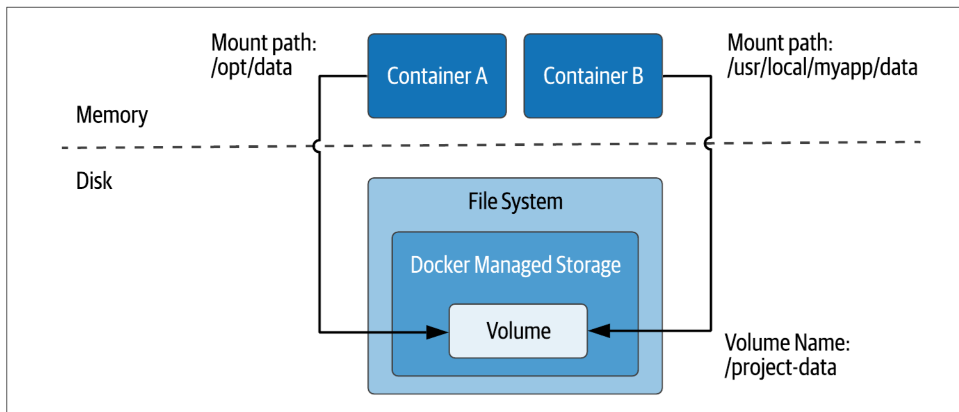


Figure 2-3. Creating Docker Volumes to share data between containers on the host

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.

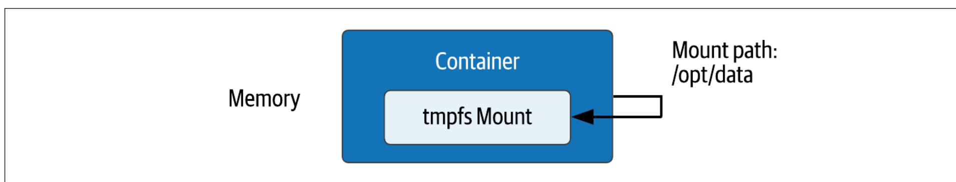


Figure 2-4. Creating a temporary volume using Docker tmpfs

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.

www.dbooks.org

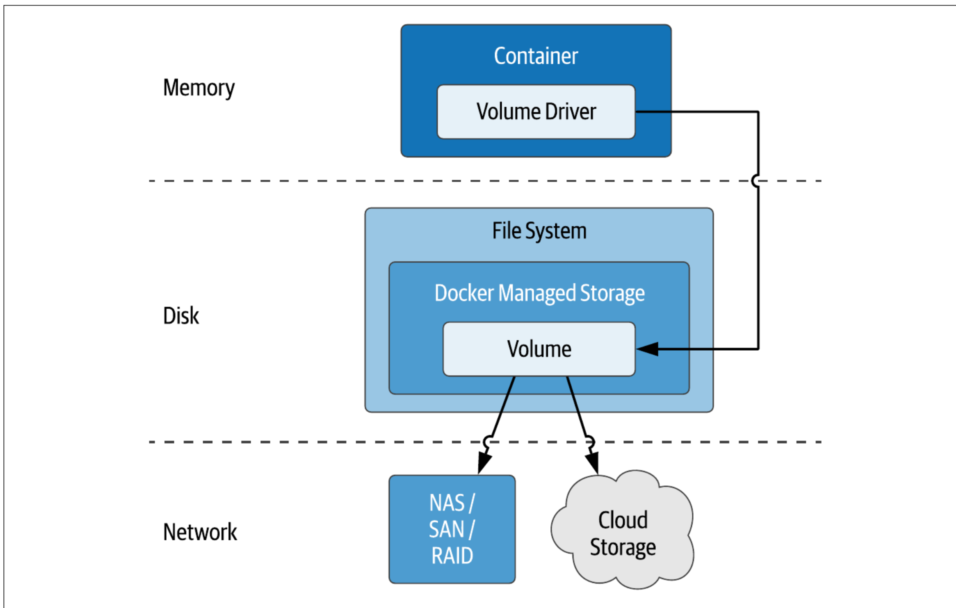


Figure 2-5. Using Docker Volume Drivers to access networked storage

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 flexi-

bility 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.

www.dbooks.org

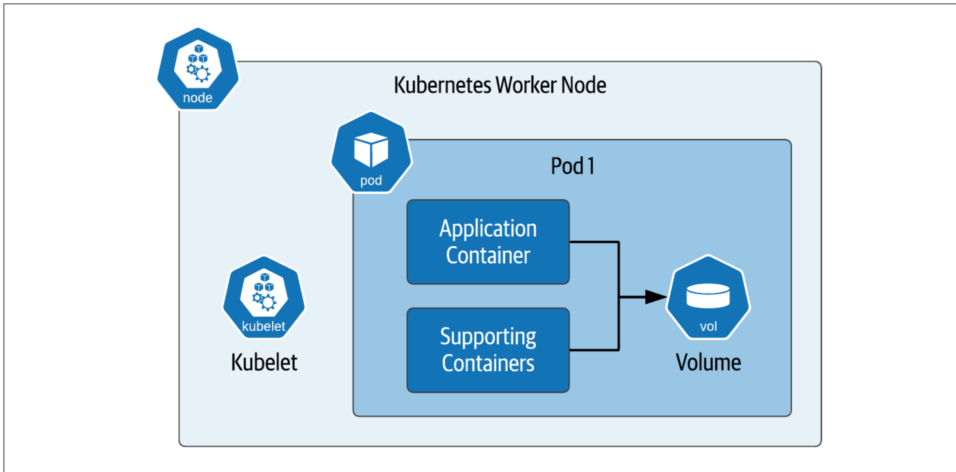


Figure 2-6. Using Volumes in Kubernetes Pods

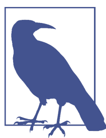
Now let's consider how persistence is supported within this pod architecture. As with Docker, the "on disk" data in a container is lost when a container crashes. The kubelet is responsible for restarting the container, but this new container is really a replacement for the original container - it will 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.

Using a volume in a pod requires two steps: defining the volume, and mounting the volume in each container that needs access. Let's look at a sample YAML configuration that defines a pod with a single application container, the Nginx web server, and a single volume (source code):

```
apiVersion: v1
kind: 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:

```
kubectl apply -f nginx-pod.yaml
```

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

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

And the results might look something like this:

www.dbooks.org

```
volumes:
- emptyDir: {}
  name: web-data
- name: default-token-2fp89
  secret:
    defaultMode: 420
```

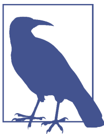
As you can see, Kubernetes supplied some additional information when creating the requested volume, defaulting it to a type of emptyDir. Other default attributes may differ depending on what Kubernetes engine you are using and we won't discuss them further here.

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

www.dbooks.org

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.

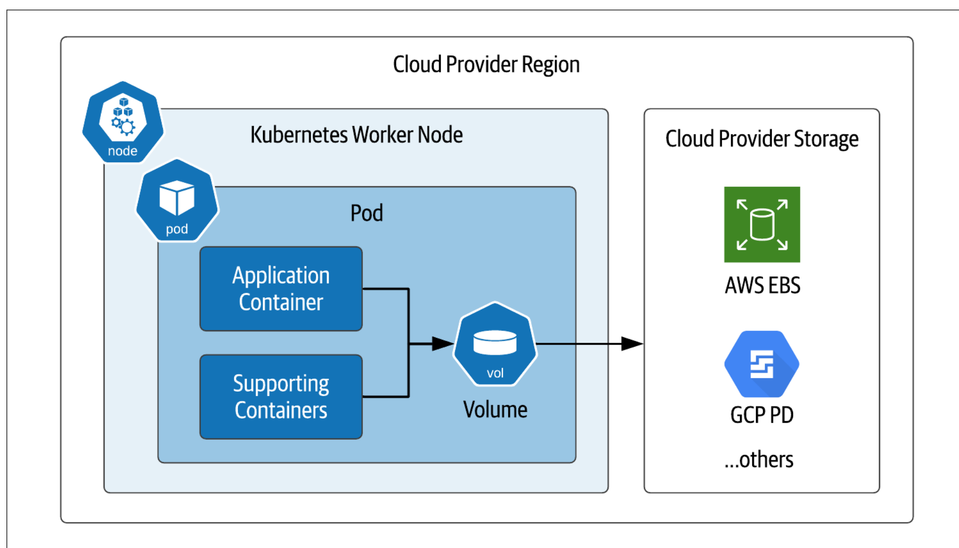


Figure 2-7. Kubernetes pods directly mounting cloud provider storage

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 Open‐Stack 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. Comparing Docker and Kubernetes storage options

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

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 vol‐

www.dbooks.org

ume. 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.

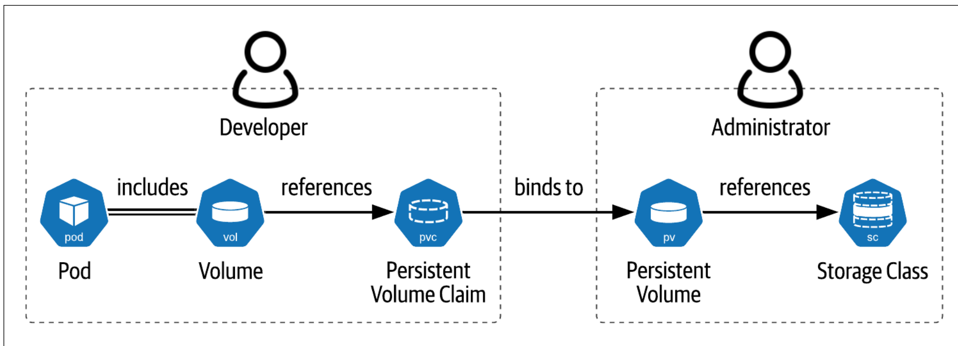


Figure 2-8. PersistentVolumes, PersistentVolumeClaims, and StorageClasses

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.

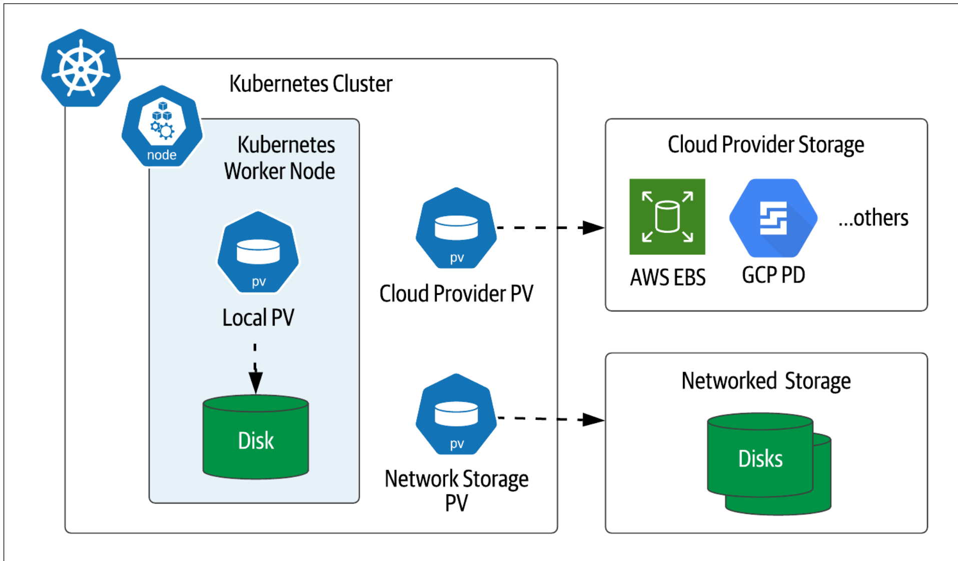


Figure 2-9. Types of Kubernetes PersistentVolumes

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.

www.dbooks.org

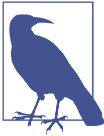
Like hostPath volumes, a local volume may also represent a directory. A key difference between local and hostPath volumes is that when a pod using a local volume is restarted, the Kubernetes scheduler ensures the pod is rescheduled on the same node so that it can be attached to the same persistent state. 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):

```
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 node at a time, although multiple pods running on that node may access the volume
- ReadOnlyMany access means the volume can be mounted by multiple nodes 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:        [kubernetes.io/pv-protection]
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:
Source:
```


www.dbooks.org

```
    Type:  LocalVolume (a persistent volume backed by local storage on a node)
    Path:  /app/data
Events:    <none>
```

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.

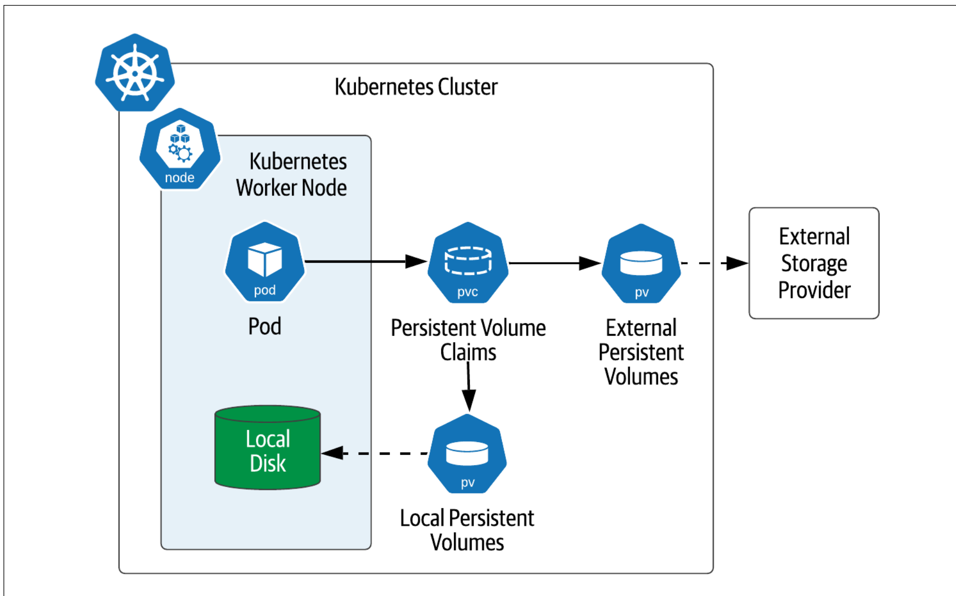


Figure 2-10. Accessing PersistentVolumes using PersistentVolumeClaims

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):

```
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):

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


www.dbooks.org

```
- name: nginx
  image: nginx
  volumeMounts:
  - mountPath: "/app/data"
    name: my-volume
volumes:
- name: my-volume
  persistentVolumeClaim:
    claimName: my-claim
```

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:

```
$ kubectl describe pvc my-claim
Name:          my-claim
Namespace:     default
StorageClass:
Status:        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.

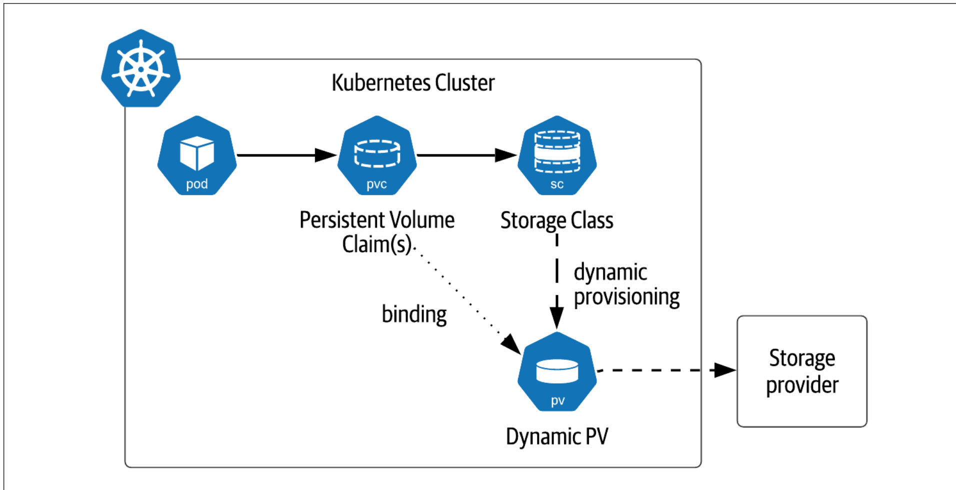


Figure 2-11. StorageClasses support dynamic provisioning of volumes

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:

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

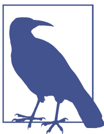
This storage provider comes pre-installed in K3s, a desktop distribution also provided by Rancher. If you take a look at the YAML configuration referenced in that statement, you'll see the following definition of a StorageClass (source code):

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-path
provisioner: rancher.io/local-path
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Delete
```


www.dbooks.org

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 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.
- 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.
- 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.
- Some StorageClasses also define parameters, 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.



Limits on dynamic provisioning

Local PVs 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):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
```

```
  name: my-local-path-claim
spec:
  storageClassName: local-path
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

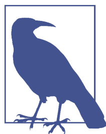
If no storageClass is specified in the claim, then the default StorageClass is used. The default StorageClass can be set by the cluster administrator. As we showed above in the Persistent Volumes section, you can opt out of using StorageClasses by using the empty string, which indicates that you are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

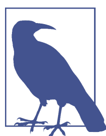
Flexvolume

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 advan-

www.dbooks.org

tage 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.

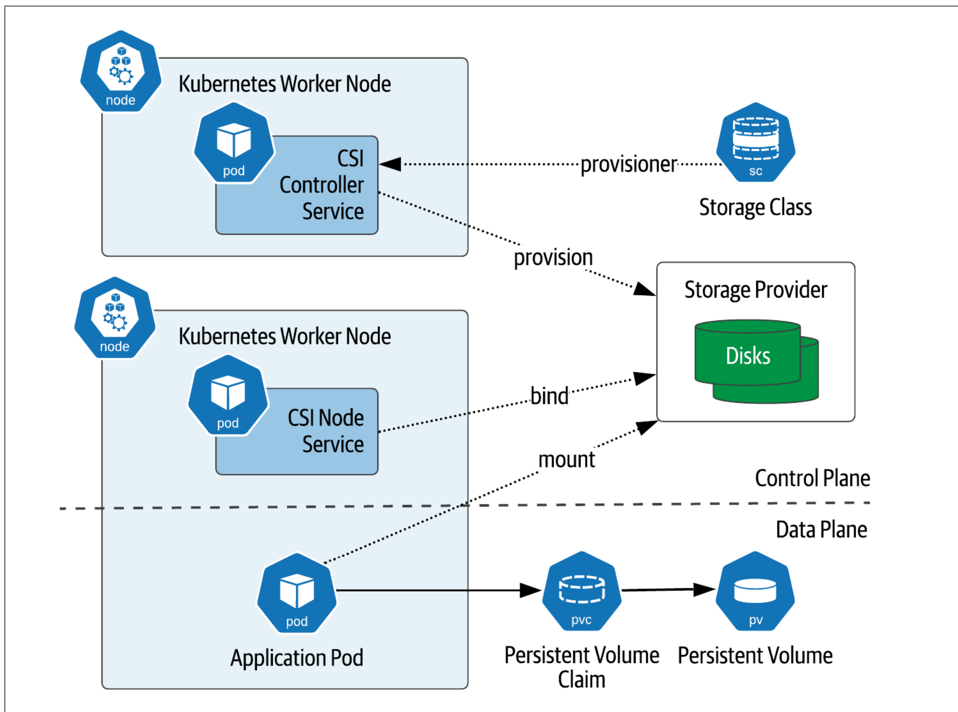
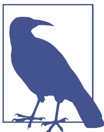


Figure 2-12. Container Storage Interface mapped to Kubernetes



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.

www.dbooks.org

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 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 environ-

ments. 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.

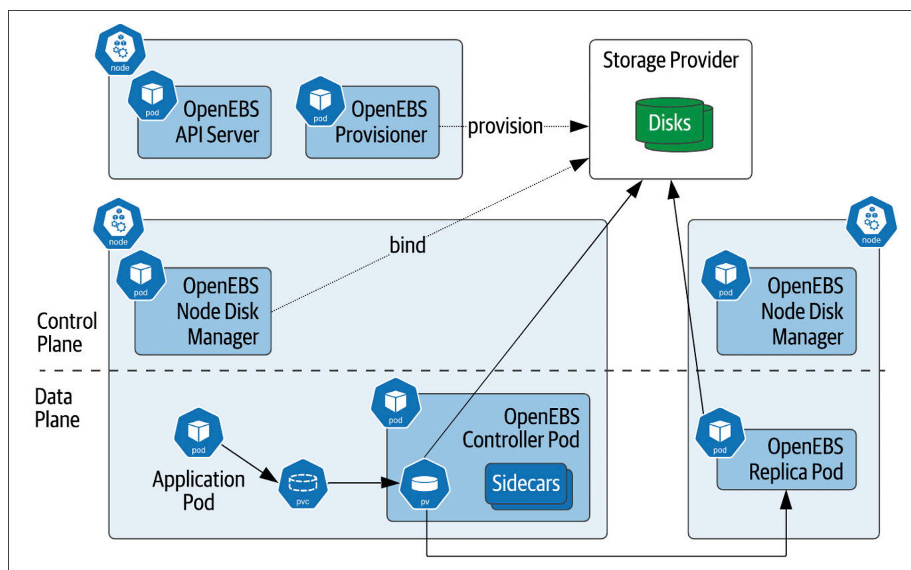


Figure 2-13. OpenEBS Architecture

www.dbooks.org

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:

www.dbooks.org

- 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.

www.dbooks.org

CHAPTER 3

Databases on Kubernetes the Hard way

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 3rd chapter of the final book. The GitHub repo is 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.

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.

www.dbooks.org

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.

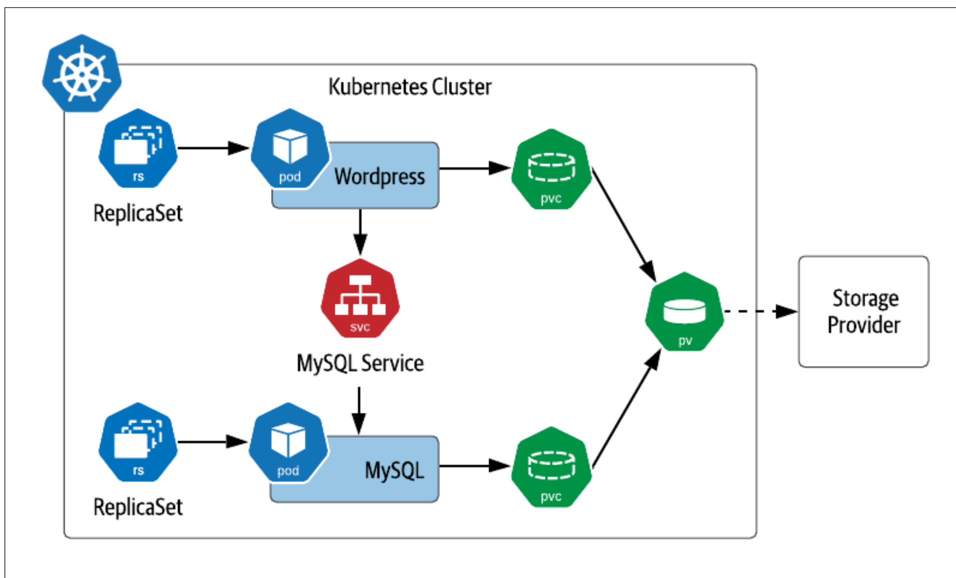


Figure 3-1. Sample Kubernetes Deployment of MySQL

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

www.dbooks.org

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:

```
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:8.0
        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.

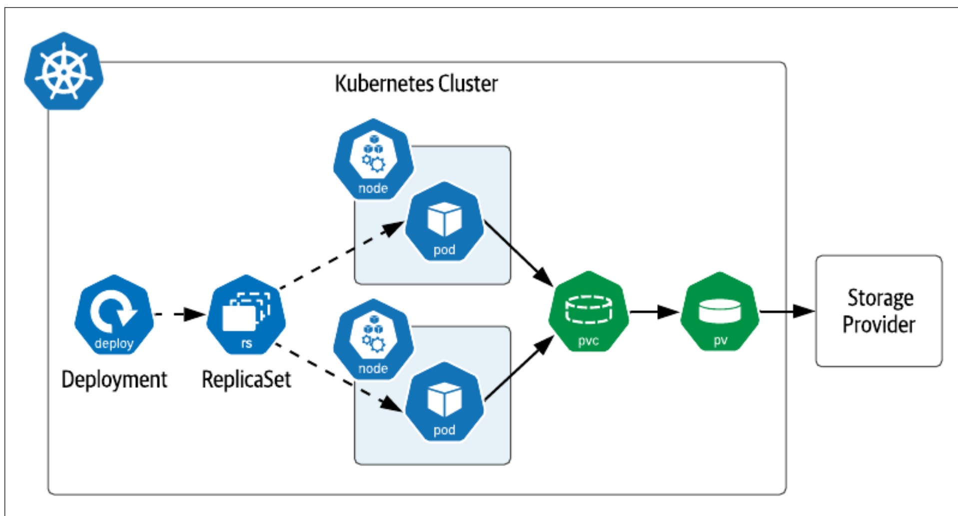


Figure 3-2. Deployments and ReplicaSets

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

www.dbooks.org

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

Next, create a Deployment with a pod template 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
kind: 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:8.0
        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 deployments,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 relationships 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:

www.dbooks.org

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 emphasizes high availability. This is also useful for data migration.

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:

```
kubectl 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):

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
```


www.dbooks.org

```
    tier: mysql
  clusterIP: None
```

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 cluster-internal IP address. ClusterIP Services are the type 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 from a range of ports (30000-32767 by default), 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 an external 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 typi-

cally 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.

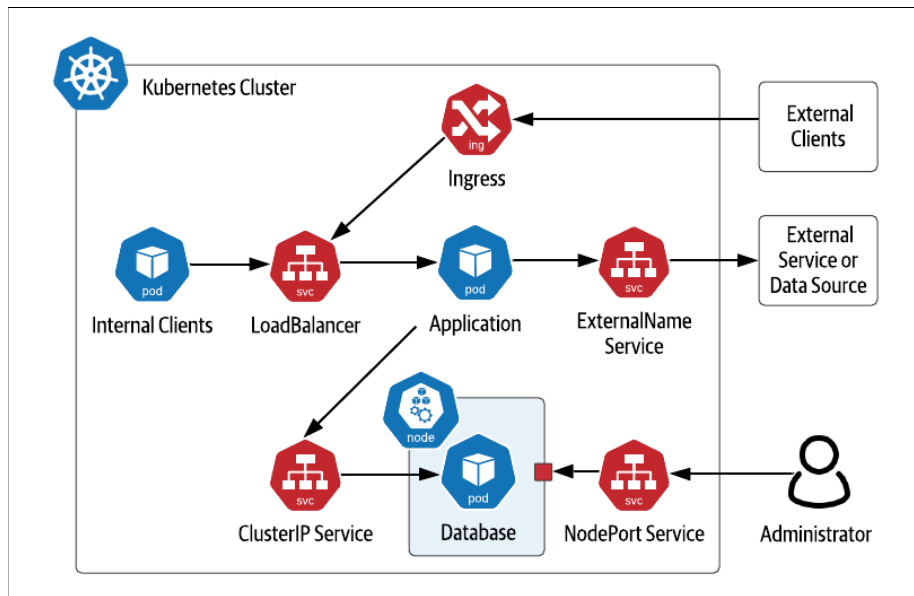


Figure 3-3. Kubernetes Service Types

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.

www.dbooks.org

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):

```
apiVersion: v1
kind: 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):

```
apiVersion: apps/v1
kind: 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:
      - image: wordpress:4.8-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
```

```
              secretKeyRef:
                name: mysql-root-password
                key: password
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

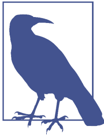
Notice that the database host and password for accessing MySQL are passed to Wordpress as environment variables. The value of the host is the name of the service you created for MySQL above. This is all that is needed for the database connection to be routed to your MySQL instance. The value for the 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):

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    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.

www.dbooks.org



Accessing Services from Kubernetes distributions

The exact details of accessing services will depend 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 distribution, 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:

```
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 datacenters 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.

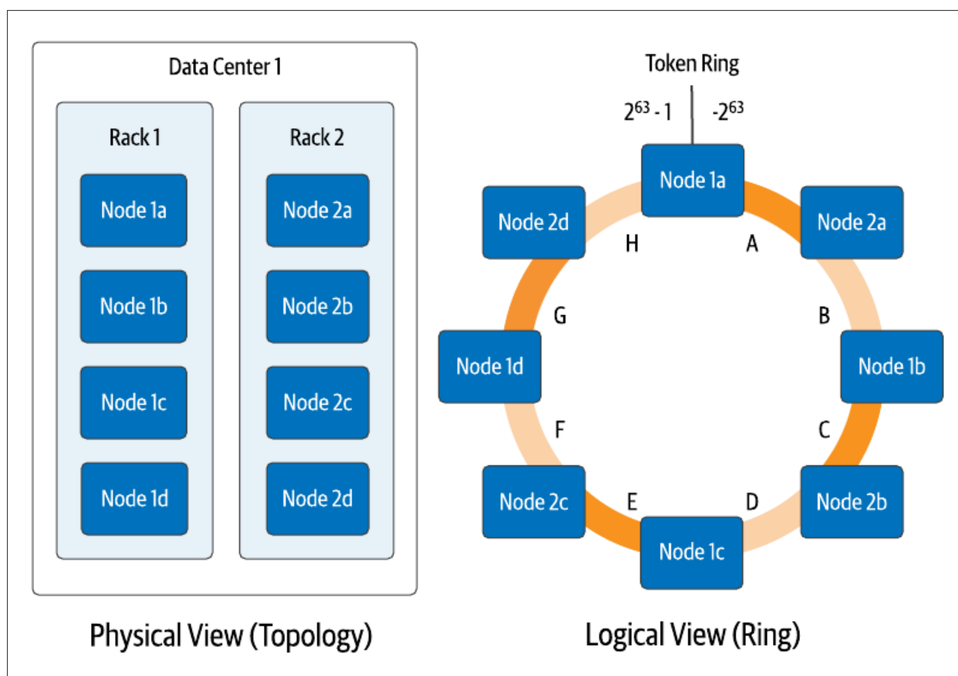


Figure 3-4. Physical and Logical Views of Cassandra's Distributed Architecture

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 −263 and 263−1. Cassandra assigns each of its nodes responsibility for one or more token ranges (shown as a single range per node labeled with letters A-H

www.dbooks.org

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 data centers and racks, and its assigned token ranges.

These requirements for identity and an association with a specific PersistentVolume-Claim 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: 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-0, cassandra-1, cassandra-2, 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 its 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. The figure also includes a seed service, which we'll discuss later in Accessing Cassandra.

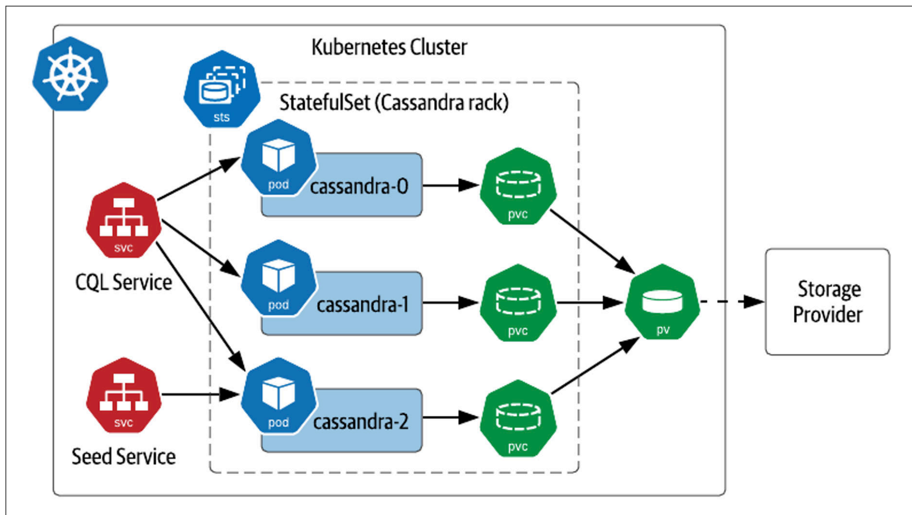


Figure 3-5. Sample Deployment of Cassandra on Kubernetes with StatefulSets

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, unlike ReplicaSets where pod name suffixes are based on UUIDs. 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.

www.dbooks.org

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 datacenter. 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. 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):

```
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:

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


www.dbooks.org

```
    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
      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 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

www.dbooks.org

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 on 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 it's 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:

```
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.

Pod Management Policies

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, 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.

www.dbooks.org

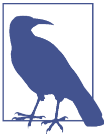
With either policy, Kubernetes manages pods according to the ordinal numbers, always adding pods with the next unused ordinal numbers when scaling up, and deleting the pods with the highest ordinal numbers when scaling down.

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 Stateful-

Set 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 affinity and anti-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.

StatefulSets: Past, Present, and Future

With Maciej Szulik, Red Hat engineer and Kubernetes SIG Apps member

The Kubernetes Special Interest Group for Applications (SIG Apps) is responsible for development of the controllers that help manage application workloads on Kubernetes. This includes the batch workloads like Jobs and CronJobs, other stateless workloads like Deployments and Replica Sets, and of course StatefulSets for stateful workloads.

The StatefulSet controller has a slightly different way of working from these other controllers. When you're thinking about Deployments, or Jobs, the controller just has to manage Pods. You don't have to worry about the underlying data, because that's either handled by persistent volumes, or are ok with just throwing each pod's data away when you destroy and recreate it. However, that behavior is not acceptable when you're trying to run a database, or any kind of workload that requires the state to be persisted between the runs. This results in significant additional complexity in the StatefulSet controller. The main challenge in writing and maturing Kubernetes controllers has been handling edge cases. StatefulSets are similar in this regard, but it's even more urgent for StatefulSets to handle the failure cases correctly, so that we don't lose data.

We've encountered some interesting use cases for StatefulSets and cases where users would like to change boundaries that have been set in the core implementation. For example, we've had pull requests submitted to change the way StatefulSets handle pods during an update. In the original implementation, the StatefulSet controllers update pods one at a time, and if something breaks during the rollout, the entire rollout is paused, and the StatefulSet requires manual intervention to make sure that data is not corrupted or lost. Some users would like the StatefulSet controller to ignore 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

www.dbooks.org

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 StatefulSets 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 was included as an Alpha feature 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 reasona-

bly 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:

```
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:

```
kubectl delete sts cassandra
```

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

```
kubectl delete -f cassandra-statefulset.yaml
```

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 you'll want to perform is to delete the CQL Service:

```
kubectl delete service cassandra
```


www.dbooks.org

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. DaemonSets support rolling updates in a similar way to StatefulSets.

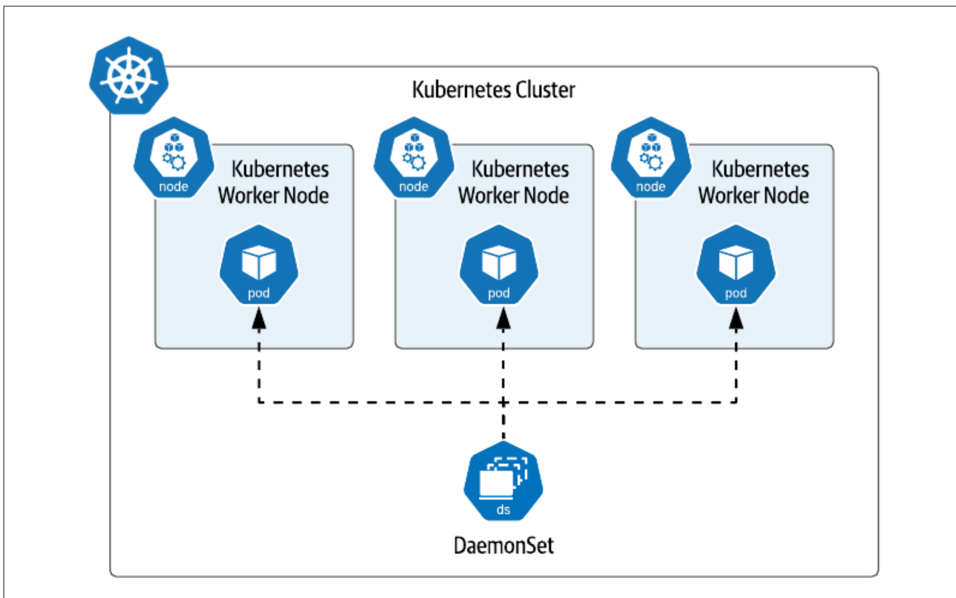


Figure 3-6. Daemon Sets run a single pod on selected worker nodes

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.

www.dbooks.org

CHAPTER 4

Automating Database Deployment on Kubernetes with Helm

> ### 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 4th chapter of the final book. The GitHub repo is 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.

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. This won't scale for large enterprises.

Second, while deploying a database is great, what about keeping it running over time? You need your 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.

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. We'll also 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 chart developers register chart and allows chart users to locate, install, and update packages on their local systems.

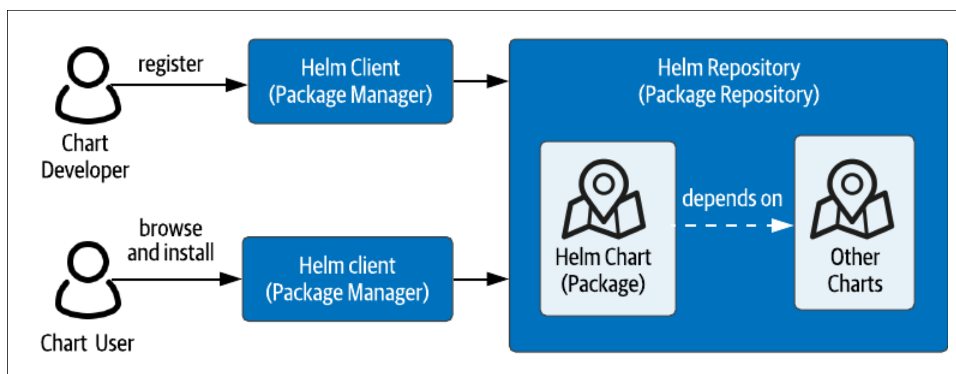


Figure 4-1. Helm, a Package Manager for Kubernetes

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

www.dbooks.org

metadata describing the binary, such as it's functionality, API, and installation instructions. In Helm, the packages are called *charts*. 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, such as Pods, Services, and PersistentVolumeClaims. For compute workloads, the descriptions point to container images 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 a 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.
- You will also need a StorageClass that supports dynamic provisioning. You may wish to follow the instructions in the StorageClasses [PROD: Add link to StorageClasses in Chapter 2.] classes section for installing a simple StorageClass and provisioner that expose local storage.

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, if it's not already on your system, you'll need to install Helm using the documentation on the Helm website. Next, add the Bitnami Helm repository:

```
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.

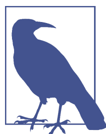
Now, let's use the Helm chart provided in the bitnami repository to deploy MySQL. In Helm's terminology, each deployment is known as a *release*. The simplest possible release that you could create using this chart would look something like this:

```
# 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):

```
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 book.

www.dbooks.org

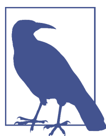
Once you've created the values.yaml file, you can start the cluster using this command:

```
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

Since we did not specify a namespace, the Helm release has been installed in the default Kubernetes namespace (unless you've separately configured a namespace in your Kubeconfig). 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):

```
helm list
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
```

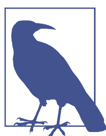
If you haven't installed the release in its own namespace, it's still simple to see the compute resources that Helm has created on your behalf by running kubectl get all, because they have all been labeled with the name of your release. It may take several minutes for all the resources to initialize, but when complete it will look something like this:

```
kubectl get all
NAME                    READY   STATUS    RESTARTS   AGE
pod/mysql-primary-0     1/1     Running   0          3h40m
pod/mysql-secondary-0   1/1     Running   0          3h40m
pod/mysql-secondary-1   1/1     Running   0          3h38m
NAME                              TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
service/mysql-primary             ClusterIP   10.96.107.156   <none>
3306/TCP   3h40m
service/mysql-primary-headless    ClusterIP   None            <none>
3306/TCP   3h40m
service/mysql-secondary           ClusterIP   10.96.250.52    <none>
3306/TCP   3h40m
service/mysql-secondary-headless  ClusterIP   None            <none>
3306/TCP   3h40m
NAME                              READY   AGE
statefulset.apps/mysql-primary    1/1     3h40m
statefulset.apps/mysql-secondary  2/2     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.



Namespaces and Kubernetes Resource Scope

If you have chosen to install your Helm release in a namespace, you'll need to specify the namespace on most of your kubectl get commands in order to see the created resources. The exception is kubectl get pv, because PersistentVolumes are one of the Kubernetes resources that are not namespaced, that is, they can be used by pods in any namespace. To learn more about which Kubernetes resources in your cluster are namespaced and which are not, run the command kubectl api-resources.

www.dbooks.org

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.

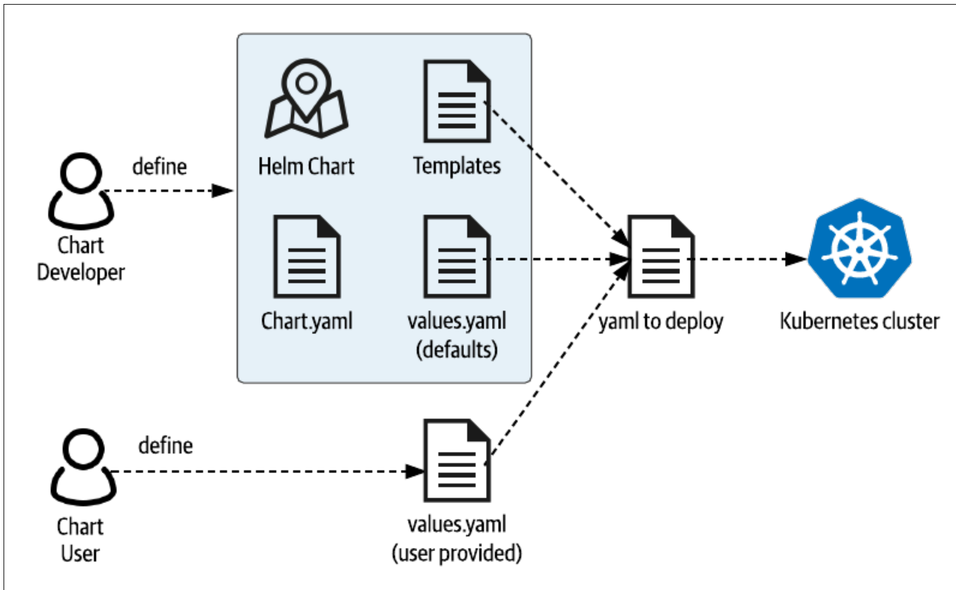


Figure 4-2. Customizing a Helm release using a values.yaml file

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 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 uses the template to generate yaml configuration code, 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:

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

Running this command will produce quite a bit of output, so you may want to redirect 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.

www.dbooks.org

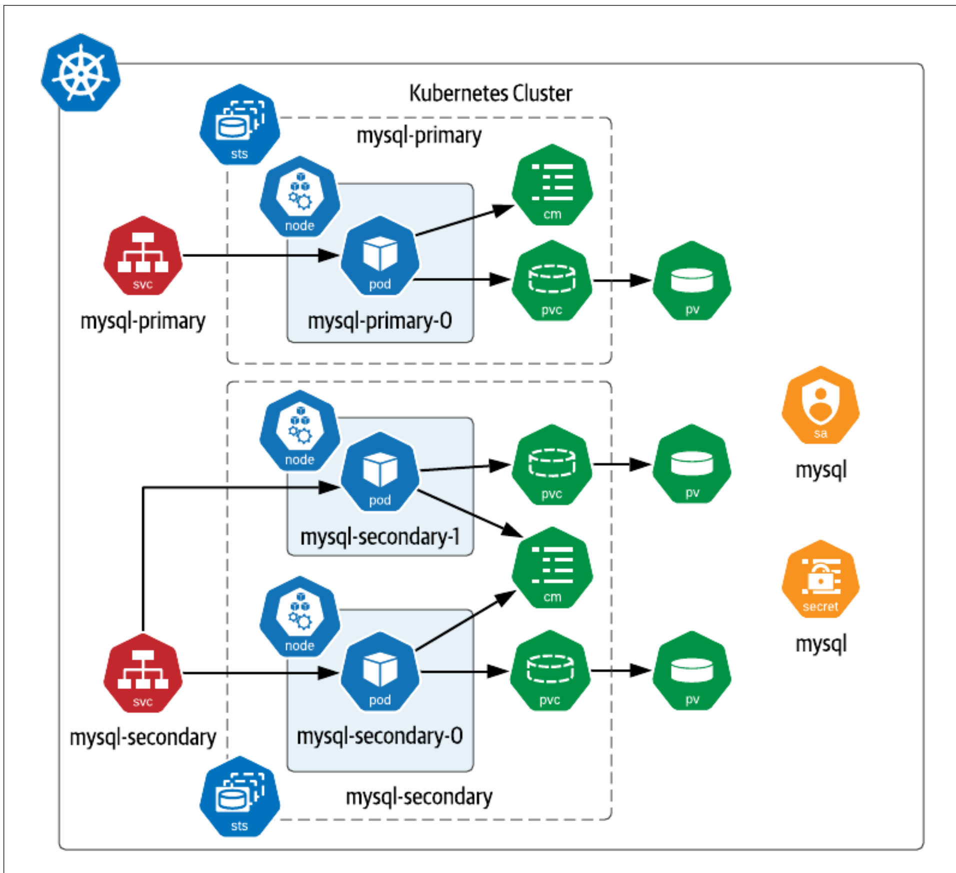


Figure 4-3. Deploying MySQL using the Bitnami Helm Chart

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:

```
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 to it. 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:

```
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. 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 give this ServiceAccount access it needs, the MySQL Helm chart creates a Role specifying the Kubernetes resources and operations, and a RoleBinding to associate the Service-Account to the Role. We'll discuss ServiceAccounts and role-based access in Chapter 5.

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:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql
  namespace: default
```


www.dbooks.org

```
  labels: ...
type: Opaque
data:
  mysql-root-password: "VzhyNEhIcmdTTQ=="
  mysql-password: "R2ZtNkFHNDhpOQ=="
  mysql-replication-password: "bDBiTWVzVmVORA=="
```

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:

```
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):

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-primary
  namespace: default
  labels: ...
spec:
  replicas: 1
```

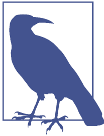
```
  selector:
    matchLabels: ...
  serviceName: mysql-primary
  template:
    metadata:
      annotations: ...
      labels: ...
    spec:
      ...
      serviceAccountName: mysql
      containers:
        - name: mysql
          image: docker.io/bitnami/mysql:8.0.26-debian-10-r60
          volumeMounts:
            - name: data
              mountPath: /bitnami/mysql
            - name: config
              mountPath: /opt/bitnami/mysql/conf/my.cnf
              subPath: my.cnf
      volumes:
        - name: config
          configMap:
            name: mysql-primary
```

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

www.dbooks.org



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:

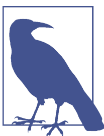
- A new version of a chart is available
- A new version of an image used by your application is available
- You want to change the selected options

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 it's 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 configurations 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

www.dbooks.org

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 set 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:

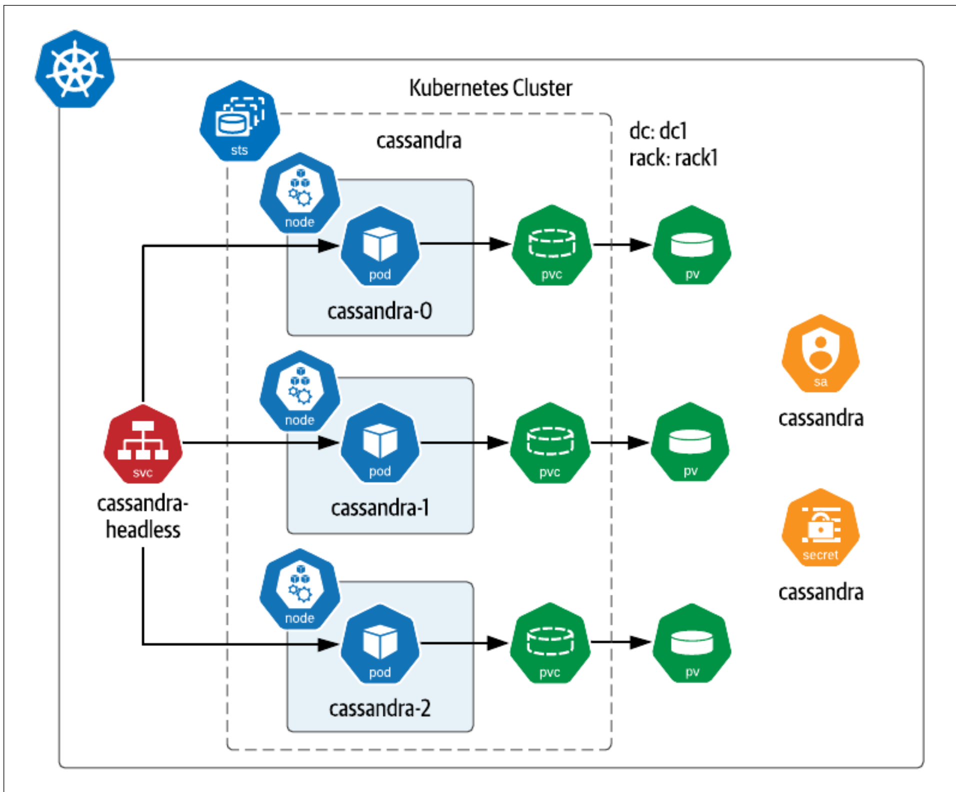


Figure 4-4. Deploying Apache Cassandra using the Bitnami Helm Chart

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.

www.dbooks.org

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:

```
kubectl 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:

```
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 the preference that a Pod is scheduled onto a node where another specific Pod is running. For example, pod affinity could be used to co-locate 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.

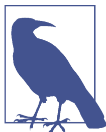
Each of these types of affinity can be expressed as either hard or soft constraints, known as

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 IgnoredDuringExcecution 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 requiredDuringSchedulingRequiredDuringExecution. 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

www.dbooks.org

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. We'll also look at how Kubernetes allows you to plug in different schedulers in Chapter 9.

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 repo remove bitnami
```

Helm Limitations

While Helm is a powerful tool for deploying 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 opera-

tional 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 functions. 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 was to implement custom Kubernetes jobs labeled as pre-upgrade hooks that Helm would execute on an

www.dbooks.org

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 database 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.

www.dbooks.org

CHAPTER 5

Automating Database Management on Kubernetes with Operators

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 5th chapter of the final book. The GitHub repo is 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.

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 such as the API Server, Scheduler, Kubelet and kubectl, along with indications of the extension points they support.

www.dbooks.org

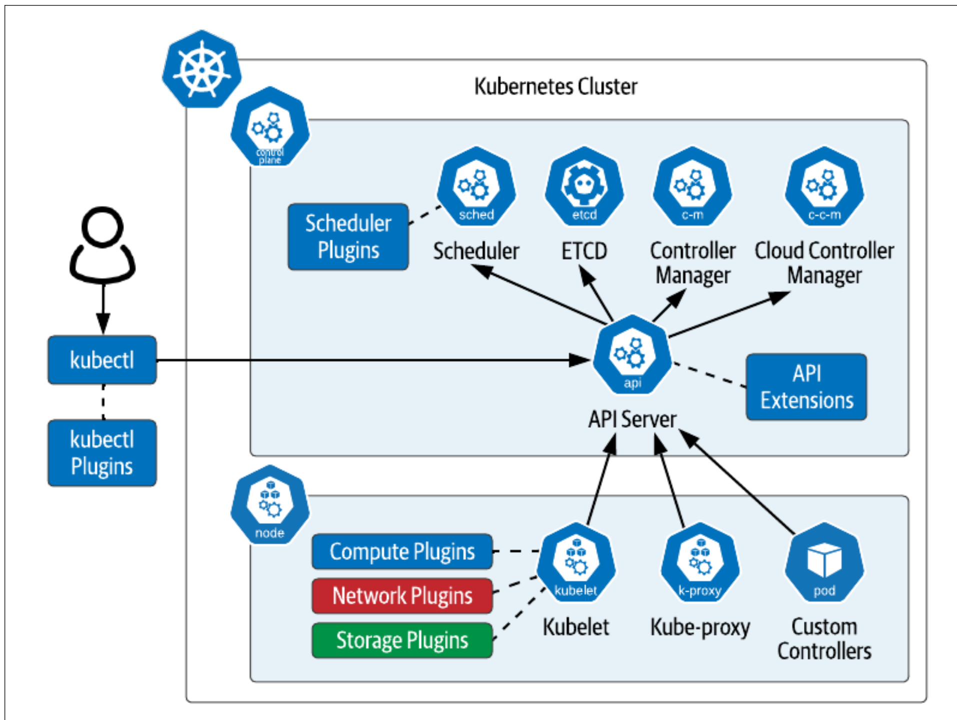


Figure 5-1. Kubernetes Control Plane and Extension Points

Now let's examine the details of extending the Kubernetes control plane, starting with components on your local client and those within the Kubernetes cluster. Many of these extension points are relevant to databases and data infrastructure.

Extending Kubernetes Clients

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 Control Plane Components

The core of the Kubernetes control plane consists of several control plane components including the API Server, Scheduler, Controller Manager, Cloud Controller Manager, and etcd. While these components can be run on any node within a Kuber-

netes cluster, they are typically assigned to a dedicated node which does not run any user application pods.

API Server
: The API Server is the primary interface for external and internal clients of a Kubernetes cluster. It exposes RESTful 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 resource definitions (CRDs) 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 9.

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 Con-

www.dbooks.org

troller 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 every node in the cluster. These worker node components include the Kubelet, Kube-Proxy, and container runtime.

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 Kubelet uses the container runtime to execute containers on the worker's operating system. Supported container runtimes for Linux include containerd and CRI-O. Docker runtime support was deprecated in Kubernetes 1.20 and removed entirely in 1.24.

Custom controllers and operators
: These controllers are responsible for managing applications installed on a Kubernetes cluster using custom resources. Although these controllers are extensions to the Kubernetes control plane, they can run on any worker 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.

www.dbooks.org

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.

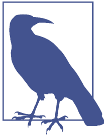
For example, consider a controller managing a distributed database with multiple replicas. When the database controller receives a notification that the desired number of replicas has increased, the controller could scale an underlying Deployment or StatefulSet that it is using to manage replicas. Later, 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 or other applications. 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 CrashLoopBackOff 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:

www.dbooks.org

```
$ kubectl api-resources
NAME                SHORTNAMES  APIVERSION  NAMESPACED  KIND
bindings                        v1          true        Binding
componentstatuses   cs          v1          false       ComponentStatus
configmaps          cm          v1          true        ConfigMap
...
```

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
etcdlockservers.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
metadata:
  annotations:
```

```
      controller-gen.kubebuilder.io/version: v0.3.0
    creationTimestamp: null
    name: vitesskeyspaces.planetscale.com
  spec:
    group: planetscale.com
    names:
      kind: VitessKeyspace
      listKind: VitessKeyspaceList
      plural: vitesskeyspaces
      shortNames:
      - vtk
      singular: vitesskeyspace
    scope: Namespaced
    subresources:
      status: {}
    validation:
      openAPIV3Schema:
        properties:
          ...
```

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.

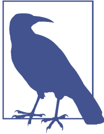
Operators

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 pro-

www.dbooks.org

vide 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.

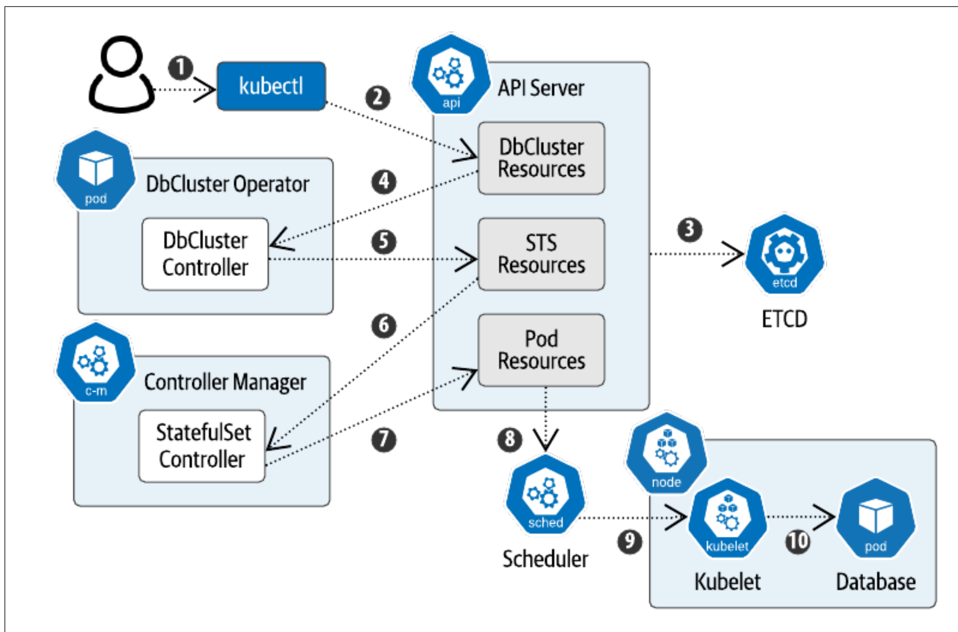


Figure 5-2. Interaction between Kubernetes controllers and operators

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 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 to the appropriate instances using a sharding approach. 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, and primary and replica tablets.

www.dbooks.org

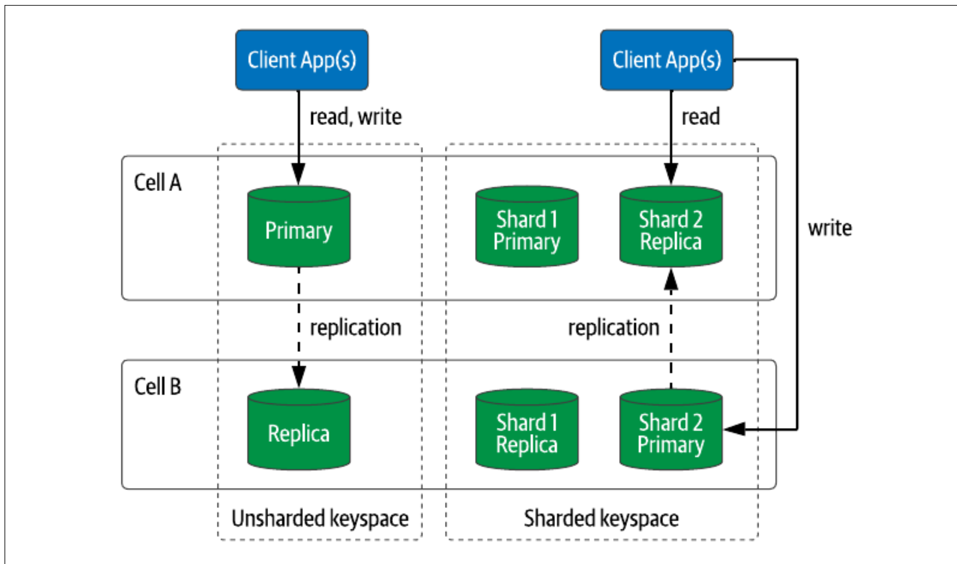


Figure 5-3. Vitess cluster topology - cells, keyspaces, and shards

At a high level, a Vitess cluster consists of multiple MySQL instances called tablets which may be spread across multiple data centers or *cells*. Each MySQL instance takes on a role as either a primary 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 primary 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 primary and replicated to the replica nodes in the background. Reads can be served from the primary 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 primary instances. On reads, Vitess gathers data from primary 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.

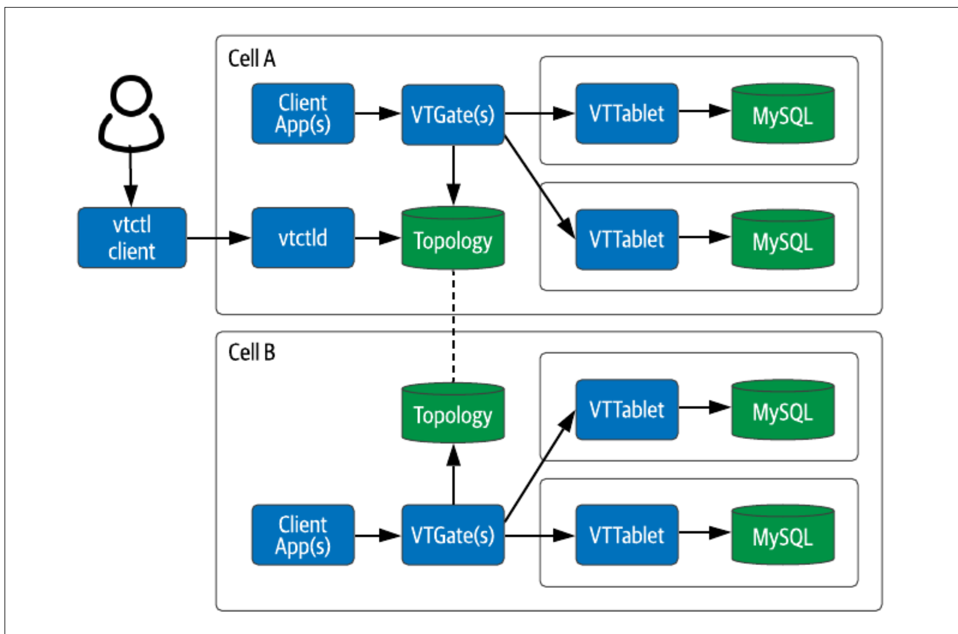


Figure 5-4. Vitess architecture including VTGate, VTTablets, Topology Service

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

www.dbooks.org

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 primary for a shard, or one of its replicas. There are two types of replica, those that can be promoted to replace a primary, 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 VTTablets exist for each shard, and which VTTablet is the primary. 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.

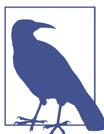
Vtctld and Vtctlclient
: The Vitess control daemon vtctld and its client vtctlclient provide the control plane used to configure and manage Vitess clusters. (There is also a command-line version called vtctl that combines the client and server as a single executable,

but it is not frequently used in cloud deployments for security reasons.) Vtctld 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. Vtctld 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. 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 its 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.

www.dbooks.org

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.

```
kubectl apply -f https://raw.githubusercontent.com/vitessio/vitess/main/exam-
ples/operator/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 which 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.

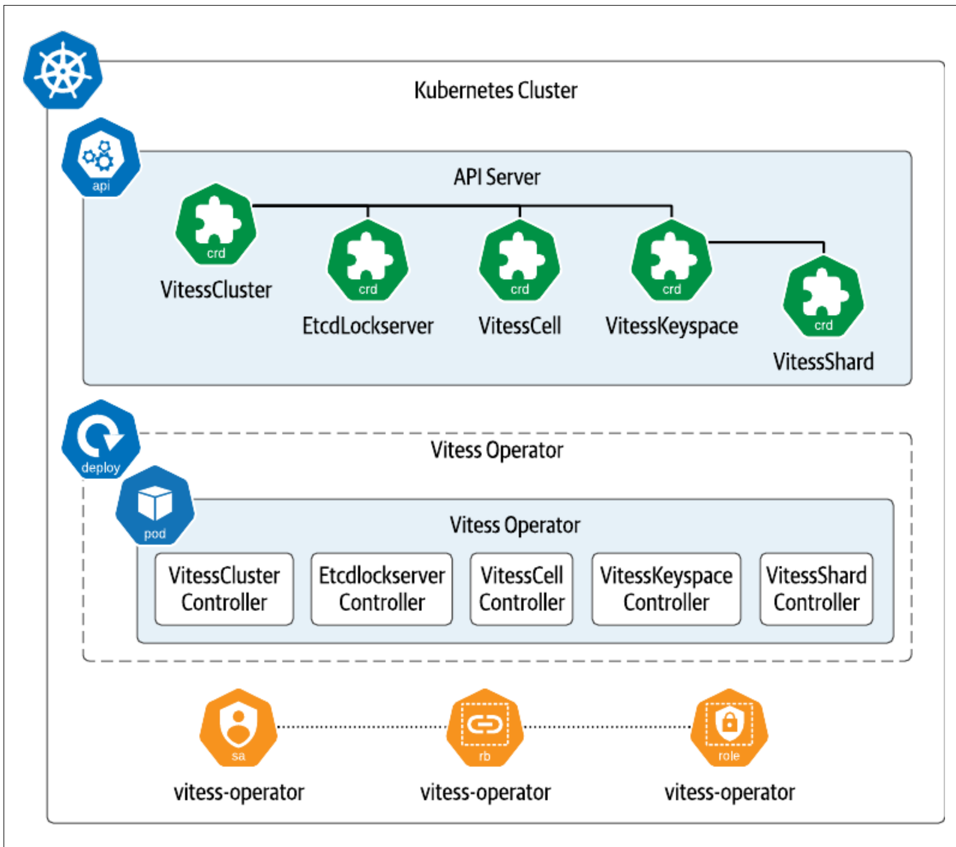


Figure 5-5. Vitess Operator and Custom Resource Definitions

Roles and RoleBindings. As shown toward the bottom of Figure 5-5, 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, examine the configuration of the vitess-operator Role from the file that you used to install the operator (you can search for "kind: Role" to locate the pertinent code):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: vitess-operator
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - services
```


www.dbooks.org

```
  - endpoints
  - persistentvolumeclaims
  - events
  - configmaps
  - secrets
  verbs:
  - '*'
...
```

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 *. If you follow the URL above and examine more of the code, you'll also see how the Role is also given access to other resources, including Deployments and ReplicaSets, and 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.

```
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.

www.dbooks.org

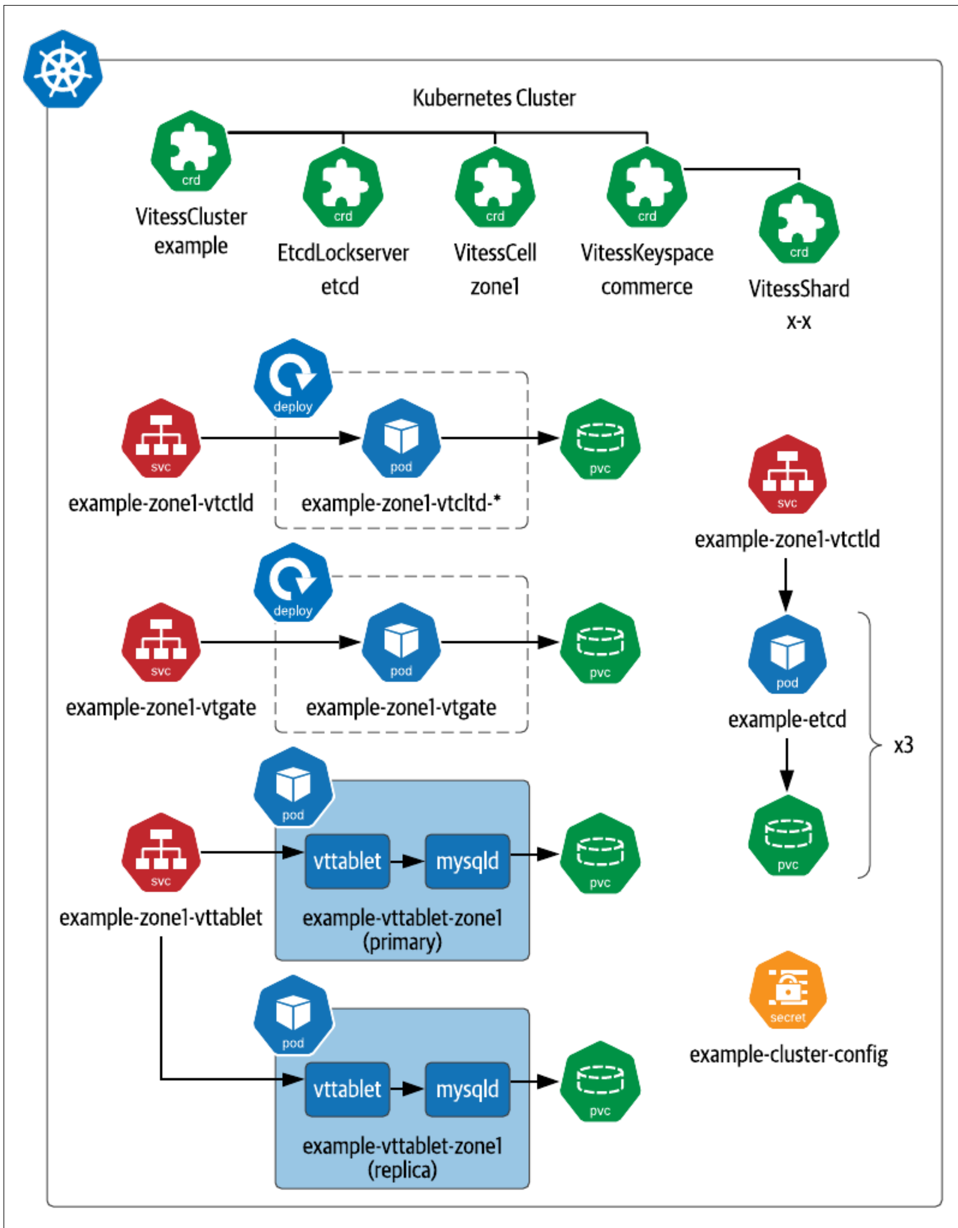


Figure 5-6. Resources managed by the VitessCluster example

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:

```
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.

```
  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.

```
  vitessDashboard:
    cells:
    - zone1
    extraFlags:
      security_policy: read-only
    replicas: 1
    resources:
      ...
```

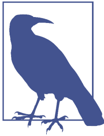
The VitessCluster also defines the commerce keyspace, which contains a single shard (essentially, an unsharded keyspace). This single shard has a pool of two VTTablets in the cell zone1, each of which will be allocated 10GB of storage.

www.dbooks.org

```
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.



Note: Visualizing larger Kubernetes applications

While its 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, or K9s, which provides a command line interface.

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. When you add additional replicas to a VitessCluster, the operator initializes their data by performing a restore from the most recent backup.

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 VSchema 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 applica-

www.dbooks.org

tion 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 the team at YouTube decided to make Vitess run on Kubernetes, they were able to do the work without a lot of changes.

Before Vitess was donated to the Cloud Native Computing Foundation (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 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 maintaining an operator requires keeping 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 will greatly improve reliability.

Another area of improvement we are focused on 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 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.

One final area of improvement is data locality. 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. 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. Let's continue to champion the benefits of working on open source software in our organizations so we can continue to grow as a community.

A Growing Ecosystem of Operators

The operator pattern has become quite popular in the Kubernetes community, aided in part by 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

www.dbooks.org

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, the Operator Hub, you should consider as your first stop when looking for an operator. 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, 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.

Capability Level	Characteristics	Database Operator Examples	Tools
Level 1 - Basic Install	Installation and configuration of Kubernetes and workloads	The operator uses custom resources to provide a central point of configuration for a database cluster. The operator deploys the database by creating resources such as Deployments, ServiceAccounts, RoleBindings, PersistentVolumeClaims and Secrets, and helps initialize the database schema.	Helm, Ansible, Go
Level 2 - Seamless Upgrades	Upgrade of the managed workload and operator	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.	

Level 3 - Full Lifecycle	Ability to create backups and restore from backups, ability to failover or replace portions of a clustered application, ability to scale the application	The operator provides a way to create a consistent backup across multiple data nodes and the ability to use those backups to restore or replace failed database nodes. The operator can respond to a configuration change to add or remove database nodes or perhaps even data centers.	Ansible, Go
Level 4 - Deep Insights	Providing capabilities including alerting, monitoring, events, or metering	The operator monitors metrics and logging output by the database software and uses this information to implement health and readiness checks. The operator pushes metrics and alerts to other infrastructure.	
Level 5 - Auto-pilot	Providing capabilities including auto-scaling, auto-healing, auto-tuning	The operator auto-scales the number of database nodes in the cluster up or down to meet performance requirements. The operator might also dynamically resize PVs or change the storage class used for various database nodes. The operator automatically performs database maintenance such as rebuilding indexes to improve slow response times. The operator detects abnormal workload patterns and takes action such as resharding to balance workloads.	

Table 5-1: Operator Capability Levels applied to databases

These levels are useful both for evaluating operators you might want to use, 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:

DataStax Kubernetes Operator for Apache Cassandra
: In 2021, several companies in the Cassandra community who had developed their own operators came together in support of an operator built by DataStax, known primarily by its nickname: 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 [PROD: Add link to Chapter 7.].

www.dbooks.org

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, Pure Storage

www.dbooks.org

As discussed in this chapter, the number of Kubernetes operators for databases has been continuously growing. Database developers want their databases to run on Kubernetes, so to their credit, projects like Vitess are stepping up and developing operators to make it easy for others. This initiative is great, but one potential drawback of this approach 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 those who are developing operators have to reckon with is how specialized to 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 in addition to a graph database or a key-value store. This forces cluster administrators to provide different types of databases while maintaining the operational simplicity of a single platform.

No Kubernetes admin wants to maintain 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 multiple operator lifecycles, you've already lost. Multiply that 10 or 15 times over, and you are completely at odds with the value Kubernetes provides. 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"? This might be possible, but 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 TopologyAwareStatefulSet as a new CRD, or a ClusterMembership 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 there's an exciting future for 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.

As you can see, the state of the art in Kubernetes operators is continuing to mature. Whether the goal is to build a unified operator or just to make it easier to build database-specific operators, it's clear that great progress can be made as multiple communities begin to collaborate on common CRDs to address problems like cluster membership, topology awareness, and leader election.

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 expand our focus to consider how those databases interact with other infrastructure.

www.dbooks.org

CHAPTER 6
Integrating Data Infrastructure in a Kubernetes Stack

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. The GitHub repo is 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.

In this book we are illuminating a future of modern, cloud native applications that run on Kubernetes. Up to this point of the book, we've noted that historically, data has been one of the hardest parts of making this a reality. In previous chapters, we've introduced the primitives Kubernetes provides for managing compute, network, and storage resources, and considered how databases can be deployed on Kubernetes using these resources. We've also examined the automation of infrastructure using controllers and the operator pattern.

Now it's time to expand our focus to consider how data infrastructure fits in your overall application architecture in Kubernetes. In this chapter we'll explore how to assemble the building blocks discussed in previous chapters into integrated data infrastructure stacks that are easy to deploy and tailor to the unique needs of each application. These stacks represent a step toward the vision of the virtual data center we introduced in Chapter 1. To learn the considerations involved in building and

using these larger assemblies, let's take an in-depth look at K8ssandra, an open-source project that provides an integrated data stack based on Apache Cassandra.

K8ssandra: Production-ready Cassandra on Kubernetes

To set the context, let's consider some of the practical challenges of moving application workloads into Kubernetes. As organizations have begun to migrate existing applications to Kubernetes and create new cloud native applications in Kubernetes, modernizing the data tier is a step that is often deferred. Whatever the causes of these delays - a belief that Kubernetes is not ready for stateful workloads, a lack of development resources, or other factors - the result has been mismatched architectures in which applications are running in Kubernetes with databases and other data infrastructure running externally. This leads to a division of focus for developers and SREs that can limit productivity. It's also common to see distinct toolsets for monitoring applications and database infrastructure, which increases cloud computing costs.

This adoption challenge became evident in the Cassandra community. Despite the growing collaboration and consensus around building a single Cassandra operator as discussed in Chapter 5, developers were still confronted with some key questions about how the database and operator would fit in the larger application context:

- How can we have an integrated view of the health of our entire stack, including both applications and data?
- How can we tailor the automation of installation, upgrades, and other operational tasks in a Kubernetes-native way that fits the way we manage our data centers?

To help address these questions, John Sanda and a team of engineers at DataStax launched an open-source project called K8ssandra with the goal of providing a production-ready deployment of Cassandra that embodies best practices for running Cassandra in Kubernetes. K8ssandra provides custom resources that help manage tasks including cluster deployment, upgrades, scaling up and down, data backup and restore, and more. You can read more about the motivations for the project in the article Why K8ssandra?

K8ssandra Architecture

K8ssandra is deployed in units known as clusters, which is similar terminology to that used by Kubernetes and Cassandra. A K8ssandra cluster includes a Cassandra cluster along with additional components depicted in Figure 6-1 to provide a full data management ecosystem. Let's consider these in roughly clockwise order starting from the top center.

www.dbooks.org

- Cass Operator is a Kubernetes operator first introduced in Chapter 5. It manages the lifecycle of Cassandra nodes on Kubernetes, including provisioning new nodes and storage, and scaling up and down.
- Cassandra Reaper manages the details of repairing Cassandra nodes in order to maintain high data consistency.
- Cassandra Medusa provides backup and restore for data stored in Cassandra
- Prometheus and Grafana are used for the collection and visualization of metrics
- Stargate is a data gateway that provides API access to client applications as an alternative to CQL.
- K8ssandra Operator orchestrates all of the other components, including multi-cluster support for managing Cassandra clusters that span multiple Kubernetes clusters.

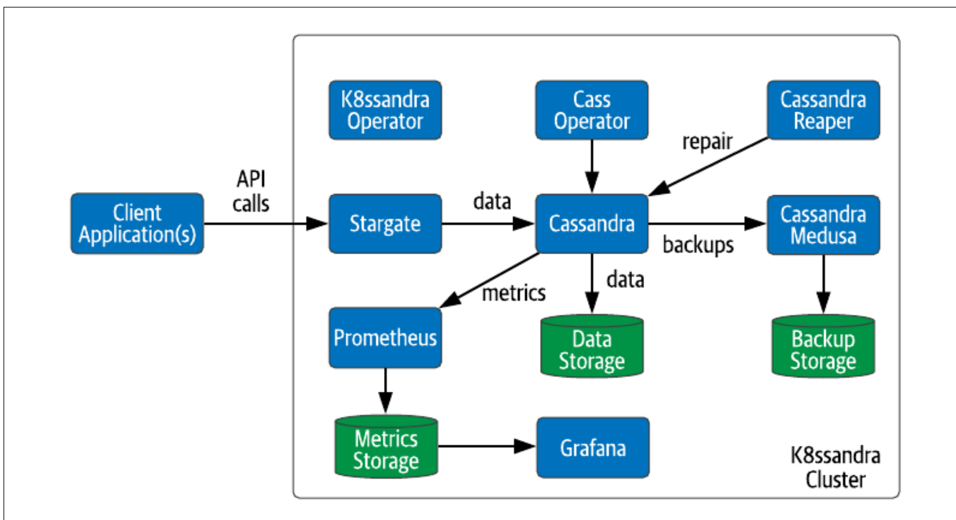


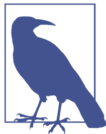
Figure 6-1. K8ssandra Architecture

In the following sections, we'll take a look at each component of the K8ssandra project to understand the role that it plays within the architecture and its relationship to other components.

Installing the K8ssandra Operator

Let's dive in with some hands-on experience of installing K8ssandra. To get a basic installation of K8ssandra running that fully demonstrates the power of the operator, you'll need a Kubernetes cluster with several worker nodes. To make the deployment simpler, the K8ssandra team has provided scripts to automate the process of creating

Kubernetes clusters and then deploying the operator to these clusters. These scripts use Kind clusters for simplicity, so you'll want to make sure you have this installed before starting. Instructions for installing on various clouds are also available on the K8ssandra website.



K8ssandra 2.0 Release Status

This chapter focuses on the K8ssandra 2.0 release, including the K8ssandra Operator. At the time of writing, K8ssandra 2.0 is still in Beta status. The instructions we provide here are based on an installation guide in the K8ssandra Operator repository. As K8ssandra 2.0 moves toward a full general availability (GA) release, the instructions on the Getting Started section of the K8ssandra website will be updated to reference the new version.

First, start by cloning the K8ssandra operator repository from GitHub:

```
git clone https://github.com/k8ssandra/k8ssandra-operator.git
```

Next, you'll want to use the provided Makefile to create a Kubernetes cluster and deploy the K8ssandra Operator into it. (This assumes you have make installed)

```
cd k8ssandra-operator
make single-up
```

If you examine the Makefile, you'll notice the operator is installed using Kustomize, which we discussed in Chapter 4. The target you just executed creates a Kind cluster with four worker nodes and changes your current context to point to that cluster, as you can see by running the following:

```
% kubectl config current-context
kind-k8ssandra-0
% kubectl get nodes
NAME                        STATUS   ROLES                  AGE     VERSION
k8ssandra-0-control-plane   Ready    control-plane,master   6m45s   v1.22.4
k8ssandra-0-worker          Ready    <none>                 6m13s   v1.22.4
k8ssandra-0-worker2         Ready    <none>                 6m13s   v1.22.4
k8ssandra-0-worker3         Ready    <none>                 6m13s   v1.22.4
k8ssandra-0-worker4         Ready    <none>                 6m13s   v1.22.4
```

Now examine the list of CustomResourceDefinitions (CRDs) that have been created as follows:

```
% kubectl get crd
NAME                                          CREATED AT
cassandrabackups.medusa.k8ssandra.io          2022-02-05T17:31:35Z
cassandradatacenters.cassandra.datastax.com   2022-02-05T17:31:35Z
cassandrarestores.medusa.k8ssandra.io         2022-02-05T17:31:35Z
cassandratasks.control.k8ssandra.io           2022-02-05T17:31:36Z
certificaterequests.cert-manager.io           2022-02-05T17:31:16Z
certificates.cert-manager.io                  2022-02-05T17:31:16Z
```


www.dbooks.org

```
challenges.acme.cert-manager.io             2022-02-05T17:31:16Z
clientconfigs.config.k8ssandra.io           2022-02-05T17:31:36Z
clusterissuers.cert-manager.io              2022-02-05T17:31:17Z
issuers.cert-manager.io                     2022-02-05T17:31:17Z
k8ssandraclusters.k8ssandra.io              2022-02-05T17:31:36Z
orders.acme.cert-manager.io                 2022-02-05T17:31:17Z
reapers.reaper.k8ssandra.io                 2022-02-05T17:31:36Z
replicatedsecrets.replication.k8ssandra.io  2022-02-05T17:31:36Z
stargates.stargate.k8ssandra.io             2022-02-05T17:31:36Z
```

As you can see, there are several CRDs associated with the Cert Manager and K8ssandra. There is also the cassandradatacenter CRD used by Cass Operator. The K8ssandra and Cass Operator CRDs are all namespaced, which you can verify using the kubectl api-resources command, meaning that resources created according to these definitions are assigned to a specific namespace. That command will also show you the acceptable abbreviations for each resource type, for example k8c for k8ssandracluster.

Next, you can examine the contents that have been installed within the Kind cluster. If you list the namespaces using kubectl get ns, you'll note two new namespaces: cert-manager and k8ssandra-operator. As you may suspect, K8ssandra is using the same Cert Manager project as Pulsar, as described in Chapter 8. Let's examine the contents of the k8ssandra-operator namespace, which are summarized in Figure 6-2 along with related K8ssandra CRDs. If you examine the workloads, you'll notice that two Deployments have been created, one for the K8ssandra Operator, and one for Cass Operator.

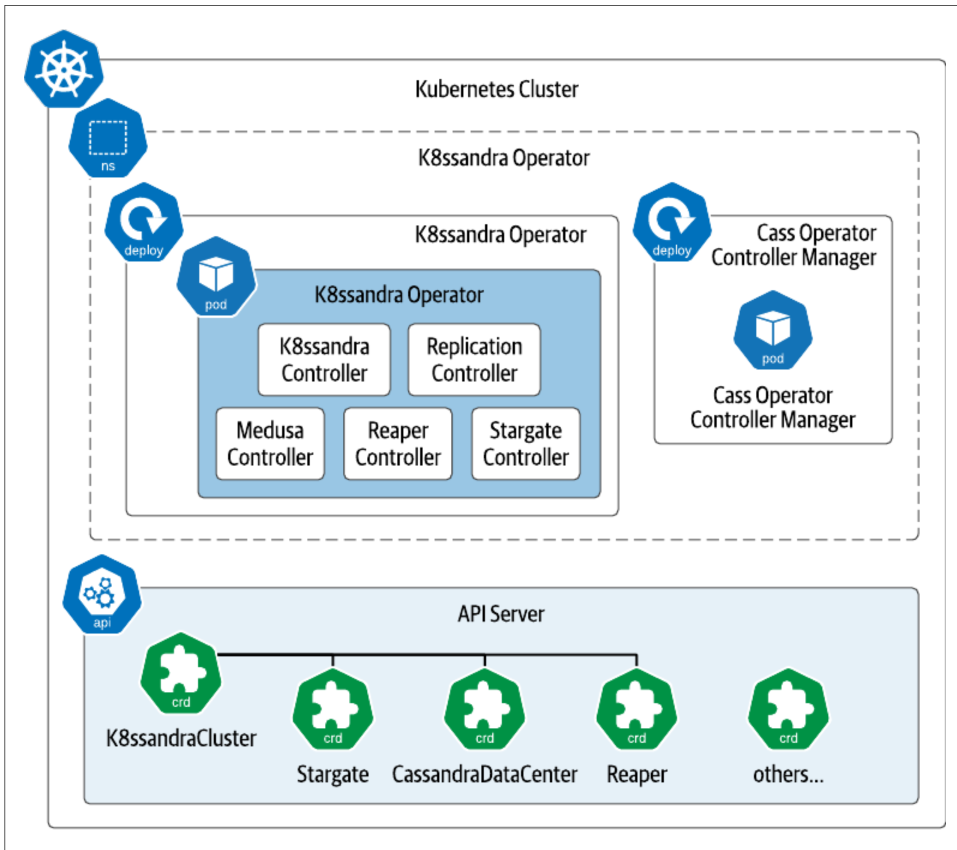


Figure 6-2. K8ssandra Operator Architecture

If you examine the K8ssandra Operator source code, you'll see that it actually contains multiple controllers, while the Cass Operator consists of a single controller. This packaging reflects the fact that Cass Operator is an independent project which can be used by itself without having to adopt the entire K8ssandra framework, otherwise it could have been included as a controller within the K8ssandra Operator. Table 7-1 describes the mapping of these various controllers to the key resources with which they interact.

Operator	Controller	Key Custom Resources
K8ssandra Operator	K8ssandra Controller	K8ssandraCluster, CassandraDataCenter
	Medusa Controller	CassandraBackup, CassandraRestore
	Reaper Controller	Reaper
	Replication Controller	ClientConfig, ReplicatedSecret
	Stargate Controller	Stargate
Cass Operator	Cass Operator Controller Manager	CassandraDataCenter

www.dbooks.org

Table 7-1: Mapping K8ssandra CRDs to Controllers

We'll introduce each K8ssandra and Cass Operator CRD in more detail in the sections below.

Creating a K8ssandraCluster

Once you've installed the K8ssandra Operator, the next step is to create a K8ssandraCluster. The source code used in this section is available at Vitess Operator Example, based on samples available in the K8ssandra Operator GitHub repo. First, have a look at the k8ssandra-cluster.yaml file:

```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: demo
spec:
  cassandra:
    cluster: demo
    serverVersion: "4.0.1"
    datacenters:
      - metadata:
          name: dc1
        size: 3
        storageConfig:
          cassandraDataVolumeClaimSpec:
            storageClassName: standard
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 1Gi
        config:
          jvmOptions:
            heapSize: 512M
        stargate:
          size: 1
          heapSize: 256M
```

This code specifies a K8ssandraCluster resource consisting of a single datacenter dc1 running three nodes of Cassandra 4.0.1, where the Pod specification for each Cassandra node requests 1GB of storage using a PersistentVolumeClaim that references the standard StorageClass. This configuration also includes a single Stargate node to provide API access to the Cassandra cluster. This is a minimal configuration that accepts the chart defaults for most of the other components. Create the demo K8ssandraCluster in the k8ssandra-operator namespace with the command:

```
% kubectl apply -f k8ssandra-cluster.yaml -n k8ssandra-operator
k8ssandracluster.k8ssandra.io/demo created
```

Once the command completes, you can check on the installation of the K8ssandraCluster using commands such as kubectl get k8ssandraclusters (or kubectl get k8c for short). Figure 6-3 depicts some of the key compute, network, and storage resources that the operator built on your behalf when you created the demo K8ssandraCluster.

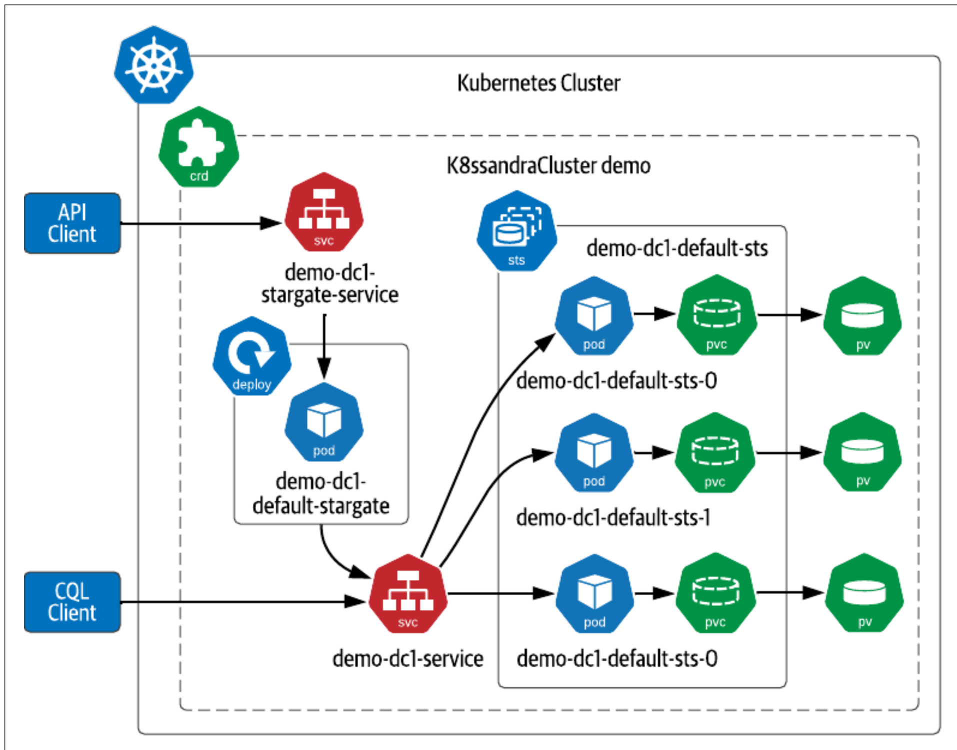


Figure 6-3. A simple K8ssandraCluster

Here are some key items to note:

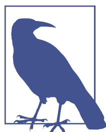
- A single StatefulSet has been created to represent the Cassandra datacenter dc1, with three pods containing the replicas you specified. As you'll learn in Managing Cassandra in Kubernetes with Cass Operator, K8ssandra actually uses a CassandraDatacenter CRD to manage this StatefulSet via the Cass Operator.
- While the figure shows a single Service demo-dc1-service exposing access to the Cassandra cluster as a single endpoint, this is a simplification. You will actually find there are multiple Services configured to provide access for various clients.
- There is a Deployment managing a single Stargate Pod, as well as Services that provide client endpoints to the various API services provided by Stargate. This is

www.dbooks.org

another simplification, and we'll explore this part of configuration in more detail in Stargate.

- Similar to examples of infrastructure we've shown in previous chapters, the K8ssandra Operator also creates additional supporting security resources such as ServiceAccounts, Roles, and RoleBindings.

Once you have a K8ssandraCluster created, you can point client applications at the Cassandra interfaces and Stargate APIs, and perform cluster maintenance operations. You can remove a K8ssandraCluster just by deleting its resource, but you won't want to do that yet as we have a lot more to explore! We'll describe several of these interactions as we examine each of the K8ssandra components in more detail. Along the way, we'll make sure to note some of the interesting design choices made by contributors to K8ssandra and related projects in terms of how they use Kubernetes resources and how they adapt data infrastructure that predates Kubernetes into the Kubernetes way of doing things.



StackGres: An integrated Kubernetes stack for Postgres

The K8ssandra project is not the only instance of an integrated data stack that runs on Kubernetes. Another example can be found in StackGres, a project managed by OnGres. StackGres uses Patroni to support clustered, highly available Postgres deployments and adds automated backup functionality. StackGres supports integration with Prometheus and Grafana for metrics aggregation and visualization, along with an optional Envoy proxy for getting more fine grained metrics at the protocol level. StackGres is composed of open-source components and uses the AGPLv3 license for its community edition.

Managing Cassandra in Kubernetes with Cass Operator

Cass Operator is the shorthand name for the DataStax Kubernetes Operator for Apache Cassandra. Cass Operator is an open source project available on GitHub that was brought under the umbrella of the K8ssandra project in 2021, replacing its previous home under the DataStax GitHub organization.

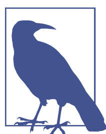
Cass Operator is a key part of K8ssandra, since a Cassandra cluster is the basic data infrastructure around which all the other infrastructure elements and tools are added. However, Cass Operator was developed before K8ssandra and will continue to exist as a separately deployable project. This is helpful since not every capability of Cass Operator is exposed via K8ssandra, especially more advanced Cassandra configuration options. Cass Operator is listed as its own project in Operator Hub and can be installed via Kustomize.

Cass Operator provides a mapping of Cassandra's topology concepts including clusters, datacenters, racks, and nodes onto Kubernetes resources. The key construct is the CassandraDataCenter CRD, which represents a datacenter within the topology of a Cassandra cluster. (Reference Chapter 3 if you need a refresher on Cassandra topology.)

When you created a K8ssandraCluster resource in the previous section, the K8ssandra Operator created a single CassandraDatacenter resource, which would have looked something like this:

```
apiVersion: cassandra.datastax.com/v1beta1
kind: CassandraDatacenter
metadata:
  name: dc1
spec:
  clusterName: demo
  serverType: cassandra
  serverVersion: 4.0.1
  size: 3
  racks:
  - name: default
```

Since you didn't specify a rack in the K8ssandraCluster definition, K8ssandra interprets this as a single rack named default. By creating the CassandraDatacenter, K8ssandra Operator is delegating the operation of the Cassandra nodes in this datacenter to Cass Operator.



Cass Operator and Multiple Datacenters

You may be wondering why Cass Operator does not define a CRD representing a Cassandra cluster. From the perspective of the Cass Operator, the concept of the Cassandra cluster is basically just a piece of metadata - the CassandraDataCenter's clusterName - rather than an actual resource. This reflects the convention that Cassandra clusters used in production systems are typically deployed across multiple physical data centers, which is beyond the scope of a Kubernetes cluster.

While you can certainly create multiple CassandraDatacenters and link them together using the same clusterName, they must be in the same Kuberneters cluster for Cass Operator to be able to manage them. It's also recommended to use a separate namespace to install a dedicated instance of Cass Operator to manage each cluster. You'll see how K8ssandra supports the ability to create Cassandra clusters that span multiple physical datacenters (and Kubernetes clusters) in Multi-cluster topologies.

When Cass Operator is notified by the API Server of the creation of the CassandraDatacenter resource, it creates resources used to implement the datacenter, including a StatefulSet to manage the nodes in each rack, as well as various Services and

www.dbooks.org

security-related resources. The StatefulSet will start the requested number of Pods in parallel. This brings up a situation in which Cass Operator provides logic to adapt between how Cassandra and Kubernetes operate.

If you have worked with Cassandra previously, you may be aware that the best practice for adding nodes to a cluster is to do so one one at a time, to simplify the process of a node joining the cluster. This process, called *bootstrapping*, includes the step of negotiating which data the node will be responsible for, and may include streaming data from other nodes to the new node. However, since the StatefulSet is not aware of these constraints, how can adding multiple nodes to a new or existing cluster one at a time be accomplished?

The answer lies in the composition of the Pod specification that Cass Operator passes to the StatefulSet, which is then used to create each Cassandra node, as shown in Figure 6-4.

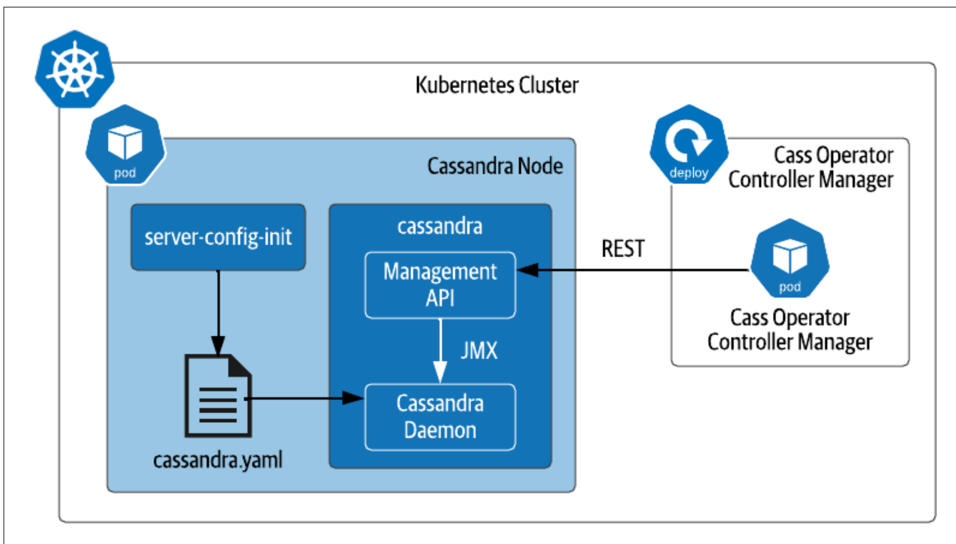
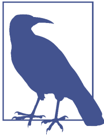


Figure 6-4. Cass Operator Interactions with Cassandra Pods

Cass Operator deploys a custom image of Cassandra in each Cassandra Pod that it manages. The Pod specification includes at least two containers: an init container called server-config-init and a Cassandra container called cassandra.

As an init container, server-config-init is started before the cassandra container. It's responsible for generating the cassandra.yaml configuration file based on the selected configuration options for the CassandraDataCenter. You can specify additional configuration values using the config section of the CassandraDataCenter resource, as described in the K8ssandra documentation.



Additional sidecar containers in Cassandra Pods

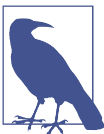
As you'll learn in the sections below, the Cassandra pod may have additional sidecar containers when deployed in a K8ssandraCluster, depending on which of the additional K8ssandra components you have enabled. For right now, though, we are focusing on the most basic installation.

The cassandra container actually contains two separate processes: the daemon that runs the Cassandra instance and a Management API. This goes somewhat against the traditional best practice of running a single process per container, but there is a good reason for this exception.

Cassandra's management interface is exposed via the Java Management Extensions (JMX). While this was a legitimate design choice for a Java-based application like Cassandra when the project was just starting out, JMX has fallen out of favor due to its complexity and security issues. While there has been some progress toward an alternate management interface for Cassandra, the work is not yet complete, so the developers of Cass Operator decided to integrate another open source project, the Management API for Apache Cassandra.

The Management API project provides a RESTful API that translates HTTP-based invocations into calls on Cassandra's legacy JMX interface. By running the Management API inside the Cassandra container, we avoid having to expose the JMX port outside of the Cassandra containers. This is an instance of a pattern frequently used in cloud native architectures to adapt custom protocols into HTTP-based interfaces, for which there is much better support for routing and security in ingress controllers.

Cass Operator discovers and connects to the Management API on each Cassandra Pod in order to perform management operations that are not related to Kubernetes. When adding new nodes, this involves the simple action of using the Management API to verify that the node is up and running successfully and updating the CassandraDatacenter's status accordingly. This sequence is described in more detail in the K8ssandra documentation.



Customizing the Cassandra image used by Cass Operator

The Management API project provides images for recent Cassandra versions in the 3.x and 4.x series which are available on Docker Hub. While it is possible to override the Cassandra image that Cass Operator uses with one of your own, Cass Operator does require that the Management API is available on each Cassandra Pod. If you need to build your own custom image including the Management API, you could use the Dockerfiles and supporting scripts from the GitHub repository as a starting point.

www.dbooks.org

While this section focused largely on the startup and scaling of Cassandra clusters described above, Cass Operator provides several features for deploying and managing Cassandra clusters:

- **Topology management** - Cass Operator uses Kubernetes affinity principles to manage the placement of Cassandra nodes (Pods) across Kubernetes worker nodes to maximize availability of your data.
- **Scaling down** - just as nodes are added one at a time to scale up, Cass Operator manages scaling down one node at a time.
- **Replacing nodes** - If a Cassandra node is lost because it crashes or the worker node on which it is running goes down, Cass Operator relies on the StatefulSet to replace the node and bind the new node to the appropriate PersistentVolume-Claim.
- **Upgrading images** - Cass Operator also leverages the capabilities of StatefulSet to perform rolling upgrades of the images used by the Cassandra Pods
- **Managing seed nodes** - Cass Operator creates Kubernetes Services to expose the seed nodes in each datacenter according to Cassandra's recommended conventions of one seed node per rack, for a minimum of three per datacenter.

You can read more about these and other features in the Cass Operator documentation.

Enabling Developer Productivity with Stargate APIs

Our focus so far in this book has been primarily on deployment of data infrastructure such as databases in Kubernetes, more than how that infrastructure is used in cloud-native applications. The usage of Stargate in K8ssandra gives us a good opportunity to have that discussion.

In many organizations, there is an ongoing conversation about the pros and cons of direct application access to databases versus abstracting the details of database interactions. This debate occurs especially frequently in larger organizations in which there is a division of responsibility between application development teams and teams that manage platforms including data infrastructure. However, it can also be observed in organizations that employ modern practices including DevOps and microservice architectures, where each microservice may have a different data store behind it.

The idea of providing abstractions over direct database access has taken many forms over the years. Even in the days of monolithic client-server applications, it was common to use stored procedures or isolate data access and complex query logic behind object-relational mapping tools such as Hibernate, or to use patterns like data access objects (DAOs).

More recently, as the software industry has moved toward service oriented architecture (SOA) and microservices, similar patterns for abstracting data access have appeared. As described in the article Data Services for the Masses, many teams have found themselves creating a layer of microservices in their architecture dedicated to data access, providing create, read, update, and delete (CRUD) operations on specific data types or entities. These services abstract the details of interacting with a specific database backend, and if well executed and maintained, can help increase developer productivity and facilitate migration to a different database when needed.

The Stargate project was born out of the realization that a number of teams were building very similar abstraction layers to provide data access via APIs. The goal of the Stargate project is to provide an open source *data API gateway* - a common set of APIs for data access to help eliminate the need for teams to develop and maintain their own custom API layers. While the initial implementation of Stargate is based on Cassandra, the goal of the project is to support multiple database backends, and even other types of data infrastructure such as caches and streaming.

With Cassandra used as the backend data store, the Stargate architecture can be described as having three layers, as shown in Figure 6-5. The *API layer* is the outermost layer, consisting of services that implement various APIs on top of the underlying Cassandra cluster. Available APIs include a REST API, a Document API that provides access to JSON documents over HTTP, a GraphQL API, and a gRPC API. The *routing layer* (or *coordination layer*) consists of a set of nodes that act as Cassandra nodes, but only perform routing of queries, not data storage. The *storage layer* consists of a traditional Cassandra cluster, which can currently be Cassandra 3.11, Cassandra 4.0, or DataStax Enterprise 6.8.

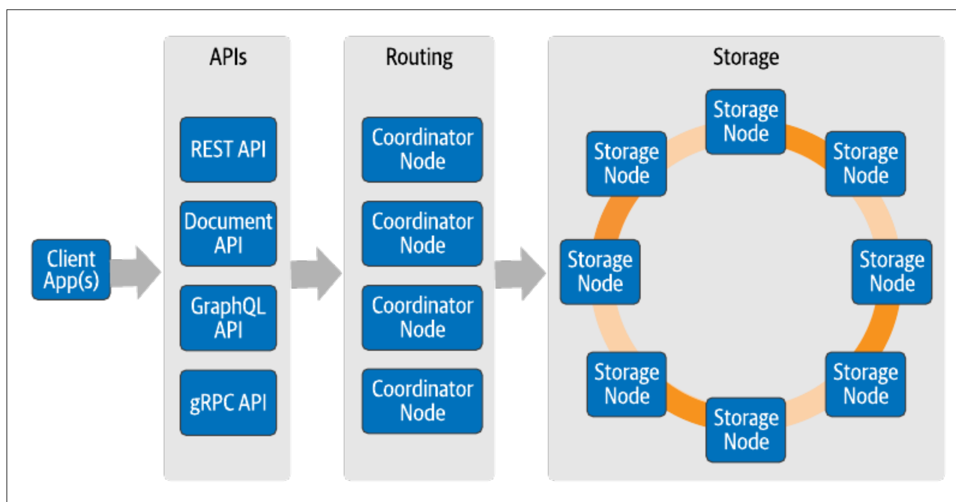


Figure 6-5. Stargate Conceptual Architecture with Cassandra

www.dbooks.org

One of the key benefits of this architecture is that it recognizes the separation of concerns for managing usage of compute and storage resources and provides the ability to scale this usage independently based on the needs of client applications:

- The number of storage nodes can be scaled up or down to provide the storage capacity required by the application.
- The number of coordinator nodes and API instances can be scaled up or down to match the application's read and write load and optimize throughput.
- APIs that are not used by the application can be scaled to zero (disabled) to reduce resource consumption.

K8ssandra supports the provision of Stargate on top of an underlying Cassandra cluster via the Stargate CRD. The CassandraDataCenter deployed by Cass Operator serves as the storage layer, and the Stargate CRD specifies the configuration of the routing and API layers. An example configuration is shown in Figure 6-6.

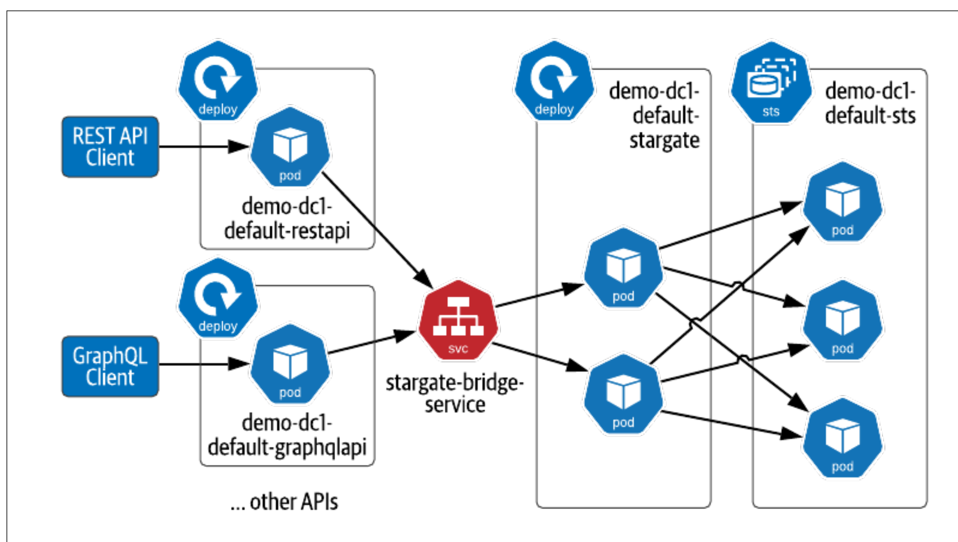


Figure 6-6. Stargate Deployment on Kubernetes

The installation includes a Deployment to manage the coordinator nodes, and a Service to provide access to the Bridge API, a private gRPC interface exposed on the coordinator nodes that can be used to create new API implementations. See the Stargate v2 Design for more details on the Bridge API. There is also a Deployment for each of the APIs that is enabled in the installation, along with a Service to provide access to client applications.

As you can see, the Stargate project provides a promising framework for extending your data infrastructure with developer-friendly APIs that can scale along with the underlying database.

Unified Monitoring Infrastructure with Prometheus and Grafana

Now that we've considered the addition of infrastructure that makes life easier for application developers, let's look at some of the more operations-focused aspects of integrating data infrastructure in a Kubernetes stack. We'll start with monitoring.

Observability is a key attribute of any application deployed on Kubernetes, since it has implications for your awareness of its availability, performance and cost. Your goal should be to have an integrated view across both your application and the infrastructure it depends on. Observability is often described as consisting of three types of data: metrics, logs, and tracing. Kubernetes itself provides capabilities for logging as well as associating events with resources, and you've already learned above how the Cass Operator facilitates the collection of logs from Cassandra nodes.

In this section, we'll focus on how K8ssandra incorporates the Prometheus/Grafana stack which provides metrics. Prometheus is a popular open source monitoring platform. It supports a variety of interfaces for collecting data from applications and services and stores them in a time series database which can be queried efficiently using the Prometheus Query Language (PromQL). It also includes an AlertManager which generates alerts and other notifications based on metric thresholds.

While previous releases of K8ssandra in the 1.X series incorporated the Prometheus stack as part of a K8ssandra, K8ssandra 2.X provides the capability to integrate with an existing Prometheus installation.

One easy way to install the Prometheus Operator is to use kube-prometheus, a repository provided as part of the Prometheus Operator project. Kube-prometheus is intended as a comprehensive monitoring stack for Kubernetes including the control plane and applications. You can clone this repository and use the library of manifests (yaml files) that it contains to install an integrated stack of components shown in Figure 6-7.

www.dbooks.org

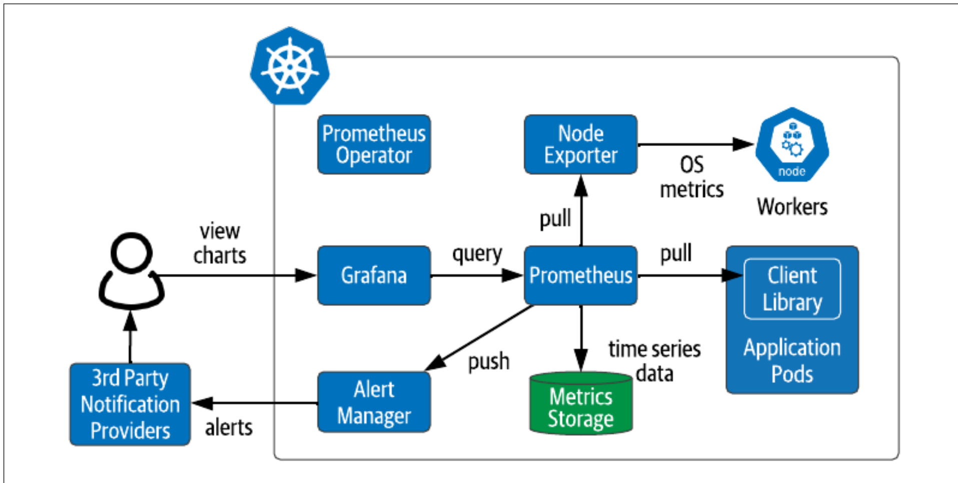


Figure 6-7. Components of the Kube-Prometheus Stack

These components include the following:

- Prometheus Operator - the operator, which is set apart in the figure, manages the other components.
- Prometheus - the metrics database is run in a high-availability configuration managed via a StatefulSet. Prometheus stores data using a time series database with a backing Persistent Volume.
- Node Exporter - the Node Exporter runs on each Kubernetes worker node, allowing Prometheus to pull operating system metrics via HTTP.
- Client Library - applications can embed a Prometheus client library, which allows Prometheus to pull metrics via HTTP.
- Alert Manager - The Prometheus Alert Manager can be configured to generate alerts based on thresholds for specific metrics for delivery via email or third-party tools such as PagerDuty. The kube-prometheus stack comes with built-in alerts for the Kubernetes cluster, and application-specific alerts can also be added.
- Grafana - this is deployed to provide charts that are used to display metrics to human operators. Grafana uses PromQL to access metrics from Prometheus, and this interface is available to other clients as well.

While not shown in the figure, the stack also includes the Prometheus Adapter for Kubernetes Metrics APIs, an optional component which exposes metrics collected by Prometheus to the Kubernetes control plane so that they can be used to auto-scale applications.

Connecting K8ssandra with Prometheus can be accomplished in a few quick steps. The instructions in the K8ssandra documentation walk you through installing the Prometheus Operator using kube-prometheus if you do not have it already. Since kube-prometheus installs Prometheus Operator in its own namespace, you'll want to make sure the operator has permissions to manage resources in other namespaces.

In order to integrate K8ssandra with Prometheus, you set attributes on your K8ssandraCluster resource to enable monitoring on Cassandra and Stargate nodes. For example, you could do something like the following to enable monitoring for nodes in all datacenters in the cluster:

```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: demo
spec:
  cassandra:
    datacenters:
      ...
    telemetry:
      prometheus:
        enabled: true
  stargate:
    telemetry:
      prometheus:
        enabled: true
```

It's also possible to selectively enable monitoring on individual datacenters, as described in the documentation.

Let's take a look at how the integration works. First, let's consider how the Cassandra nodes expose metrics. As discussed above in the Cass Operator section, Cassandra exposes management capabilities via JMX, and this includes metrics reporting. The Metrics Collector for Apache Cassandra (MCAC) is an open source project that exposes metrics so that they can be accessed by Prometheus or other backends that use the Prometheus protocol via HTTP. K8ssandra and Cass Operator use a Cassandra Docker image that includes MCAC as well as the Management API as additional processes that run in the Cassandra container. This configuration is shown on the left side of Figure 6-8.

www.dbooks.org

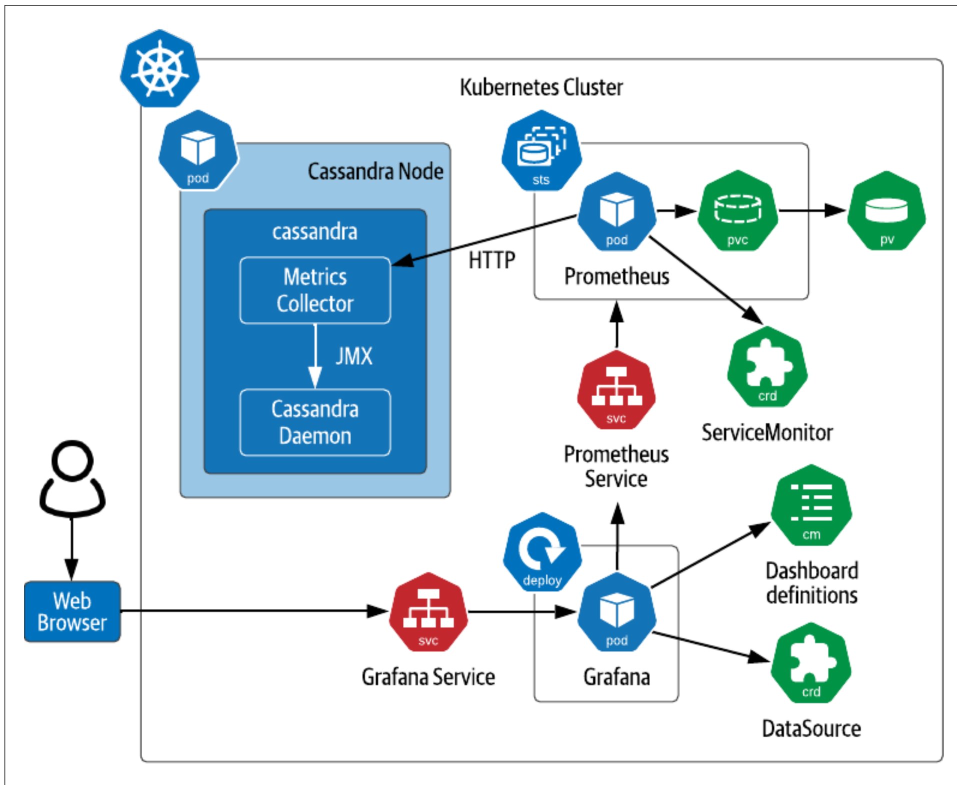


Figure 6-8. Monitoring Cassandra with Kube-Prometheus Stack

The right side of Figure 6-8 shows how Prometheus and Grafana are configured to consume and expose the Cassandra metrics. The K8ssandra Operator creates Service-Monitor resources for each CassandraDatacenter for which monitoring has been enabled. The ServiceMonitor is a CRD defined by Prometheus Operator which contains configuration details describing how to collect metrics from a set of Pods, including the following:

- A selector that references the name of a label which identifies the Pods
- Connection information such as the scheme (protocol), port, and path to use to gather metrics from each Pod
- The interval at which metrics should be pulled
- Optional metricRelabelings, which are instructions that indicate any desired renaming of metrics, or even indicate metrics that should be dropped and not ingested by Prometheus.

K8ssandra creates separate ServiceMonitor instances for Cassandra and Stargate nodes, since the metrics exposed are slightly different. To observe the ServiceMonitors deployed in your cluster, you can execute a command such as kubectl get servicemonitors -n monitoring.

Prometheus provides access to its metrics to Grafana and other tools via a PromQL endpoint exposed as a Kubernetes service. The kube-prometheus installation configures an instance of Grafana to connect to Prometheus using an instance of the Grafana DataSource CRD. Grafana accepts dashboards defined using yaml files, which you can provide as ConfigMaps. See the K8ssandra documentation for guidance on loading dashboard definitions that display Cassandra and Stargate metrics. You may also wish to create dashboards that display your application metrics alongside the data tier metrics provided by K8ssandra for an integrated view of application performance.

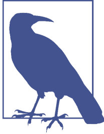
As you can see, kube-prometheus provides a comprehensive and extensible monitoring stack for Kubernetes clusters, much as K8ssandra provides a stack for data management. The integration of K8ssandra with kube-prometheus is a great example of how you can assemble integrated stacks of Kubernetes resources to form even more powerful applications.

Performing Repairs with Cassandra-Reaper

As a NoSQL database, Cassandra emphasizes high performance (especially for writes) and high availability by default. If you're familiar with the CAP theorem, you'll understand that this means that sometimes Cassandra will temporarily sacrifice consistency of data across nodes in order to deliver this high performance and high availability at scale, an approach known as *eventual consistency*. Cassandra does provide the ability to tune the amount of consistency to your needs via options for specifying replication strategies and the consistency level required per query. Users and administrators should be aware of these options and their behavior in order to use Cassandra effectively.

Cassandra has multiple built-in "anti-entropy" mechanisms such as hinted handoff and repair that help maintain consistency of data between nodes over time. Repair is a background process by which a node compares a portion of the data it owns with the latest contents of other nodes that are also responsible for that data. While these checks can be somewhat optimized through the use of checksums, repair can still be a performance intensive process and is best performed when a cluster is under reduced or off-peak load. Combined with the fact that there are a number of different options available including full and incremental repairs, executing repairs has traditionally been a process that requires some tailoring for each cluster. It also has tended to be a manual process that was unfortunately frequently neglected by some Cassandra cluster administrators.

www.dbooks.org



More detail on repairs in Cassandra

For a deeper treatment of repair, see Cassandra: The Definitive Guide, where repair concepts and the available options are described in Chapters 6 and 12, respectively.

Cassandra Reaper was created to take the difficulty out of executing repairs on Cassandra clusters and optimize repair performance to minimize the impact of running repairs on heavily used clusters. Reaper was created by Spotify and enhanced by The Last Pickle, who currently manage the project on GitHub. Reaper exposes a RESTful API for configuring repair schedules for one or more Cassandra clusters, and also provides a command line tool and web interface which guides administrators through the process of creating schedules.

K8ssandra provides the option to incorporate an instance of Cassandra Reaper as part of a K8ssandraCluster. The K8ssandra Operator includes a Reaper Controller that is responsible for managing the local Cassandra Reaper manager process through its associated Reaper CRD. By default, enabling Reaper in a K8ssandraCluster will cause an instance of Reaper to be installed in each Kubernetes cluster represented in the installation, but you can also use a single instance of Reaper to manage repairs across multiple datacenters, or even across multiple Cassandra clusters, provided they are accessible via the network.

How important is it to be Kubernetes-native?

K8ssandra's usage of Reaper is an example of the tradeoffs involved in building more complex stacks of data infrastructure. For example, a more Kubernetes-native design for the Reaper Manager might involve factoring out each repair task into a Kubernetes CronJob that could be scheduled alongside the associated CassandraDatacenter, thus making more use of Kubernetes built-in resources. For now the K8ssandra project has made the choice to integrate Reaper as-is.

We saw another example of this "wrap vs. rewrite" type of decision in Chapter 5, where the Vitess Operator reuses the Vitess control daemon *vtctld* and its *vtctlclient* as-is. In both of these examples, the project developers have made pragmatic choices to do initial deployments that do "just enough" to port existing infrastructure to run in Kubernetes, while leaving room for more Kubernetes-native approaches in the future. In Chapter 7 we'll examine what it looks like to start with a Kubernetes-native approach from scratch on new infrastructure projects.

Backing up and Restoring Data with Cassandra Medusa

Managing backups is an important part of maintaining high availability and disaster recovery planning for any system that stores data. Cassandra supports both full and

differential backups by creating hard links to the SSTable files it uses for data persistence. Cassandra itself does not take responsibility for copying the SSTable files to backup storage. Instead, this is left to the user. Similarly, recovering from backup involves copying the SSTable files to the Cassandra node where the data is to be reloaded, then Cassandra can be pointed to the local files to restore their contents.

Cassandra's backup and restore operations are traditionally executed on individual nodes using nodetool, a command line tool that leverages Cassandra's JMX interface. Cassandra Medusa is an open source command line tool created by Spotify and The Last Pickle that executes nodetool commands to perform backups, including synchronization of backups across multiple nodes. Medusa supports Amazon S3, Amazon S3, Google Cloud Storage (GCS), Azure Storage, and S3 compatible such as MinIO, Ceph Object Gateway, and can be extended to support other storage providers via the Apache Libcloud project.

Medusa can restore either individual nodes to support fast replacement of a downed node, or entire clusters in a disaster recovery scenario. Restoring to a cluster can either be to the original cluster or to a new cluster. Medusa is able to restore data to a cluster with a different size or topology than the original cluster, which has traditionally been a challenge to figure out manually.

K8ssandra has incorporated Medusa in order to provide backup and restore capabilities for Cassandra clusters running in Kubernetes. To configure the use of Medusa in a K8ssandraCluster, you'll want to configure the medusa properties:

```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: demo
spec:
  cassandra:
    ...
  medusa:
    storageProperties:
      storageProvider: google_storage
      storageSecretRef:
        name: medusa-bucket-key
      bucketName: k8ssandra-medusa
      prefix: test
      ...
```

The options shown here include the storage provider, the bucket to use for backups, an optional prefix to add to directory names used to organize backup files, and the name of a Kubernetes Secret containing login credentials for the bucket. See the documentation for details on the contents of the Secret. Other available options include enabling SSL on the bucket connection, and setting the policies for purging old backups such as a maximum age or number of backups.

www.dbooks.org

Creating a backup

Once the K8ssandraCluster has been started, you can create backups using the CassandraBackup CRD. For example, you could initiate a backup of the CassandraDatacenter dc1 using a command like this:

```
cat <<EOF | kubectl apply -f -n k8ssandra-operator -
apiVersion: medusa.k8ssandra.io/v1alpha1
kind: CassandraBackup
metadata:
  name: medusa-backup1
spec:
  cassandraDatacenter: dc1
  name: medusa-backup1
EOF
```

The steps in processing of this resource are shown in Figure 6-9. (1) When you apply the resource definition, (2) kubectl registers the resource with the API Server, (3) which notifies the Medusa Controller running as part of the K8ssandra Operator. (4) Medusa Controller contacts a sidecar container which K8ssandra has injected into the Cassandra Pod because you chose to enable Medusa on the K8ssandraCluster. (5) The Medusa sidecar container uses nodetool commands to a backup on the Cassandra node via JMX (the JMX interface is only exposed within the Pod). (6) Cassandra performs a backup, marking the SSTable files on the PersistentVolume that mark the current snapshot. (7) The Medusa sidecar copies the snapshot files from the PV to the bucket. Steps 4-7 are repeated for each Cassandra Pod in the CassandraDatacenter.

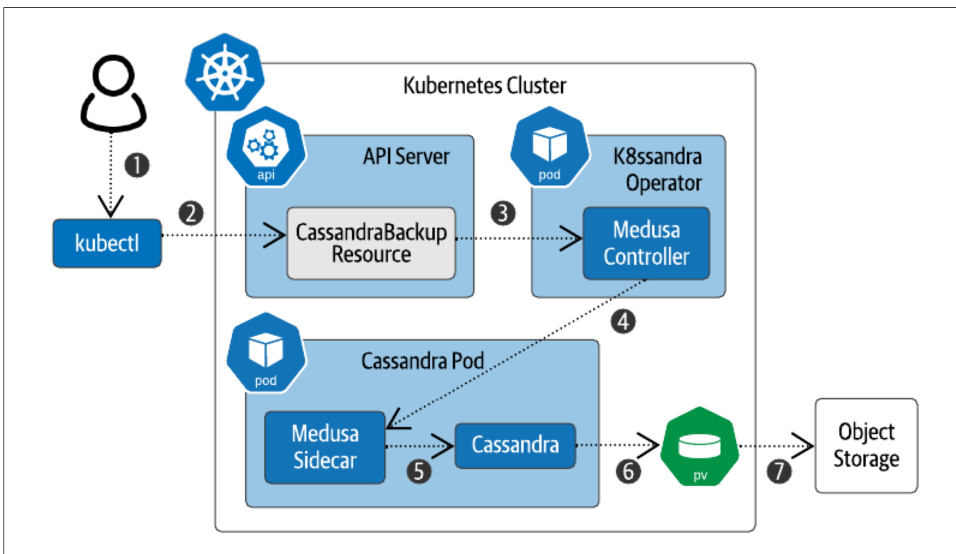


Figure 6-9. Performing a Datacenter backup using Medusa

You can monitor the progress of the backup by checking the status of the resource:

```
kubectl get cassandrabackup/medusa-backup1 -n k8ssandra-operator -o yaml
kind: CassandraBackup
metadata:
    name: medusa-backup1
spec:
  backupType: differential
  cassandraDatacenter: dc1
  name: medusa-backup1
status:
  ...
  ...
  finishTime: "2022-02-26T09:21:38Z"
  finished:
  - demo-dc1-default-sts-0
  - demo-dc1-default-sts-1
  - demo-dc1-default-sts-2
  startTime: "2022-02-26T09:21:35Z"
```

You'll know the backup is complete when the finishTime attribute is populated. The Pods that have been backed up are listed under the finished attribute.

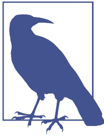
Restoring from backup

The process of restoring data from a backup is similar. To restore an entire datacenter from backed up data, you could create a CassandraRestore resource like this:

```
cat <<EOF | kubectl apply -f -n k8ssandra-operator -
apiVersion: medusa.k8ssandra.io/v1alpha1
kind: CassandraRestore
metadata:
  name: restore-backup1
spec:
  cassandraDatacenter:
    name: dc1
    clusterName: demo
  backup: medusa-backup1
  inPlace: true
  shutdown: true
EOF
```

When the Medusa Controller is notified of the new resource, it locates the Cassandra-Datacenter and updates the Pod spec template within the StatefulSet that is managing the Cassandra Pods. The updates consist of adding a new init container called medusa-restore and setting environment variables that medusa-restore will use to locate the data files that are to be restored. The update to the Pod spec template causes the StatefulSet controller to perform a rolling update of the Cassandra Pods in the StatefulSet. As each Pod restarts, medusa-restore copies the files from object storage onto the PersistentVolume for the node and then the Cassandra container starts as usual. You can monitor the progress of the restore by checking the status of the CassandraRestore resource.

www.dbooks.org



A common language for data recovery?

It is interesting to note the similarities and differences between the ways backup and restore operations are supported by the K8ssandra Operator we've discussed in this chapter and the Vitess Operator discussed in Chapter 5.

In K8ssandra, the CassandraBackup and CassandraRestore resources function in a manner similar to Kubernetes Jobs - they represent a task that you would like to have performed as well as the results of the task. In contrast, the VitessBackup resource represents a record of a backup that the Vitess Operator has performed based on the configuration of a VitessCluster resource. There is no equivalent resource to the CassandraRestore operator in Vitess.

Although there are significant differences between K8ssandra and Vitess in the approach to managing backups, they both represent each backup task as a resource. Perhaps this common ground could be the starting point toward the development of common resource definitions for backup and restore operations, helping fulfill the vision introduced in Chapter 5.

Similar to the behavior of Cassandra Reaper, a single instance of Medusa can be configured to manage backup and restore operations across multiple datacenters or Cassandra clusters. See the K8ssandra documentation for more details on performing backup and restore operations with Medusa.

Deploying Multi-cluster applications in Kubernetes

One of the main selling points of a distributed database like Cassandra is its ability to support deployments across multiple data centers. Many users take advantage of this in order to promote high availability across geographically distributed datacenters, to provide lower latency reads and writes for applications and their users.

However, Kubernetes itself was not originally designed to support applications that span multiple Kubernetes clusters. This has traditionally meant that creating such multi-region applications leaves a lot of work to development teams. This work takes two main forms: creating the network infrastructure to connect the Kubernetes clusters, and coordinating interactions between resources in those clusters.

Let's examine these requirements and the implications for an application like Cassandra:

Multi-cluster networking requirements
: From a networking perspective, the key is to have secure, reliable networking between datacenters. If you're using a single cloud provider for your application, this may be relatively simple to achieve using virtual private cloud (VPC) capabilities offered by the major cloud vendors.

If you're using multiple clouds, you'll need a third-party solution. For the most part, Cassandra requires routable IPs between its nodes and does not rely on name resolution, but it is helpful to have domain name resolution (DNS) in place as well to simplify the process of managing Cassandra's seed nodes.

The blog Deploy a Multi-Data Center Cassandra Cluster in Kubernetes describes an example configuration in Google Cloud Platform (GCP) using the CloudDNS service, while Multi-Region Cassandra on EKS with K8ssandra and Kubefed describes a similar configuration on Amazon's Elastic Kubernetes Service (EKS).

Multi-cluster resource coordination requirements

Managing an application that spans multiple Kubernetes clusters means that there are distinct resources in each cluster which have no relationship to resources in other clusters that the Kubernetes control plane is aware of. In order to manage the lifecycle of an application including deployment, upgrade, scaling up and down, and teardown, you need to coordinate resources across multiple datacenters.

The Kubernetes Cluster Federation project (KubeFed for short) provides one approach to providing a set of APIs for managing resources across clusters that can be leveraged to build multi-cluster applications. This includes mechanisms that represent Kubernetes clusters themselves as resources. While KubeFed is still in beta, the K8ssandra Operator uses a similar design approach for managing resources across clusters. We'll examine this in more detail in Kubernetes Cluster Federation.

In order to achieve a multi-cluster Kubernetes deployment of Cassandra, you'll need to establish networking between datacenters according to your specific situation. Given that foundation, the K8ssandra Operator provides the facilities to manage the lifecycle of resources across the Kubernetes clusters. For a simple example of deploying a multi-region K8ssandraCluster, use the instructions found in the K8ssandra documentation, again using the Makefile:

```
make multi-up
```

This builds two Kind clusters, deploys the K8ssandra Operator in each of them, and creates a multi-cluster K8ssandraCluster. One advantage of using Kind for a simple demonstration is that Docker provides the networking between clusters. We'll walk through some of the key steps in this process in order to describe how the K8ssandra Operator accomplishes this work.

The K8ssandra Operator supports two modes of installation - control plane (the default) and data plane. For a multi-cluster deployment, one Kubernetes cluster must be designated as the control plane cluster, and the others as data plane clusters. The control plane cluster can optionally include a CassandraDatacenter, as in the configuration shown in Figure 6-10.

www.dbooks.org

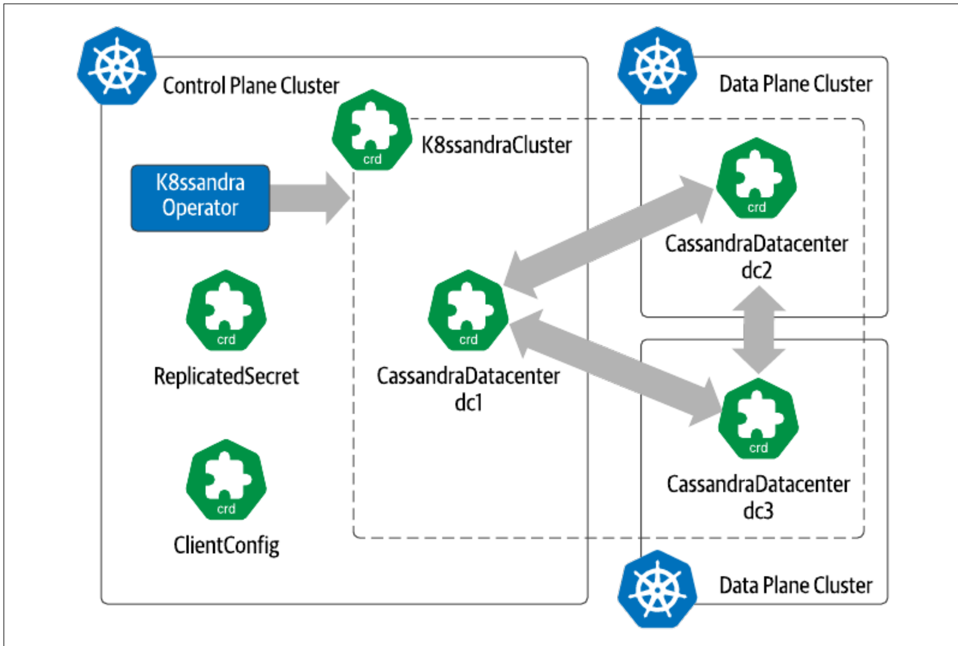


Figure 6-10. K8ssandra Multi-cluster Architecture

When installed in control plane mode, the K8ssandra Operator uses two additional CRDs to manage multi-cluster deployments: ReplicatedSecret and ClientConfig. You can see evidence of the ClientConfig in the K8ssandraCluster configuration that was used, which looks something like the following:

```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: demo
spec:
  cassandra:
    serverVersion: "4.0.1"
    ...
    networking:
      hostNetwork: true
    datacenters:
      - metadata:
          name: dc1
        size: 3
        stargate:
          size: 1
      - metadata:
          name: dc2
        k8sContext: kind-k8ssandra-1
        size: 3
```

```
      stargate:
        size: 1
```

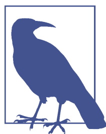
This configuration specifies a K8ssandraCluster demo consisting of two CassandraDatacenters, dc1 and dc2. Each datacenter has its own configuration so that you can select a different number of Cassandra and Stargate nodes, or different resource allocations for the Pods. In the demo configuration, dc1 is running in the control plane cluster kind-k8ssandra-0, and dc2 is running in the data plane cluster kind-k8ssandra-1.

Notice the line in the configuration that says k8sContext: kind-k8ssandra-1. This is a reference to a ClientConfig resource that was created by the make command. A ClientConfig is a resource that represents the information needed to connect to the API server of another cluster, similar to the way kubectl stores information about different clusters on your local machine. The ClientConfig resource references a Secret that is used to store access credentials securely. The K8ssandra Operator repo includes a convenience script that can be used to create ClientConfig resources for Kubernetes clusters.

When you create a K8ssandraCluster in the control plane cluster, it uses the ClientConfigs to connect to each remote Kubernetes cluster in order to create the specified resources. For the configuration shown above, this includes CassandraDatacenter and Stargate resources, but can also include other resources such as Medusa, Prometheus ServiceMonitor.

The ReplicatedSecret is another resource involved in sharing access credentials. The control plane K8ssandra Operator uses this resource to keep track of Secrets that it creates in each remote cluster. These secrets are used by the various K8ssandra components in order to communicate securely with each other, for example the default Cassandra administrator credentials. The K8ssandra Operator creates and manages ReplicatedSecret resources itself, you don't need to interact with them.

The K8ssandraCluster, ClientConfig, and ReplicatedSecret resources only exist in the control plane cluster, and when the K8ssandra Operator is deployed in data plane mode, it does not even run the controllers associated with those resource types.



More detail on the K8ssandra Operator

This is a quick summary of a complex design for a multi-cluster operator. For more details on the approach, see the K8ssandra Operator architecture overview and John Sanda's presentation at the Data on Kubernetes community meetup.

Now let's consider a more general approach to building multi-cluster applications that we can compare and contrast with K8ssandra's approach.

www.dbooks.org

Kubernetes Cluster Federation

With Irfan Ur Rehman, Senior Engineer, Turbonomic (an IBM company)

KubeFed is a project for building multi-cluster applications, managed by the Kubernetes Multi Cluster SIG. The project was initially called Federation, but was renamed to Kubernetes Cluster Federation (or KubeFed for short), in order to disambiguate it from usage of the term federation in other projects outside of Kubernetes.

KubeFed defines federation as joining a set of clusters into a pool, which then provides a unified API to the user to distribute applications into those clusters. To use Kubefed, you create federated resources in a base cluster. A federated resource contains templates for Kubernetes built-in or custom resources. KubeFed acts as a resource reconciler, using the templates you provide to push resources to the member clusters.

You might want the templates to be applied in slightly different ways in each member cluster, so KubeFed supports concepts called placements and overrides. A placement is a definition of where applications and their resources are deployed. For example, you could use a placement to push resources in cluster one, but not cluster two, or to indicate you want more replicas in one cluster than another cluster. Overrides allow you to provide different values for resource attributes for a specific cluster.

KubeFed also provides resources to support higher order things you might want to do. The ReplicaScheduler is a resource that manages Deployments and ReplicaSets. This allows you to deploy your application by specifying the total number of replicas desired across clusters, without worrying about which clusters they go to. You can do something similar for StatefulSets.

Cluster federation, placements, and overrides are three key concepts defined by Kubefed, along with others defined on the concepts page. These terms have gained wide popularity and are used across other projects as well. For example, ArgoCD is a GitOps toolset for Kubernetes which employs similar concepts such as placement rules and overrides.

There are other multi-cluster projects in the Kubernetes ecosystem which have similar goals but differ in implementation and scope:

KArmada is a project sponsored by Huawei which is similar to KubeFed but takes a different API approach. KArmada reuses existing Kubernetes resources but extends them with additional attributes in order to provide the appearance of a single Kubernetes cluster.

Crossplane is a CNCF incubating project which aims to provide a single API surface for you to distribute resources and consume services from multiple clouds. Crossplane uses the same declarative approach as Kubernetes but goes beyond just Kubernetes resources, allowing you to incorporate offerings from the major cloud providers such as database as a service (DBaaS) or network as a service (NaaS).

Open Cluster Management is a project sponsored by Redhat which provides an ecosystem of components for working across multiple Kubernetes clusters.

Each of these projects take a similar approach at a high level but have their own opinionated APIs and nuances which might be more suitable to different users.

KubeFed and these similar projects are primarily concerned with resource replication. In order to have multi-cluster applications, you also need networking solutions, which can get a little more complex. One approach is to create cross cluster network overlays using open source projects like Submariner or Cilium.

Even with the network in place, you still have the problem of discovering applications and resources across clusters and connecting them securely. The Multi-Cluster API is a proposal in the Kubernetes Multi-cluster SIG for providing this discovery. It is based on a concept called endpoint slices which allow a cluster to discover services from another cluster. An alpha implementation is available.

Although KubeFed is still in Beta status, it is in a mature state and some organizations are already using it in production. The core functionality of reconciling resources across clusters is something that just works. The main item in the KubeFed roadmap is a general availability release, which should lead to further adoption.

Adoption can be a chicken and egg problem, because organizations often prefer to back established projects. Throughout its history, Kubefed has had support from RedHat/IBM, Huawei, D2IQ and others, and backing by larger organizations is important for driving adoption by the larger community.

It is challenging to come up with a single standard. There is a lack of incentive for major cloud providers to contribute to these efforts vs. supporting tooling centered on their own platforms, so it is up to us in the open source community to invest in this area.

As you can see, there is a lot of potential for growth in the area of Kubernetes federation and the ability to manage resources across Kubernetes cluster boundaries. For example, as a database whose primary superpower is running across multiple data centers, Cassandra seems like a great match for a multi-cluster solution like KubeFed.

The K8ssandra Operator and KubeFed have taken similar architectural approaches, where custom "federated" resources provide templates used to define resources in other clusters. This commonality points to the possibility for future collaboration across these projects and others based on similar design principles. Perhaps in the future, CRDs like K8ssandra's ClientConfig and ReplicatedSecret can be replaced by equivalent functionality provided by Kubefed.

www.dbooks.org

Summary

In this chapter you've learned how data infrastructure can be composed with other infrastructure to build reusable stacks on Kubernetes. Using the K8ssandra project as an example, you've learned about aspects including integrating data infrastructure with API gateways and monitoring solutions to provide more full-featured solutions.

You've also learned some of the opportunities and challenges with adapting existing technologies onto Kubernetes and creating multi-cluster data infrastructure deployments. In the next chapter we'll explore how to design new cloud-native data infrastructure that takes advantage of everything that Kubernetes provides without requiring adaptation and discover what new possibilities that opens up.

www.dbooks.org

CHAPTER 7
The Kubernetes Native Database

> **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 7th chapter of the final book. The GitHub repo is 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.

The software industry is flush with terms that define major trends in a single word or short phrase. You can see one of them in the title of this book: cloud native. Another example is microservice", a major architectural paradigm that touches much of the technology we're discussing here. More recently, terms like *Kubernetes native* and *serverless* have emerged.

While succinct and catchy, distilling a complex topic or trend down to a single sound bite leaves room for ambiguity, or at least for reasonable questions like, "What does this *actually* mean"? To further muddy the waters, terms such as these are frequently used in the context of marketing products as a way to gain leverage or differentiate against other competitive offerings. Whether the content you're consuming makes an overt statement or it's just the subtext, you may have wondered whether a given technology must be better to run on Kubernetes than other offerings because it's labeled "Kubernetes native".

Of course, for these terms to be useful to us in evaluating and picking the right technologies for our applications, the real task is to unpack what they really mean, as we did with the term *cloud native data* in Chapter 1. In this chapter, we'll look at what it means for data technology to be Kubernetes native and see if we can arrive at a definition that we can agree on. To do this, we'll examine a couple of projects that claim these terms and derive the common principles: TiDB and Astra DB. Are you ready? Let's dive in!

Why a Kubernetes native approach is needed

First, let's discuss why the idea of a Kubernetes native database came up in the first place. Up to this point in the book, we've focused on deployment of existing databases on Kubernetes including MySQL and Cassandra. These are mature databases that were around before Kubernetes existed and have proven themselves over time. They have large install bases and user communities, and because of this investment you can see why there's a large incentive to run these databases in Kubernetes environments, and why there has been such interest in creating operators to automate them.

At the same time, you've probably noticed some of the awkwardness in adapting these databases to run on Kubernetes. While it is pretty straightforward to point a database to Kubernetes-based storage just by changing a mount path, tighter integration with Kubernetes to manage databases that consist of multiple nodes can be a bit more involved. This can range from relatively simple tasks like deploying a legacy management UI in a Pod and exposing access to the HTTP port, to the more complex deployment of sidecars that we saw in Chapter 6 to provide APIs for management and metrics collection.

The recognition of this complexity has led some innovators to develop new databases that are designed to be cloud native or Kubernetes native from day one. It's a well known axiom in the database industry that it takes 5-10 years for a new database engine to reach a point of maturity. Because of this, cloud native databases tend not to be completely new implementations, but rather refactoring of existing databases into microservices that can be scaled independently, while maintaining compatibility with existing APIs that developers are accustomed to. Thus, the trend of decomposing the monolith has arrived at the data tier. The emerging generation of databases will be based on new architectures in order to truly maximize the benefits of the cloud.

In order to help us assess these new databases and their architectures, recall the cloud native data principles introduced in Chapter 1. We'll repeat the principles briefly here and add a few questions we might ask about any data technology running on Kubernetes:

www.dbooks.org

Principle 1: Leverage compute, network, and storage as commodity APIs
: How does the database use Kubernetes compute resources (Pods, Deployments, StatefulSets), network resources (Services and Ingress), and storage resources (PersistentVolumes, PersistentVolumeClaims, StorageClasses)?

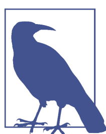
Principle 2: Separate the control and data planes
: Is the database deployed and managed by an operator? What custom resources does it define? Are there other workloads in the control plane besides the operator?

Principle 3: Make observability easy
: How do the various services in the architecture expose metrics and logs to support collection by the Kubernetes control plane and third party extensions?

Principle 4: Make the default configuration secure
: Do the database and associated operator make use of Kubernetes Secrets to share credentials, and use Roles, RoleBindings to manage access by role? Do services minimize the number of exposed points and require secure access to them?

Principle 5: Prefer declarative configuration
: Extending principle 2, can the database be managed entirely by creating, updating or deleting Helm charts and Kubernetes resources (whether built-in or custom resources), or are other tools required?

In the sections that follow, we'll explore the answers to these questions for specific data technologies and see what else we can learn about what it means to be Kubernetes native.



What's the difference between cloud native and Kubernetes native?

These are two similar sounding terms that are often used interchangeably, but do they have distinct meaning? We think so. After all, on the one hand, it's possible to run technology on Kubernetes in a way that does not embody cloud native principles. On the other hand, it's possible to take a cloud native approach in deploying to other environments besides Kubernetes. For some quick definitions, let's think of cloud native as implying a microservice approach with independently scalable services, and Kubernetes native as an application of cloud native principles while using as many Kubernetes resources and APIs as possible. We'll provide a checklist at the end of this chapter that will help solidify these definitions.

With this background, let's look at some examples of databases that embody cloud native principles.

Hybrid data access at scale with TiDB

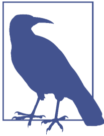
The databases that have received most of our focus in this book so far represent two major trends in database architecture that trace their lineage back for decades or more. MySQL is a relational database that provides its own flavor of the Standard Query Language (SQL), based on rules developed by Edgar Codd dating back to the 1970s.

In the early 2000s, companies building web scale applications began to push the limits of what could be accomplished with the relational databases of the day. As database sizes began growing beyond what could feasibly be managed on a single instance, techniques like sharding were used to scale across multiple instances. These were frequently expensive, difficult to operate, and not always reliable.

In response to this need, Cassandra and other so called "NoSQL" databases emerged in a period of intense innovation and experimentation. These databases provide linear scalability through adding additional nodes. They offer different data models, or ways of representing data: for example, key-value stores such as Redis, document databases such as MongoDB, graph databases such as Neo4J, and others. NoSQL databases tended to provide weaker consistency guarantees and omit support for more complex behaviors like transactions and joins in order to achieve high performance and availability at scale, a tradeoff documented by Eric Brewer in his CAP Theorem.

Because of the continued developer demand for traditional relational semantics such as strong consistency, transactions, and joins, multiple teams began to revive the idea of supporting these capabilities in distributed databases starting around 2012. These so-called "NewSQL" databases were based on more efficient and performant consensus algorithms. Two key papers helped drive the emergence of the NewSQL movement. First, the Calvin paper introduced a global consensus protocol which represented a more reliable and performant approach for guaranteeing strong consistency, later adopted by FaunaDB and other databases. Second, Google's Spanner paper introduced a design for a distributed relational database using sharding and a new consensus algorithm that leveraged the improved ability of cloud infrastructure to provide time synchronization across data centers. Besides Google Spanner, this approach was implemented by databases including CockroachDB and YugaByteDB.

www.dbooks.org



More on consistency and consensus

While we don't have space in this book to dive deeply into the tradeoffs between various consensus algorithms and how they are used to provide various data consistency guarantees, an understanding of these concepts is helpful in choosing the right data infrastructure for your cloud applications. If you're interested in learning more in this area, Martin Kleppmann's Desigining Data Intensive Applications (O'Reilly) is a great source, especially Chapter 9, Consistency and Consensus.

TiDB (where Ti stands for "Titanium") represents a continuation of the NewSQL trend in the cloud native space. TiDB is an open source, MySQL-compatible database that was initially developed and primarily supported by PingCAP, Inc.

TiDB Architecture

A key characteristic of TiDB which distinguishes it from other databases we've examined so far in this book is its ability to support transactional and analytic workloads. This approach, known as Hybrid Transactional and Analytical Processing (HTAP), supports both types of queries without a separate extract, transform, and load (ETL) process. As shown in Figure 7-1, TiDB does this by providing two different database engines under the hood: TiKV and TiFlash. This approach was inspired by Google's F1 project, a layer built on top of Spanner.

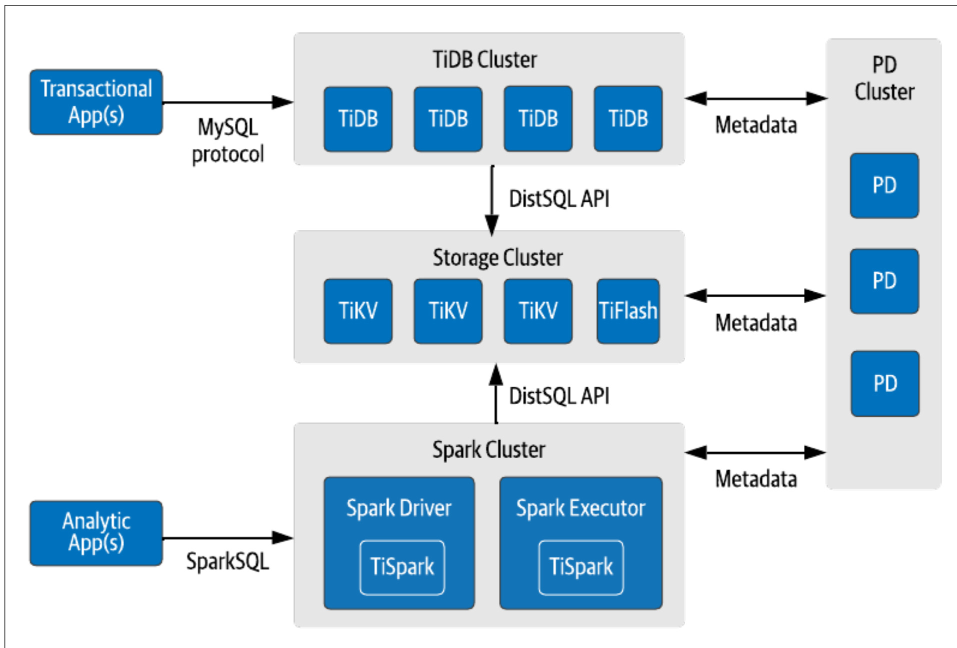


Figure 7-1. TiDB Architecture

One key aspect that gives TiDB a cloud native architecture is the packaging of compute and storage operations into separate components, each of which are composed of independently scalable services organized in clusters. Let's examine the roles of each of these components.

TiDB
: Each TiDB instance is a stateless service that exposes a MySQL endpoint to client applications. TiDB parses incoming SQL requests and uses metadata from the Placement Driver to create an execution plan containing queries to make on specific TiKV and TiFlash nodes in the storage cluster. TiDB executes these queries, assembles the results, and returns to the client application. The TiDB cluster is typically deployed with a proxy in front of it to provide load balancing.

TiKV
: The storage cluster consists of a mixture of TiKV and TiFlash nodes. First, let's examine TiKV. TiKV is an open source, distributed key-value database that uses RocksDB as its backing storage engine. TiKV exposes a custom Distributed SQL API that the TiDB nodes use to execute queries to store and retrieve data and manage distributed transactions. TiKV stores multiple replicas of your data, typically at least three, to support high availability and automatic failover. TiKV is a CNCF Graduated project which can be used independently from TiDB, as we'll discuss below.

www.dbooks.org

TiFlash

The storage cluster also includes TiFlash nodes, to which data is replicated from TiKV nodes as it is written. TiFlash is a columnar database based on the open source Clickhouse analytic database, which means that it organizes data storage in columns rather than rows. Columnar databases can provide a significant performance advantage for analytic queries requiring the extraction of the same column across multiple rows.

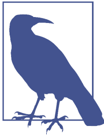
TiSpark

TiSpark is a library built for Apache Spark to support complex analytic (OLAP) queries. TiSpark integrates with the Spark Driver and Spark Executors, providing the capability to ingest data from TiFlash instances using the Distributed SQL API. We'll examine the Spark architecture and the details of deploying Spark on Kubernetes in Chapter 9.

Placement Driver (PD)

The Placement Driver manages the metadata for a TiDB installation. Placement Driver instances are deployed in a cluster of at least three nodes. TiDB uses a range based sharding mechanism where the keys in each table are divided into ranges called regions. The Placement Driver is responsible for determining the ranges of data that are assigned to each region, and the TiKV nodes that will store the data for each region. It monitors the amount of data in each region and splits regions that become too large in order to facilitate scaling up, and merging smaller regions in order to scale down.

Because the TiDB architecture consists of well defined interfaces between the different components, it is an extensible architecture in which different pieces can be plugged in. For example, TiKB provides a distributed key-value storage solution that can be reused in other applications. The TiPrometheus project is an example, providing a Prometheus-compliant compute layer on top of TiKB. For another example, you could provide an alternate implementation of TiKB that implements the Distributed SQL API on top of a different storage engine.



Pluggable storage engines

In this chapter so far we've made several mentions of "storage engines" or "database engines." This term refers to the part of the database that manages the storage and retrieval of data on persistent media. In distributed databases, a distinction is often made between the storage engine and the proxy layer which sits on top of it to manage data replication between nodes. Chapter 3, Storage and Retrieval from Desigining Data Intensive Applications (O'Reilly) includes discussion of different storage engine types such as the B-Trees used in most relational databases and the Log-Structured Merge Tree (LSM Tree) used in Apache Cassandra and other NoSQL databases.

One interesting aspect of TiDB is the way in which it reuses existing technology. We've seen examples of this above in the usage of components including RocksDB and Spark. TiDB also makes use of algorithms developed by other organizations. Here are a couple of examples:

Raft Consensus Protocol
: At the TiKB layer, the Raft consensus protocol is used to manage consistency between replicas. Raft is similar to the Paxos algorithm used by Cassandra in terms of its behavior, but is designed to be much simpler to learn and use. TiKB uses a separate Raft group for each region, where a group typically consists of a leader and two or more replicas. If a leader node is lost, an election is run to select a new leader, and a new replica can be added to ensure the desired number of replicas. In addition, the TiFlash nodes are configured as a special type of replica called learner replicas. Data is replicated to learner replicas from the TiKB nodes, but they cannot be selected as a leader. You can read more about how TiDB uses Raft for high availability and other related topics on the PingCap Blog.

Percolator transaction management
: At the TiDB layer, distributed transactions are supported using an implementation of the Percolator algorithm with optimizations specific to the TiDB project. Percolator was originally developed at Google for supporting incremental updates to search indexes.

One of the arguments we're making in this chapter is that part of what it means for data infrastructure to be cloud native is to compose existing APIs, services and algorithms wherever possible, and TiDB is a great example of this.

Deploying TiDB in Kubernetes

While TiDB can be deployed in a variety of ways including bare metal and VMs, the TiDB team has invested a large effort in tooling and documentation to make TiDB a

www.dbooks.org

truly Kubernetes native database. The TiDB Operator manages TiDB clusters in Kubernetes, including deployment, upgrade, scaling, backup and resorts, and more.

The operator documentation provides quick start guides for desktop Kubernetes distributions such as Kind, Minikube, as well as GKE. These instructions guide you through steps including installing CRDs and the TiDB operator using Helm, and a simple TiDB cluster including monitoring services. We'll use the quick start instructions as a vehicle to talk about what makes TiDB a Kubernetes native database.

Installing the TiDB CustomResourceDefinitions. After making sure you have a Kubernetes cluster that meets the defined prerequisites such as having a default StorageClass, the first step in deploying TiDB using the operator is installing the CRDs used by the operator. This is done using an instruction such as the following (note the actual operator version number v1.3.2 may vary):

```
kubectl create -f https://raw.githubusercontent.com/pingcap/tidb-operator/
v1.3.2/manifests/crd.yaml
```

This results in the creation of several CRDs which you can observe by running the command kubectl get crd as we have done in previous chapters. We'll quickly discuss the purpose of each resource since several of them hint at additional features of interest:

- The TidbCluster is the primary resource that describes the desired configuration of a TiDB cluster. We'll look at an example below.
- The TidbMonitor resource is used to deploy a Prometheus-based monitoring stack to observe one or more TidbClusters. As we have seen with other projects, Prometheus (or at least its API) has become a de-facto standard for metrics collection for databases and other infrastructure deployed on Kubernetes.
- The Backup and Restore resources represent the actions of performing a backup or restoring from a backup. This is similar to other operators we've examined previously from the Vitess and K8ssandra projects. The TiDB operator also provides a BackupSchedule resource that can be used to configure regular backups.
- The TidbInitializer is an optional resource that you can create to perform initialization tasks on a TidbCluster including setting administrator credentials and executing SQL statements for schema creation or initial data loading.
- The TidbClusterAutoScaler is another optional resource which can be used to configure auto scaling behavior of a TidbCluster. The number of TiKV or TIDB nodes in a TidbCluster can be configured to scale up or down between minimum and maximum limits based on CPU utilization. The addition of scaling rules based on other metrics is on the project roadmap. As we discussed in Chapter 5, autoscaling is considered a feature of an operator at Level 5 or "Autopilot", the highest maturity level.

- The TidbNGMonitoring is an optional resource that configures a TidbCluster to enable continuous profiling down to the system call level. The resulting profiling data and flame graph visualizations can be observed using the TiDB Dashboard, which is deployed separately. This is typically used by project engineers looking to optimize the database, but application and platform developers may find this useful as well.
- The DMCluster resource is used to deploy an instance of the TiDB Data Migration (DM) platform that supports migration of MySQL and MariaDB database instances into a TidbCluster. It can also be configured to migrate from an existing TiDB installation outside of Kubernetes to a TidbCluster. The ability to deploy data migration services alongside a destination TidbCluster in Kubernetes managed by the same operator is a great example of what it means to develop data ecosystems in Kubernetes, a pattern that we hope to see more of in the future.

For the remainder of this section we'll focus on the TidbCluster and TidbMonitoring resources.

Installing the TiDB Operator

After installing the CRDs, the next step is to install the TiDB Operator using Helm. You'll need to add the Helm repository first before installing the TiDB Operator in its own namespace:

```
helm repo add pingcap https://charts.pingcap.org/
helm install -create-namespace --namespace tidb-admin tidb-operator pingcap/
tidb-operator --version v1.3.2
```

You can watch the resulting pods come online using kubectl get pods and referencing the tidb-admin namespace. Figure 7-2 provides a summary of the elements that you've installed up to this point. This includes Deployments to manage the TiDB Operator (labeled as tidb-controller-manager) and the TiDB Scheduler.

www.dbooks.org

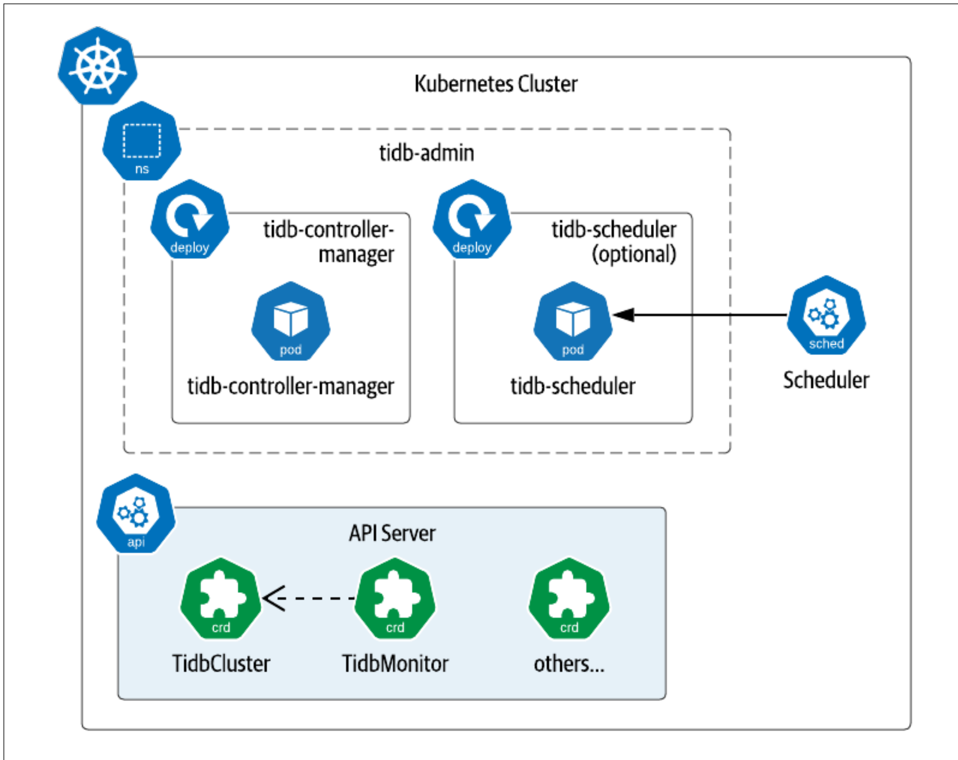
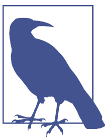


Figure 7-2. Installing the TiDB Operator and CRDs

The TiDB Scheduler is an optional extension to the Kubernetes built-in scheduler. While it is deployed by default as part of the TiDB Operator, it can be disabled. Assuming the TiDB Scheduler is not disabled, using it for a specific TidbCluster still requires opting in by setting the schedulerName property to tidb-scheduler. If this property is set, the TiDB Operator will assign the TiDB Scheduler as the scheduler that Kubernetes will use when creating TiKV, and PD Pods.



TiDB Operator Helm Chart options

This installation omits usage of a values.yaml file, but you can see the available options by running following command:

```
helm show values pingcap/tidb-operator
```

This includes the option to disable the TiDB Scheduler.

The TiDB Scheduler extends the Kubernetes built-in scheduler to add custom scheduling rules for Pods that are part of a TidbCluster, helping to achieve high availability of the database while spreading the load evenly across the available worker nodes in the Kubernetes cluster. While for many types of infrastructure, the existing mecha-

nisms Kubernetes offers for influencing the default scheduler such as affinity rules, taints and tolerations are sufficient, TiDB provides a useful example of when and how to implement custom scheduling logic. We'll look at Kubernetes scheduler extensions in more detail in Chapter 9.

Creating a TidbCluster

Once the TiDB Operator has been installed, you're ready to create a TidbCluster resource. While there are many example configurations available in the TiDB Operator GitHub repository, let's use the one referenced in the quick start guide:

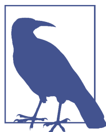
```
kubectl create namespace tidb-cluster
kubectl -n tidb-cluster apply -f https://raw.githubusercontent.com/pingcap/tidb-
operator/master/examples/basic/tidb-cluster.yaml

While the TidbCluster is being created, you can reference the contents of this
file, which look something like this (with comments and some details removed):
apiVersion: pingcap.com/v1alpha1
kind: TidbCluster
metadata:
  name: basic
spec:
  version: v5.4.0
  ...
  pd:
    baseImage: pingcap/pd
    maxFailoverCount: 0
    replicas: 1
    requests:
      storage: "1Gi"
    config: {}
  tikv:
    baseImage: pingcap/tikv
    maxFailoverCount: 0
    evictLeaderTimeout: 1m
    replicas: 1
    requests:
      storage: "1Gi"
    config:
      ...
  tidb:
    baseImage: pingcap/tidb
    maxFailoverCount: 0
    replicas: 1
    service:
      type: ClusterIP
    config: {}
```

Notice that this results in the creation of a TidbCluster named basic in the tidb-cluster namespace, with one replica each of TiDB, TiKV and PD, using the standard PingCap images for each. Additional options are used to specify the minimum

www.dbooks.org

amount of compute and storage resources required to achieve a functioning cluster. There are no TiFlash nodes included in this simple configuration.



TidbCluster API

The full list of options for a TidbCluster can be found as part of the API available in the GitHub repository. This same page includes options for the other CRDs used by the TiDB Operator. As you explore the options for these CRDs, you'll see evidence of the common practice of allowing many of the options that will be used to specify underlying resources to be overridden, for example, the Pod specification that will be set on a Deployment.

We encourage you to take the opportunity to use kubectl or your favorite visualization tool to explore the resources that are created as part of the TidbCluster, a summary of which is provided in Figure 7-3.

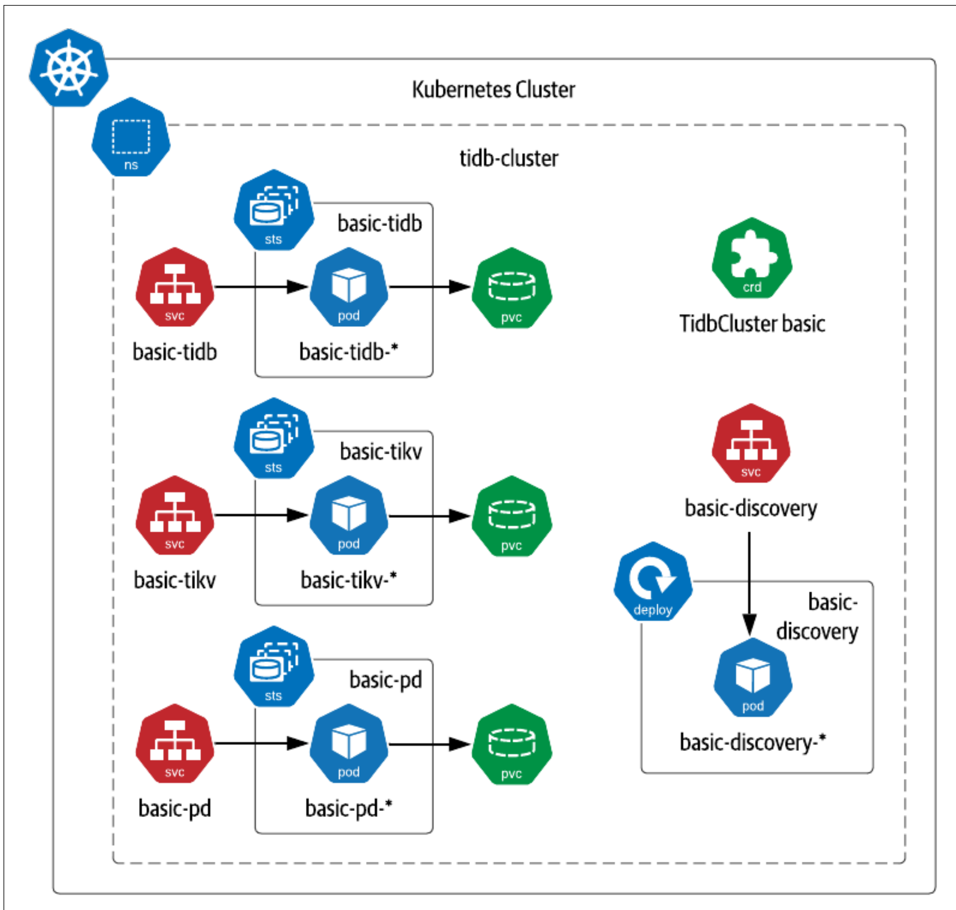


Figure 7-3. A Basic TidbCluster

As you can see in the figure, the TiDB Operator creates StatefulSets to manage the TiDB, TiKV and Placement Driver instances, allocating a PVC for each instance. As an input/output intensive application, the default configuration is to use local PersistentVolumes as the backing store.

In addition, a Deployment is created to run a Discovery Service which the various components use to learn of each other's location. The Discovery Service performs a similar role to that of etcd in other data technologies we've examined in the book. The TiDB Operator also configures services for each StatefulSet and Deployment that facilitate communication within the TiDB cluster as well as exposing capabilities to external clients.

www.dbooks.org

The TiDB Operator also supports the deployment of a Prometheus monitoring stack that can manage one or more TiDB clusters. You can add monitoring to the cluster created previously using the following command:

```
kubectl -n tidb-cluster apply -f https://raw.githubusercontent.com/pingcap/tidb-
operator/master/examples/basic/tidb-monitor.yaml
```

While this is deploying, let's examine the contents of the tidb-monitor.yaml configuration file:

```
apiVersion: pingcap.com/v1alpha1
kind: TidbMonitor
metadata:
  name: basic
spec:
  replicas: 1
  clusters:
  - name: basic
  prometheus:
    baseImage: prom/prometheus
    version: v2.27.1
  grafana:
    baseImage: grafana/grafana
    version: 7.5.11
  initializer:
    baseImage: pingcap/tidb-monitor-initializer
    version: v5.4.0
  reloader:
    baseImage: pingcap/tidb-monitor-reloader
    version: v1.0.1
  prometheusReloader:
    baseImage: quay.io/prometheus-operator/prometheus-config-reloader
    version: v0.49.0
  imagePullPolicy: IfNotPresent
```

As you can see, the TidbMonitor resource can point to one or more TidbClusters. This TidbMonitor is configured to manage the cluster named basic you created above. The TidbMonitor resource also allows you to specify the versions of Prometheus, Grafana and additional tools that are used to initialize and update the monitoring stack. If you examine the contents of the tidb-cluster namespace, you'll see additional workloads that have been created to manage these elements.

TiDB uses the Prometheus stack in a similar way to the K8ssandra project we discussed in Chapter 6. In both of these projects, the Prometheus stack is supported as an optional extension to provide a monitoring capability you can use with very little customization. The configurations and provided visualizations focus on the key metrics that drive awareness of database health. Even if you are already managing your own monitoring infrastructure or using a third party SaaS solution, the configurations and charts can give you a head start on incorporating database monitoring into the rest of your observability approach.

A roadmap for cloud-native databases on Kubernetes

With Dongxu (Ed) Huang, Co-founder & CTO, PingCAP

TiDB was created out of the experience of maintaining a storage system for a large internet company who ran an Android App Store. The distributed MySQL sharding cluster we were using was innovative at the time but also too hard to maintain. With manual sharding, you cannot do cross shard joins, or transactions. It's really painful for the application developer. The Google Spanner and F1 papers provided the inspiration for future databases like TiDB with scalability and high availability, consistency, full featured SQL and global transaction support. From the application developers perspective, it should feel like going back to the old days of single node development, but now with horizontal scalability.

The problem statement was straightforward. We wanted to provide scalable OLTP queries with reduced migration cost and an easy to use MySQL interface. At that time, there was no open source implementation of Spanner, so we started to build TiKV and donated it to the CNCF. As more and more users started running OLAP queries on top of their real time data in TiDB, we expanded our OLAP capability to create a hybrid approach called HTAP. The TiFlash engine that supports OLAP queries has recently been open sourced as well.

The TiDB architecture does have some cloud-native aspects from its original design, especially since it has a shared nothing architecture. However, from another point of view, it is not very cloud native. There is a higher standard to be called a cloud native database. A cloud native database should make maximum use of the infrastructure your cloud vendor provides, for example, a storage engine that leverages S3, or uses a cloud's serverless features. By this standard, the only cloud native database is Snowflake. The approach that customers need is this: pay for only what you use. If you have to buy it by the node, it's not serverless.

It's more accurate to refer to TiDB as a Kubernetes native database. When we saw the first etcd operator released in 2016, we were inspired to create our own operator. At that time, Kubernetes was not as mature as it is today. We didn't have CRDs, just third party resources. We had to build our own scheduler to make sure we could handle failover correctly. The hardest part was handling local storage. Kubernetes was not designed from the database engineer's point of view. At Google, they didn't focus on providing access to local disk for databases, since most of their systems were built on top of columnar stores. They didn't care about local state, but as a database engineer you have to be very careful with your use of local disk. Since there was no local storage API in Kubernetes when we started, we wrote our own controller to manage local disks. We put a lot of resources into this effort. It was very complicated and might have been the wrong decision.

Today things are a lot better. Kubernetes networking, StatefulSets and CRDs are mature and good enough for application developers and database engineers to use. At

www.dbooks.org

PingCAP we use Kubernetes to run our managed service on public clouds. We have a lot of users and it's very stable. We can work with it.

In the future of cloud native architectures, storage and compute will be separated more and more clearly over time. In the past, you would never have built a database on top of remote storage. But now it might be time to give up doing persistence on local disks. We're working on a new storage engine for TiDB built on top of shared storage.

One area where Kubernetes needs to improve is support for multi-tenancy. Today, building a multi-tenant application in Kubernetes is really hard. Namespaces are not enough of an abstraction to support multi-tenancy. Similar to control groups (cgroups) for Linux, Kubernetes needs a virtualized cluster or some other multi-tenancy mechanism within the cluster. There is a Kubernetes SIG which is looking into multi-tenancy, and the work on virtual clusters (VClusters) is really promising.

A second area where Kubernetes could improve is better support for hypervisors. When you have large clusters, you don't want to have virtualization on top of hardware and then run Kubernetes on top of that. The Kubernetes community could be more ambitious and put more resources toward embracing hypervisors such as Cloud Hypervisor.

For its part, the database world really needs to be more focused on Kubernetes. DevOps and application engineers are the mainstream Kubernetes community, but the database folks are outside of that. Most database operators are not written by experienced DBAs. Once you get beyond deploying the database, tuning a database is a hard job, you have a lot of maintenance to do. Once you put a database in Kubernetes pods, tuning it requires going inside the pods. For the DBA or DevOps engineer, that's the most tricky part. As a user, you should always prefer an operator provided by the database vendor. If you're a database vendor, you need to help the user by making it easier to tune in the Kubernetes environment, not just deployment or upgrades. The real world is not like running a demo.

As you can see, TiDB is a database with a flexible, extensible architecture that has been designed with cloud native principles in mind. It also has a strong bias toward being able to deploy and manage a database effectively in Kubernetes and has provided us with some valuable insights on what it means to be Kubernetes native. Consult the TiDB documentation for more information on features such as deploying to multiple Kubernetes clusters.

Serverless Cassandra with DataStax Astra DB

Since the advent of cloud computing in the early 2000s, public cloud providers and infrastructure vendors have made continual advances in commoditizing various layers of our architectural stacks as service offerings. This trend began with offering

compute, network and storage as *Infrastructure as a Service (IaaS)* and proceeded into other trends including *Platform as a Service (PaaS)*, *Software as a Service (SaaS)*, and *Functions as a Service (FaaS)*, sometimes conflated with the term *serverless*.

Most pertinent to our investigation here is the emergence of managed data infrastructure offerings known as *Database as a Service (DBaaS)*. This category includes:

- Traditional databases offered as a managed cloud service, such as Amazon Relational Database Service (RDS) and PlanetScale
- Cloud databases like Google BigTable, Amazon Dynamo, and Snowflake that are available only as cloud offerings
- Managed NoSQL or NewSQL databases that are also available as open source projects, for example MongoDB Atlas, DataStax Astra DB, TiDB, and Cockroach DB

Over the past several years, many of the vendors behind these DBaaS services have begun migrating onto Kubernetes in order to automate operations, manage compute resources more efficiently, and make their solutions portable across clouds. DataStax was one of several vendors that began offering Cassandra as a Service. These typically used an architecture based on running traditional Cassandra clusters in a cloud environment, with various "glue code" to integrate aspects like networking, monitoring, and management that didn't quite fit target deployment environments like Kubernetes and public cloud IaaS. These include techniques like using sidecars to collect metrics and logs, or deploying Cassandra nodes using StatefulSets to manage scaling up and down in an orderly fashion.

Even with these workarounds for running in Kubernetes, Cassandra's monolithic architecture doesn't readily promote the separation of compute and storage, which can lead to some awkwardness when scaling. You scale up a Cassandra cluster by adding additional nodes, where each node has the following capabilities:

- *Coordination* - receiving read and write requests and forwarding them to other nodes as needed to achieve the requested number of replicas (also known as *consistency level*).
- *Writing and Reading* - writing data to in-memory cache (memtables) and persistent storage (SSTables), and reading it back as needed
- *Compaction and Repair* - since Cassandra is a LSM-tree database, it does not update data files once they are written to persistent storage. Compaction and repair are tasks that run in the background as separate threads. Compaction helps Cassandra stay performant by consolidating SSTables written at different times, ignoring obsolete and deleted values. Repair is the process of comparing stored values across nodes to ensure consistency.

www.dbooks.org

Each node in a Cassandra cluster implements all of these capabilities and consumes equivalent compute and storage resources. This makes it difficult to scale compute and storage independently and can lead to situations where a cluster is overprovisioned in compute or storage resources.

In 2021, DataStax published a paper entitled DataStax Astra DB: Designing a Serverless Cloud-Native Database-as-a-Service that describes a different approach. Astra DB is a version of Cassandra that has been refactored into microservices in order to allow more fine grained scalability and to take advantage of the benefits of Kubernetes. In fact, Astra DB is not only Kubernetes native, it is essentially a Kubernetes-only database. Figure 7-4 shows the Astra DB architecture at a high level, broken into a control plane, data plane, and supporting infrastructure.

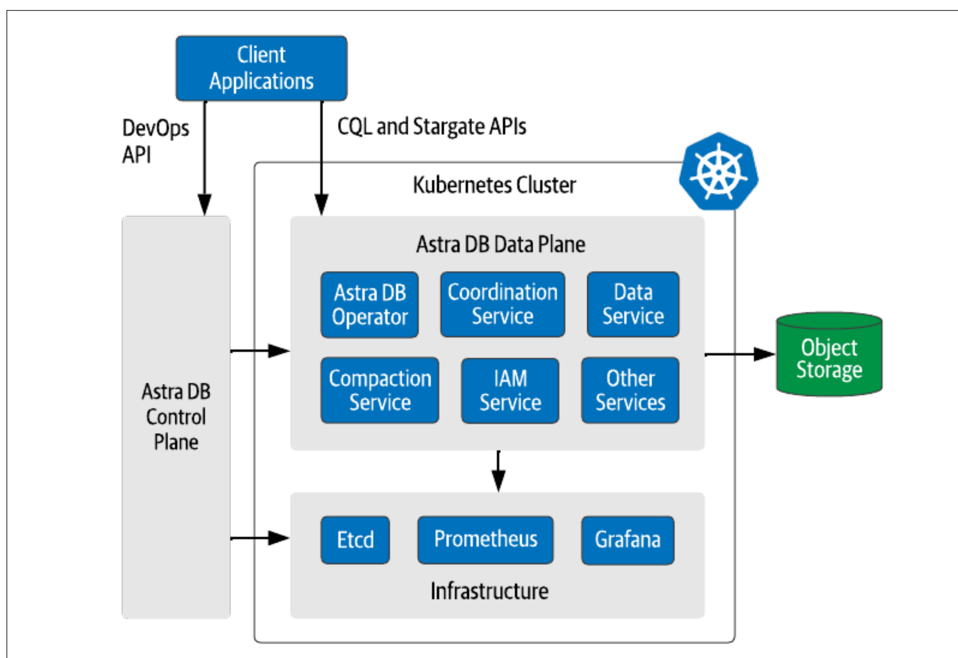


Figure 7-4. Astra DB Architecture

Let's take a quick overview of the layers in this architecture:

Astra DB Control Plane

The control plane is responsible for provisioning Kubernetes clusters in various cloud provider regions. It also provisions Astra DB clusters within those Kubernetes clusters and provides the APIs that allow clients to create and manage databases, either through the Astra DB web application, or programmatically through the DevOps API. The blog post How We Built the DataStax Astra DB Control

Plane describes the architecture of the control plane and how it was migrated to be Kubernetes native.

Astra DB Data Plane
: The data plane is where the actual Astra DB databases run. The data plane consists of multiple microservices which together provide the capabilities that would have been a part of a single monolithic Cassandra node. Each database is deployed in a Kubernetes cluster in a dedicated namespace and may be shared across multiple tenants, as described in more detail below.

Astra DB Infrastructure
: Each Kubernetes cluster also contains a set of infrastructure components that are shared across the Astra DB databases in that cluster, including Etcd, Prometheus, and Grafana. Etcd is used to store metadata including the assignment of tenants to databases and database schema for each tenant. It also stores information about the cluster topology, replacing the role of gossip in the traditional Cassandra architecture. Prometheus and Grafana are deployed in a similar way as described in other architectures in this book.

Now let's dig more into a few of the microservices in the data plane:

Astra DB Operator
: The Astra DB Operator manages the Kubernetes resources required for each database instance as described by a DBInstallation custom resource, as shown in Figure 7-5 below.

Coordination Service
: The Coordination Service is responsible for handling application queries including reads, writes, and schema management, Each Coordination Service is an instance of Stargate that exposes endpoints for CQL and other APIs, with an Astra DB-specific plugin that enables it to route requests intelligently to Data Service instances to actually store and retrieve data. Factoring this compute-intensive routing functionality into its own microservice enables it to be scaled up or down based on query traffic, independent of the volume of data being managed.

Data Service
: Each Data Service instance is responsible for managing a subset of the data for each assigned tenant based on its position in the Cassandra token ring. The Data Service takes a tiered approach to data storage, maintaining in-memory data structures such as memtables, using local disk for caching, commit logs and indexes, and object storage for longer term persistence of SSTables. The usage of object storage is one of the key differentiators of Astra DB from other databases we've examined so far, and we'll examine other benefits of this approach throughout this section.

www.dbooks.org

Compaction Service

The Compaction Service is responsible for performing maintenance tasks including compaction and repair on SSTables in object storage. Compaction and repair are compute-intensive tasks that experienced Cassandra operators have historically scheduled for off-peak hours in order to limit their impact on cluster performance. In Astra DB, these tasks can be performed at any time the need arises without impacting query performance. The work is handled by a pool of Compaction Service instances which can scale up or down independently to generate repaired, compacted SSTables which are immediately accessible to Data Services.

Iam Service

All incoming application requests are routed through the Identity and Access Management (IAM) Service, which uses a standard set of roles and permissions defined in the control plane. While Cassandra has long had a pluggable architecture for authentication and authorization, factoring this out into its own microservice allows for more flexibility and support for additional providers such as Okta.

The data plane includes additional services which have been omitted from Figure 7-4 for simplicity, including a Commitlog Replayer Service for recovery of failed Data Service instances, and an Autoscaling Service which uses analytics and machine learning to recommend to the operator when to scale the number of instances of each service up or down.

Figure 7-5 shows what a typical DBInstallation looks like in terms of Kubernetes resources, . Let's walk through a few typical interactions focusing on individual instances of key services to demonstrate how each resource plays its part.

- A Kubernetes ingress is configured for each cluster to manage incoming requests from client applications (1) and route requests to Coordinator Services by tenant using a Kubernetes Service (2).
- The Coordinator Service (3) is a stateless service managed by a Deployment which delegates authentication and authorization checks on each call to the Iam Service (4).
- Authorized requests are then routed to one or more Data Services based on the tenant, again using a Kubernetes Service (5).
- Data Services (6) are managed using StatefulSets, which are used to assign each instance to a local PersistentVolume used for managing intermediate data files such as the commit log, which is populated immediately on writes. When possible, reads are served directly from in-memory data structures.
- As is typical for Cassandra and other LSM tree storage engines, the Data Service occasionally writes SSTable files out to a persistent store (7). For Astra DB, that persistent store is an externally object store managed by the cloud provider for

high availability. A separate object storage bucket is used per tenant to ensure data privacy.

- The Compaction Service can perform compaction and repair on SSTables in the object store asynchronously (8), with no impact to write and read queries.

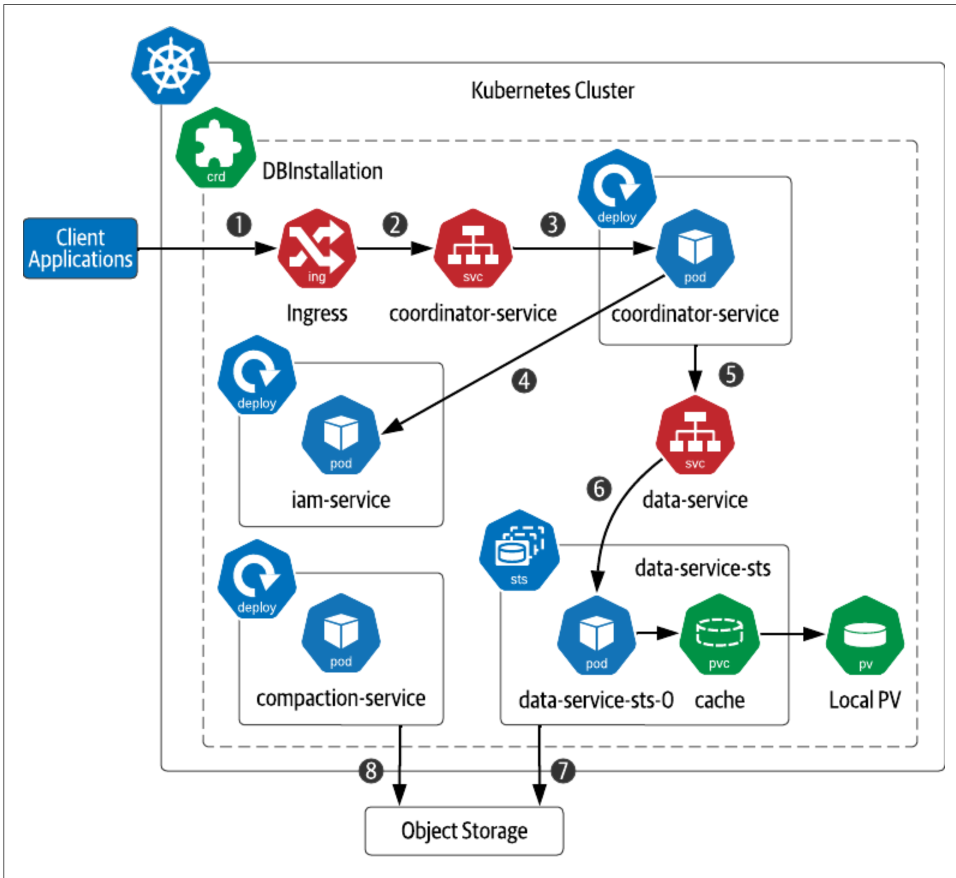


Figure 7-5. Astra DB Cluster in Kubernetes

Astra DB also supports multi-region database clusters, which by definition span multiple Kubernetes clusters. Coordinator and Data Services are deployed across datacenters (cloud regions) and racks (availability zones) using an approach similar to that described for K8ssandra in Chapter 6.

Astra DB's microservice architecture allows it to make more optimal use of compute and storage resources and isolate compute-intensive operations, leading to overall cost savings to operate Cassandra clusters in the cloud. These cost savings are extended by the addition of multi-tenant features that allow each cluster to be shared

www.dbooks.org

across multiple tenants. The Astra DB whitepaper describes a technique called *shuffle sharding* which is used to match each tenant to a subset of the available Coordinator and Data Services, effectively creating a separate Cassandra token ring per tenant. As the population of tenants in an Astra DB instance change, this topology can be easily updated to rebalance load without downtime, and larger tenants can be configured to use their own dedicated databases (DBInstallations). This approach helps minimize cost while meeting service level agreements (SLAs) for performance and availability.

What is a serverless database?

With Jake Luciani, Engineering Leader, DataStax

Cassandra has always been considered a cloud native database, but it's not Kubernetes native. The K8ssandra project represents a first step in the direction of making Cassandra more Kubernetes native. It's a systematic way to run Cassandra on Kubernetes in a more traditional cloud native way, but it represents more of a "lift and shift" approach. The Astra DB approach is more like throwing all of our bags on the Kubernetes bus. It's a version of Cassandra that you can't run without Kubernetes.

We realized early on in the process of building Astra DB that we had to make some modifications to Cassandra's architecture to make it work in an even more cloud native way. The cost of running stateful systems in the cloud can get very expensive if you do it the wrong way. If you focus on optimizing for cost, you'll actually end up with the most cloud native solution, because the services that are cheap in the clouds are the ones that have become the most commoditized. They're also the most hardened parts of the system. By standing on the shoulders of proven cloud technologies like object storage and etcd, you'll end up with a more reliable solution.

We often refer to Astra DB as a serverless database, which came from the original inspiration for the project: "How do we make Cassandra more serverless?" Serverless is really just a term that describes techniques engineers use in the cloud to make applications more scalable and more stateless. The first breakthrough was separating compute from storage. Storing SSTables as immutable data on the object store allows us to scale our IOPS the same way we scale our processing engine. You can remove any component and it doesn't matter to the system. In the same way you can scale up lambda serverless functions, it's the same idea with your database.

The topology is completely ephemeral, just like Cassandra; it can change on the fly. Cassandra has traditionally used a gossip type protocol to coordinate topology and replicate state across nodes in an eventually consistent way. But since Cassandra was first built, systems like etcd have come along that do a great job of maintaining metadata about schema and topology in a transactional way. Etcd is a stateful service in its own right, but we really only use it as a way to transition from one state to the next. The object store is ultimately the source of truth for the entire system. You can lose an entire Kubernetes cluster with all of the databases running on it, rebuild the cluster, wipe the disks, and bring the whole system back. This is a great feeling when you go

to sleep at night. Currently we have to run our own etcd inside of Kubernetes, even though Kubernetes runs its own etcd cluster. It would be great if we could utilize that infrastructure that's already running. Instead we've had to build up our own etcd expertise to make sure we know how to run it.

We use StatefulSets to manage the Cassandra nodes in each availability zone. StatefulSets provide the exact behavior we need in terms of scaling up and down in a fixed order. Even though typically only use local ephemeral disks and the local path provisioner, we're not precluded from using a PVC with persistent storage if we needed to, it would just be more expensive. In order to perform upgrades, we create an entire StatefulSet with new Cassandra nodes. Once all of the nodes have joined, we can delete the old StatefulSet. We treat the StatefulSets as immutable infrastructure, throwing them away and starting over.

One big problem with data on Kubernetes is the rough edges in working with attached disks. Many databases need to stripe disks before using them. On Kubernetes, this means mounting volumes as raw disks and then striping them during pod startup. To scale the available IOPS, you have to attach more raw disks and stripe them as well. We avoided this problem in Astra DB by going all in on object storage and local ephemeral disks. The ephemeral disks are just a cache of what's in the object storage, but they give us the IOPS we need. Cassandra uses a Log-Structured Merge tree (LSM tree) style of storage engine, similar to RocksDB. This provides a great opportunity for a cloud native separation of disk and storage, because the data files are immutable. We never need to perform in-place updates of data on disk, which works out well because object storage doesn't allow that anyway. Compaction can run as a separate process and scale on its own right, which keeps the reads fast.

Another challenge with Kubernetes is choosing the right VM types and figuring out how to map pods to them efficiently. Unfortunately, the Kubernetes APIs are really decoupled from the underlying cloud provider capabilities. You have to do a lot of math in your head in order to set up quotas and node groups, and we haven't even gotten to disks. There's a massive market opportunity out there for someone who can solve this problem.

When you're running a SaaS, the way you lower prices and keep margins is by being as efficient as possible. For us this means multi-tenancy and the ability to shift resources between tenants based on usage. We use a giant shared pool of pods and resources, which gives us the ability to move users and their data to different parts of the fleet. This allows us to provide a usage based pricing model for developers who just want to use it and go, and it empowers them to build some really cool applications.

Zooming back out, Kubernetes did a great job with stateless services from the beginning, but stateful workloads are harder. People in Kubernetes want to solve this with changes to Kubernetes, and people who build infrastructure, want to solve this in the infrastructure. We'll get there eventually through a combination of the two. In the meantime we're circumventing the issue by using the immutable systems that work

www.dbooks.org

well on Kubernetes and moving the state out into object storage. This is the way open source technology works. People try things and make progress. Adopting a new architecture can be a big risk, but once you do, the payoff can be huge.

In this section we've focused on the architecture Astra DB uses to provide a multi-tenant, serverless Cassandra that embodies both cloud native and Kubernetes native principles. This continues the tradition of the Amazon Dynamo and Google BigTable papers in generating public discussion around novel database architectures. In addition, several open source projects mentioned in this book including Cass-Operator, K8ssandra, and Stargate trace their origins to Astra DB. There is a lot of innovation going on in areas such as the core database, control plane, change data capture, streaming integration, data migration and more, so look for more open source contributions and architecture proposals from this team in the future.

What to look for in a Kubernetes Native Database

After everything you've learned in the past few chapters about what it takes to deploy and manage various databases on Kubernetes, we are in a great position to define what you should look for in a Kubernetes native database.

Basic requirements

Following our cloud native data principles, we'll outline a few areas that should be considered basic requirements.

Maximum leverage of Kubernetes APIs

The database should be as tightly integrated with Kubernetes APIs as possible, for example, using PersistentVolumes for both local and remote storage, using Services for routing rather than maintaining lists of IPs of other nodes, and so on. Kubernetes extension points described in Chapter 5 should be used to supplement built-in Kubernetes functionality.

There may be areas where the existing Kubernetes APIs don't provide the exact behavior required for a given database or other application, as demonstrated by the creation of alternate StatefulSet implementations by the Vitess and TiDB projects. In these cases, every effort should be made to donate improvements back to the Kubernetes project.

Automated, declarative management via operators

Databases should be deployed and managed on Kubernetes using operators and custom resources. Operators should serve as the primary control plane elements for managing databases. While it is arguably helpful to have command line tools or kubectl extensions that allow database administrators to intervene manually to optimize database performance and fix issues, these are ultimately functions that

should be performed by an operator as it achieves the higher levels of maturity discussed in Chapter 5.

The goal should be that all required changes to a database can be accomplished by updating the desired state in a custom resource and letting the operator handle the rest. We'll be in a great place when we can configure a database in terms of service-level objectives such as latency, throughput, availability, and cost per unit. Operators can determine how many database nodes are needed, what compute and storage tiers to use, when to perform backups, and so on.

Observable through standard APIs

We're beginning to see common expectations for observability for data infrastructure on Kubernetes in terms of the familiar triad of metrics, logs, and tracing. The Prometheus-Grafana stack is somewhat of a de-facto standard for metrics collection and visualization, with exposure of metrics from database services using the Prometheus format a minimum criteria. Projects providing Prometheus integration should be flexible enough to provide their own dedicated stack, or push metrics to an existing installation shared with other applications.

Logs from all database application containers should be pushed to stdout (using sidecars if necessary) so they can be collected by log aggregation services. While it may take longer to see adoption for tracing, the ability to follow individual client requests through application calls down into the database tier through APIs such as Open Tracing will be an extremely powerful debugging tool for future cloud native applications.

Secure by default

The Kubernetes project itself provides a great example of what it means to be secure by default, for example by only exposing access to ports on Pods and containers when specifically enabled, and by providing primitives like Secrets that we can use to protect access to login credentials or sensitive configuration data.

Databases and other infrastructure need to make use of these tools and adopt industry standards and best practices for zero trust, including changing default administrator credentials, limiting exposure of application and management APIs. Exposed APIs should prefer encrypted protocols such as HTTPS. Data stored in PersistentVolumes should be encrypted, whether this encryption is performed by the application, the database, or the StorageClass provider. Audit logs should be provided as part of application logging, especially with respect to actions that configure user access.

As you can see, basic requirements and more advanced expectations for what it means to be Kubernetes native have begun to solidify. But what comes next?

www.dbooks.org

The Future of Kubernetes Native

We're starting to see common patterns within projects deploying databases on Kubernetes that could point to where things are headed in the future. These are admittedly a bit more fuzzy, but let's try to bring a couple of them into focus.

Scalability through Multi-dimensional Architectures

You may have noticed the repetition of several terms throughout the past few chapters such as multi-cluster, multi-tenancy, microservices, and serverless. A common thread uniting these terms is that they represent architectural approaches to scalability, as shown in Figure 7-5.

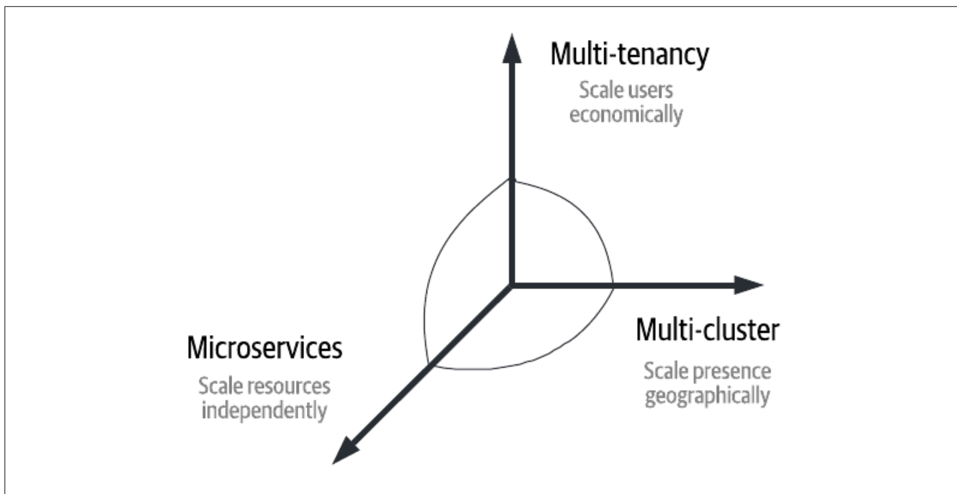


Figure 7-6. Architectural approaches for scaling in multiple dimensions

Consider how each of these approaches provides an independent axis for scalability. The visualization in Figure 7-5 depicts the impact of your application as a three dimensional surface that grows as you scale along each axis.

- **Microservice architectures** break the various functions of a database into independently scalable services. The serverless approach builds on this, encouraging the isolation of persistent state to a limited number of stateful services or even external services as much as possible. Kubernetes storage APIs in the Persistent Volume subsystem make it possible to leverage both local and networked storage options. These trends allow a true separation of compute and storage and scale these resources independently.
- **Multi-cluster** refers to the ability to scale an application across multiple Kubernetes clusters. Along with related terms like multi-region and multi-datacenter, this implies expanding the geographic footprint of the capabilities provided. This

distribution of capability has positive implications for meeting users where they are with minimum latency, as well as disaster recovery. As we discussed in Chapter 6, Kubernetes has historically not been as strong in its support for cross-cluster networking and service discovery. It will be interesting to track how databases and other applications take advantage of expected advances in Kubernetes federation in the coming years.

- **Multi-tenancy** is the ability to share infrastructure between multiple users in order to achieve the most efficient use of resources. As the public cloud providers have demonstrated in their IaaS offerings, a multi-tenant approach can be very effective at providing users a low-cost, low-risk access to infrastructure for innovative new projects, and then providing additional resources as these applications grow. There is great potential value in adopting a a multi-tenant approach for data infrastructure as well, so long as security guarantees are properly met and there is a seamless transition path to dedicated infrastructure for high-volume users before they become "noisy neighbors".

While you may not have immediate need for all three of these axes of scalability for applications or data infrastructure you're building, consider how growing in each of them can enhance the overall value you're offering the world.

Community-focused innovation through open source and cloud services

Another pattern you may have noticed in our narrative is the continual innovation loop between open source database projects and DBaaS offerings. PingCap took the open source MySQL and Clickhouse databases, created a database service leveraging Kubernetes to help it manage the databases at scale, and then released open source projects including TiDB and TiFlash. DataStax took open source Cassandra, factored it into microservices, added an API layer and deployed it on Kubernetes for its Astra DB, and has created multiple open source projects including Cass-Operator, K8ssandra, and Stargate. In the spirit of Dynamo, BigTable, Calvin and other papers, these companies have open sourced their architectures as well.

This innovation loop mirrors that of the larger Kubernetes community, in which the major cloud providers and storage vendors have helped drive the maturation of the core Kubernetes control plane and Persistent Volume subsystem, respectively. It's interesting to observe that the highest momentum and fastest cycle time occurs within innovation loops that center around cloud services, rather than around the classic open core model focused on enterprise versions of open source projects.

As a software vendor, providing a cloud service allows you to iterate and evaluate new architectures and features more quickly. Flowing these innovations back to open source allows you to grow adoption by supporting a flexible consumption model. Both "run it yourself" and "rent it from us" become legitimate deployment options for your customers, with the ability to flex between approaches for different use cases.

www.dbooks.org

Customers gain confidence in the overall maturity and security of your technology, knowing that the open source version they can inspect and contribute to is largely the same as what you are running in your DBaaS.

A final side effect of these innovation trends is an implicit pull toward proven architectures and components. Consider these examples:

- Etcd is used as a metadata store across multiple projects we've examined in this book, including Vitess and Astra DB.
- TiDB leverages the architecture of F1, implemented the Raft consensus protocol, and extended the Clickhouse columnar store.
- Astra DB leverages both the Persistent Volume subsystem and S3-compliant object storage.

Instead of inventing new technologies to solve problems like metadata management and distributed transactions, these projects are investing their innovation in new features, developer experience and the scalability axes we've examined in this chapter.

Summary

In this chapter we've taken a deep look at TiDB and Astra DB in order to search out what makes them Kubernetes native. What was the point of this exercise? Our hope is that this analysis provides a deeper understanding to help consumers ask more insightful questions about the data infrastructure they are consuming, and to help those building data infrastructure and ecosystems to create technology that meets those expectations. We believe that data infrastructure that is not only cloud native but Kubernetes native will lead to the best outcomes for everyone in terms of performance, availability, and cost.

www.dbooks.org

CHAPTER 8
Streaming Data on Kubernetes

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 8th chapter of the final book. The GitHub repo is 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 you think about data infrastructure, persistence is the first thing that comes to mind for many—storing the state of running applications. Accordingly, our focus up to this point has been on databases and storage. It's now time to consider the other aspects of the cloud native data stack.

For those of you managing data pipelines, streaming may be your starting point, with other parts of your data infrastructure being of secondary concern. Regardless of your starting place, data movement is a vitally important part of the overall data stack. In this chapter, we'll examine how to use streaming technologies in Kubernetes to share data securely and reliably in your cloud native applications.

Introduction to Streaming

In Chapter 1, we defined streaming as the function of moving data from one point to another and, in some cases, processing data 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. Data was transferred slowly but deliberately, which resembled something closer to postal mail. Messaging infrastructure put data in a place where it could be read asynchronously, in order, with delivery guarantees. This met a critical need when using more than one computer and is one of the foundations of distributed computing.

Modern application requirements have evolved from 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 the creation of data 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

Let's take a closer look at each of these areas.

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 to efficiently plan how data flows from producer to consumer. For example, "Does my consumer need exclusive access?" The answer will drive which system fits the requirements. Figure 8-1 shows two of the most common choices in streaming systems: Point to point and publish/subscribe.

www.dbooks.org

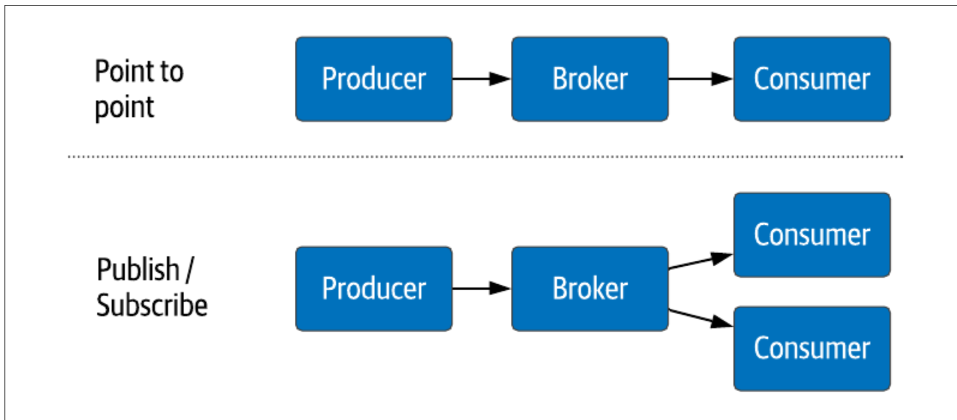


Figure 8-1. Delivery Types

Point to point
: In this data flow, 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 de-couple direct connections from producer to consumer. It serves as an excellent feature for resilience as consumers can be removed and added with no data loss. At the same time, the broker maintains the order and last message read, addressable by the consumer using an offset.

Publish / Subscribe (pub/sub)
: In this delivery method, 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 typical delivery types are shown in Figure 8-2: at-most-once, at-least-once, and exactly once. The diagram shows the important relationship between when the producer sends a message and the expectation of how the consumer receives the message.

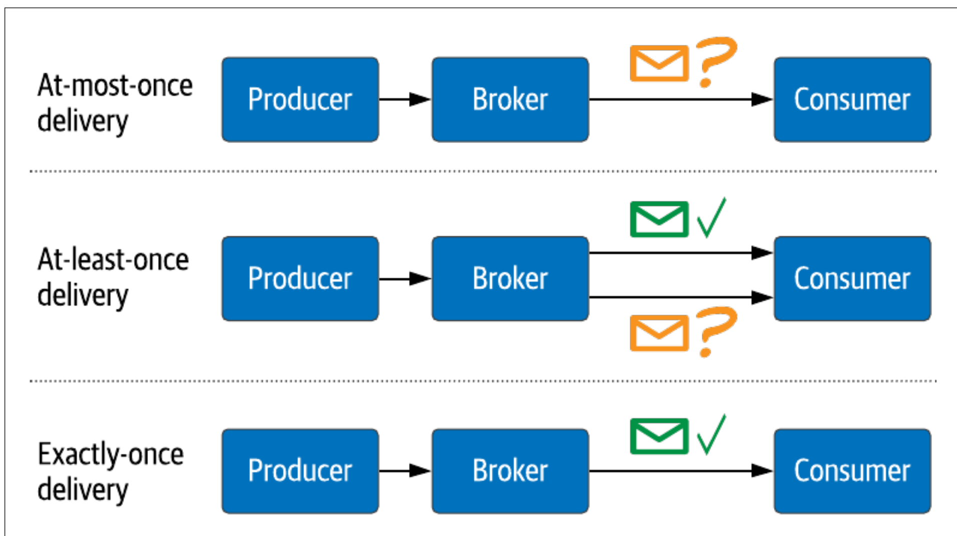


Figure 8-2. Delivery Guarantees

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 redelivery 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 more significant burden on the broker to maintain, so you will need to adjust the resources allocated to the broker and your expected throughput.

www.dbooks.org

You should exercise care in selecting 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

Many streaming technologies are 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. Still, 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 price 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 8-3.

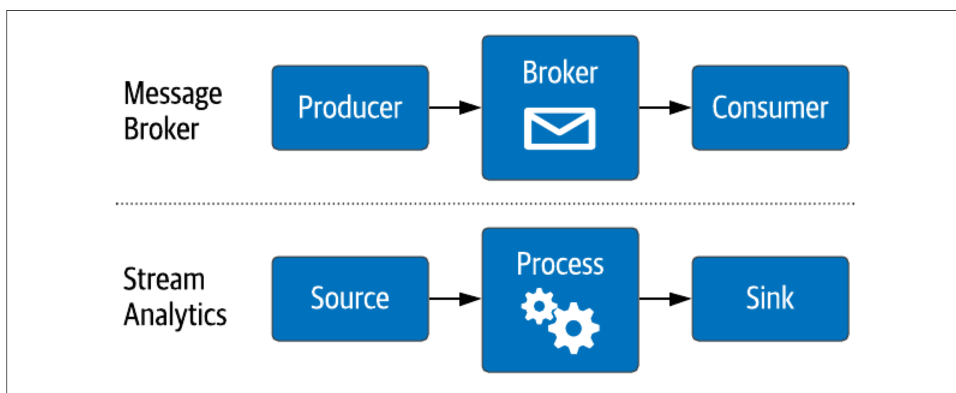


Figure 8-3. Streaming Types

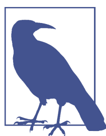
Message broker
: The simplest form of streaming technology that facilitates the moving of data from one point to another with one or more of the delivery methods and guarantees listed above. It's easy to discount this feature's simplistic appearance, but it's the backbone of modern cloud native applications. 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. In contrast, 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

www.dbooks.org

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. Creating custom ways of moving data puts a huge burden on application and infrastructure teams. Data streaming tools are built for a specific purpose, with large communities of users and vendors to aid in your success.

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 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 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 of uptime 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 greatly impact 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 focusing on 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.

www.dbooks.org

Disaster Recovery Planning

Moving data efficiently is an important problem to solve but just as important is how to manage inevitable failure. Without understanding your data flows and durability requirements, you can't just rely on Kubernetes to handle recovery. Disaster recovery 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 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 to 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 was mostly 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. There are additional projects, like Apache Kafka or RabbitMQ, that may suit your application's needs, but they will 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 8-4. Deployment options give operators the flexibility to install and scale a running Pulsar cluster based on use case and workload. It was also designed with multi-tenancy in mind, making 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. 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. We'll walk through these next.

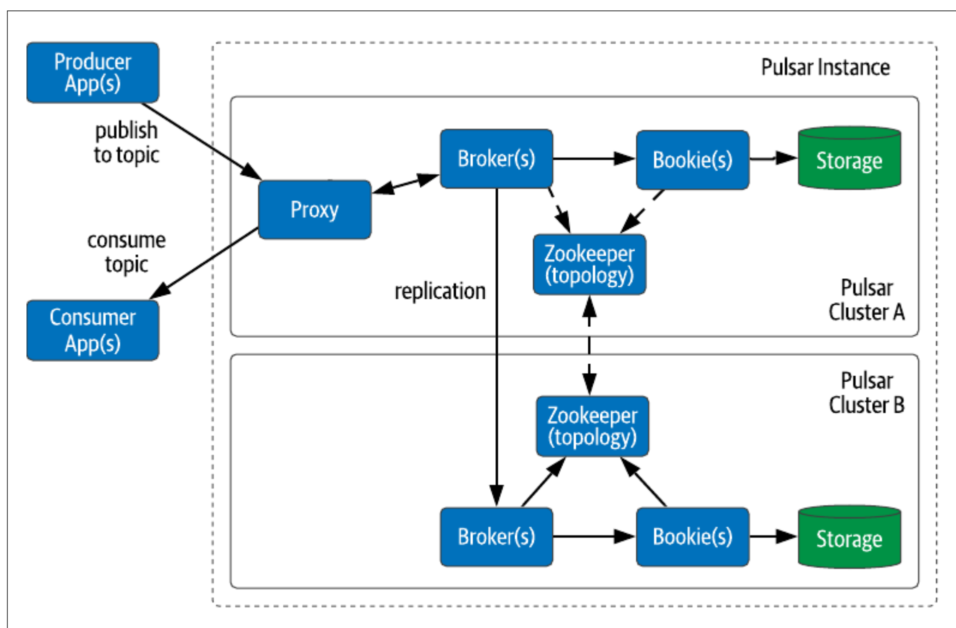


Figure 8-4. Apache Pulsar™ Architecture

Broker (Compute)

Producers and consumers pass messages via the broker, a stateless cluster component. This means it is purely a compute scaling unit and can be dynamically allocated based on the number of tenants and connections. Brokers maintain an HTTP endpoint used for client communication which presents a few options for network traffic in a Kubernetes deployment. When multiple clusters are used, the brokers support replication between clusters in the instance. Brokers can run in a memory-only configuration, or with bookies when message durability is required.

www.dbooks.org

Apache Bookkeeper™ (Storage)
: The Bookkeeper project provides infrastructure for managing distributed write-ahead logs. In Pulsar, the individual instances used are called *bookies*. The storage unit is called a *ledger*; each topic can have one or more ledgers. Multiple bookie instances provide load-balancing and failure protection. They also offer storage tiering functionality, allowing operators to offer 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, an append-only data structure. Bookies provide a single reference to the ledger but manage the replication 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, similar to how 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 like 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)
: Since Pulsar Functions operate independently and consume their own compute resources, we chose not to include them in Figure 8-4. However, they're worth mentioning in this context because Pulsar Functions work in conjunction with the message broker. When deployed, they take data from a topic, alter it with user code, and return it to a different topic. The component added to a Pulsar cluster is the worker, which accepts function runtimes on an ad-hoc basis. Operators can 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. Since every user will have unique needs, we recommend you check the official documentation for the most complete and up-to-date information on installing Pulsar in Kubernetes before reading this section. The examples within this section will take a closer

look at the choices available and how they pertain to different cloud native application use cases to help inform your decision-making.

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

```
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 you will want to tune the values for your specific environment. The examples directory has various deployment scenarios. If you choose the default installation as described in the TBD file, you'll have a set of resources like that shown in Figure 8-5. As you can see, the installation wraps the proxy and brokers in Deployments and presents a unified service endpoint for applications.

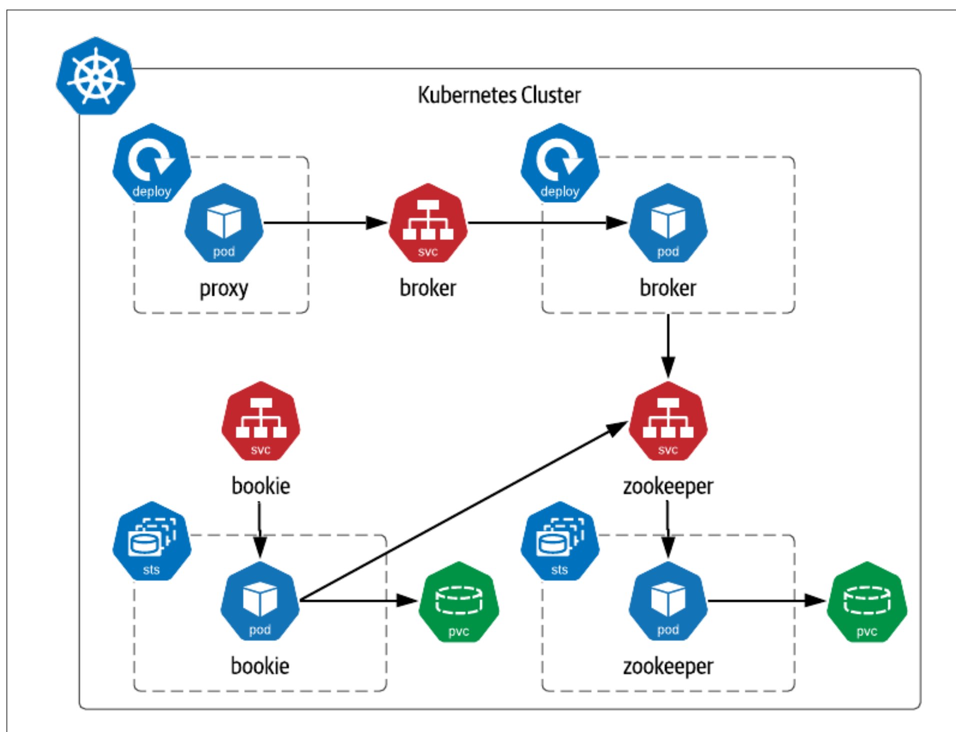


Figure 8-5. - A Simple Pulsar Installation on Kubernetes

Affinity is a mechanism built into Kubernetes to create rules for which pods can and cannot be co-located on the same physical node (if needed, refer back to the more in-

www.dbooks.org

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 not allowing 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. Still, more replicas can be added and configuration updated via Helm as traffic increases.

```
zookeeper:
  replicaCount: 1
bookkeeper:
  replicaCount: 1
broker:
  replicaCount: 1
proxy:
  replicaCount: 1
```

You now have a foundational understanding of how to move data to and from applications and outside of your Kubernetes cluster reliably. Pulsar is a great fit for cloud native application deployments because it can scale compute and storage independently. The declarative nature of deployments makes it easy for data engineers and SREs to deploy easily with consistency. Now that we have the means for data communication let's take it a step further with the right kind of network security.

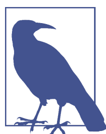
Securing Communications by Default with Cert-manager

An unfortunate reality we face at the end of product development is what gets left to complete: 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 on starting earlier without compromise!

As you can see, installing Pulsar has created a lot of infrastructure and communication between the elements. High traffic volume 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 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). 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 combination allows you to create 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

www.dbooks.org

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:
  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. The examples provided by the project are quite large so for brevity, we'll look at some key highlighted areas.

```
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.

Sidebar: 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

www.dbooks.org

don't like GNU Privacy Guard(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 search 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 have 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 as 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 initalize=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 an incredible scale. As we discussed at the beginning of the chapter, streaming systems come in many different flavors, and this is a perfect example. Flink has its competencies that overlap very little with other systems, in fact, it's widespread 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

www.dbooks.org

- Delivery guarantee: exactly-once
- Feature scope for streaming: analytics

The two main components of the Flink architecture are shown in Figure 8-6: the Job Manager and Task Manager.

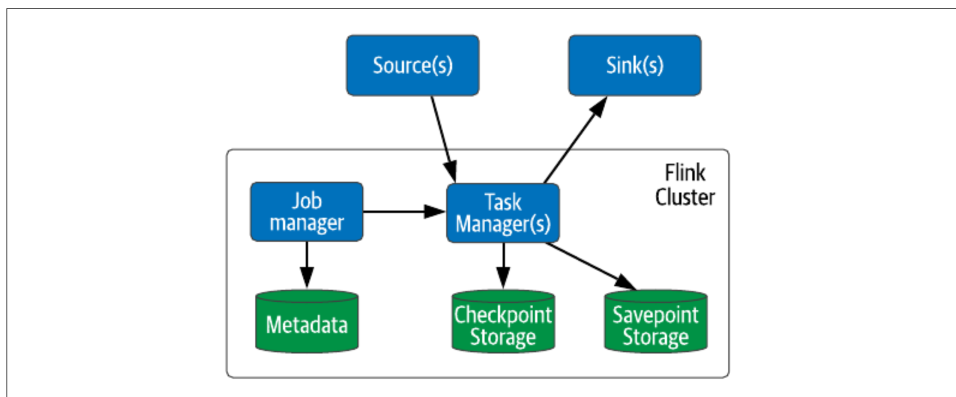


Figure 8-6. Apache Flink Architecture

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, 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. The JobManger uses TaskManagers 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 TaskManger 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.

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 essential. 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 specific 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 data window size to be processed. The limit of the boundary size is constrained mainly 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 continuous deployment workloads in Kubernetes and feature parity with cloud native applications in the areas of reliability and observability:

Continuous Deployment
: The core 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 and compiled into a jar file to create a Flink application. Flink provides a Docker image that encapsulates the application in a form that makes deployment on Kubernetes an easy task and facilitates continuous deployment.

Reliability
: Flink also has built-in support for savepoints, which makes updates easier by pausing and resuming jobs before and after system updates. Savepoints can also be used for fast recovery if a processing pod fails mid-job. Tighter integration with Kubernetes allows Flink to self-heal on failure by restoring pods and restarting jobs with savepoints.

Observability
: Cluster metrics are instrumented to output in Prometheus format. Operations teams can keep track of lifecycle events inside the Flink cluster with time-based details. Application developers can expose custom metrics using the Flink Metric System for further integrated observability.

Flink provides 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

www.dbooks.org

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 solid 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 kubectl or Helm. These choices may change in the future. 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.

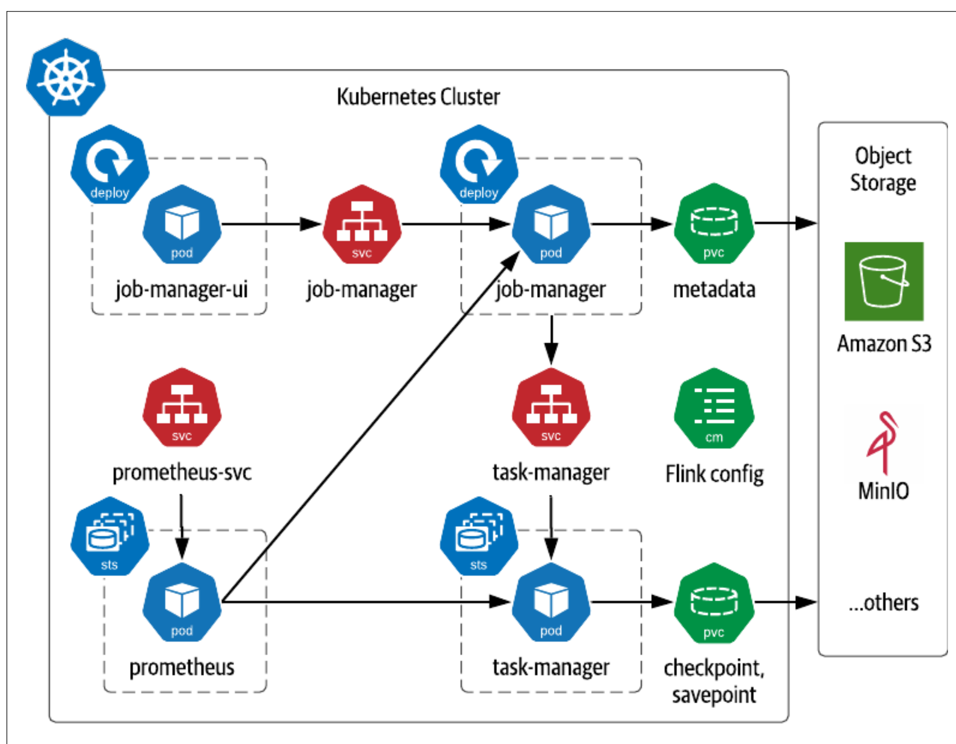


Figure 8-7. Deploying Flink on Kubernetes

As shown in Figure 8-7, a running Flink cluster has two main components we'll deploy in pods: the *Job Manager* and *Task Manager*. These are the basic units, but choosing which deployment mode is the critical 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 8-8 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.

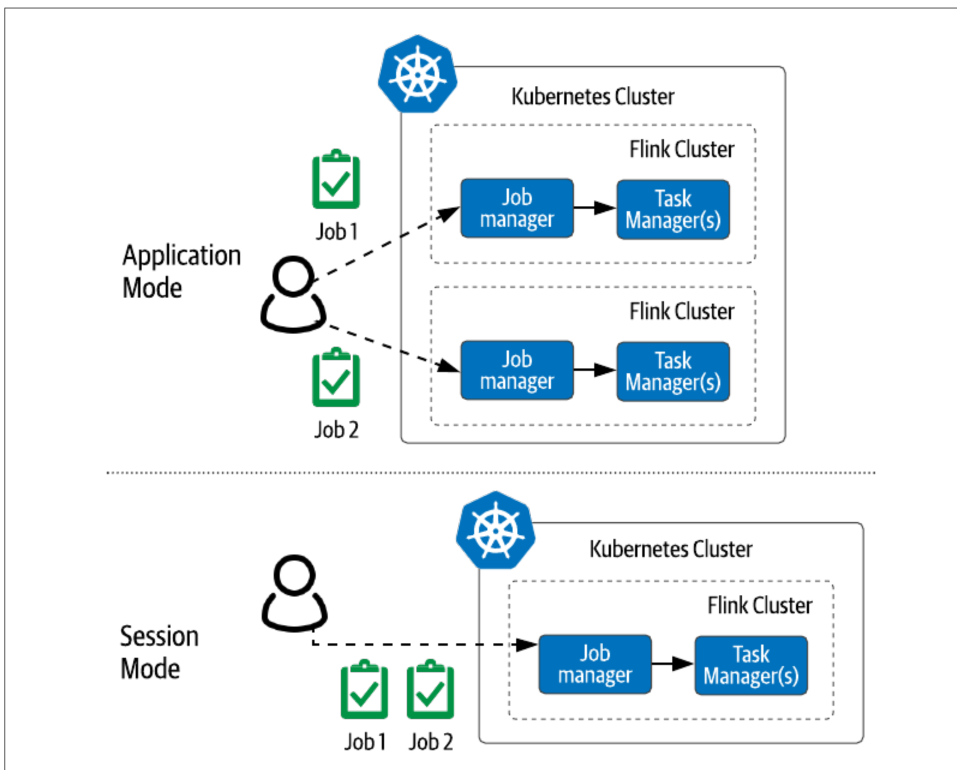


Figure 8-8. Apache Flink™ Modes

The selection of either Application Mode or Session Mode comes down to resource management inside your Kubernetes cluster, so let's look at both to 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.

www.dbooks.org

To start in Application Mode, you invoke the flink command line 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 \
    -Dkubernetes.cluster-id=<kubernetes cluster name> \
    -Dkubernetes.container.image=<custom docker image name> \
    local:///opt/flink/usrlib/my-flink-job.jar
```

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 will take several applications with it if it fails. 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.

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

This was a quick fly-by of a massive topic, but hopefully, it inspires you to look further. One resource we recommend is Stream Processing with Apache Flink (O'Reilly). 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. Flink uses containers as encapsulation lends itself to working with other development workflows.

Summary

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.

www.dbooks.org

CHAPTER 9

Data Analytics on Kubernetes

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 9th chapter of the final book. The GitHub repo is 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.

> *"Progress in technology is when we have the ability to be more lazy."*
> —Dr. Lurian Chirica

In the early 2000s, Google captivated the internet with a declared public goal: "to organize the world's information and make it universally accessible and useful." This was an ambitious goal that and accomplishing it would, to paraphrase, take "computer sciencing" the bits out of it. Given the increasing rate of data creation, Google needed to invent (and re-invent) ways of managing data volumes no one had ever considered. An entirely new community, culture, and industry were born around analyzing data called *analytics*, or what was eventually labeled "Big Data." Today analytics is a full-fledged member of almost every application stack and not just relegated to a "Google" problem. Now it's an everyone problem that can no longer be an art form restricted to a small club of people that know how to make it work. Organizations need reliable and fast ways to deploy applications, including analytics, and

because you are reading this book, you already know why: we need to do more with less.

The laziness Dr. Chirica was talking about in a tongue-in-cheek way in the quote that opened this chapter describes an ideal future. Instead of having a hundred-person team working night and day to analyze a petabyte of data, what if you could reduce that to one person and a few minutes? The cloud native way of running data infrastructure is a path we should all work towards to achieve that kind of glorious laziness.

We've already looked at several aspects of moving stateful workloads onto Kubernetes, including storage, databases, and streaming. In this chapter it's time to look at analytics to complete the picture. As a bit of a preview, Figure 9-1 shows how Data Analytics fits as the final part of our roadmap of managing the complete data stack using Kubernetes.

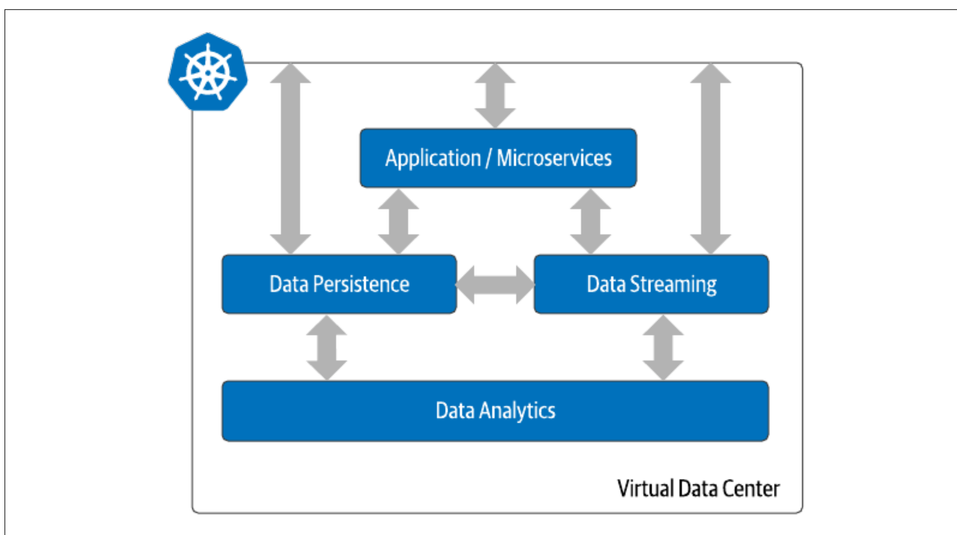


Figure 9-1. The Cloud Native Virtual Datacenter

In this architecture, there are no more external network requirements bridging to resources in or out of the Kubernetes cluster, just a single, virtual data center that serves our bespoke needs for cloud native applications. The large blocks represent the macro components of data infrastructure we discussed in Chapter 1, with the addition of user application code, deployed in microservices.

Introduction to Analytics

Analytic workloads and the accompanying infrastructure operations are much different from other workloads. Analytics isn't just another containerized system to orches-

www.dbooks.org

trate. The typical stateful applications like databases we have examined in previous chapters have many similar characteristics but tend to stay static or predictably slow-growing once deployed. However, there is one aspect of analytic workloads that strikes fear in many administrators: volume. While persistent data stores like databases can consume gigabytes to terabytes of storage, analytic volumes can easily soar into petabytes, creating an entirely new class of problems to solve. They don't call it "Big Data" for nothing.

The Oxford Dictionary defines analytics as "the systematic computational analysis of data or statistics." Wikipedia adds "It is used for the discovery, interpretation, and communication of meaningful patterns in data." Combine those definitions with large volumes of data and what sort of outcome should we expect for cloud native applications? Let's break down the different types of analytics workflows and methodologies:

Batch Analytics
: In computer science, a batch is a series of instructions applied to data with little or no user interaction. The idea of running batch jobs is as old as general-purpose computing. In distributed systems such as Apache Hadoop or Apache Spark, each individual job consists of a program that can operate on smaller bits of data in parallel and in stages or pipelines. The smaller results are combined into a single, final result at the end of a job. An example of this is MapReduce discussed later in the book. In most cases, statistical analysis is done, such as count, average, and percentile measurement. Batch analytics is the focus of this chapter.

Stream Analytics
: As discussed in Chapter 8, stream analytics is about what is *happening,* whereas batch analytics is about what *happened*. Many of the same APIs and developer methodologies are used in both stream analytics and batch analytics. This can be confusing and lead people to believe that they are the same thing when in fact they have very different use cases and implementations. A good example is fraud detection. The time frames for detecting and stopping fraud can be measured in milliseconds to seconds which fits the stream analytics use case. Batch analytics would be used to find fraud patterns over larger time periods.

Artificial Intelligence / Machine Learning (AI/ML)
: While artificial intelligence and machine learning can be considered a subset of batch analytics, they are such specialized fields that they deserve a special call-out. AI and ML are often mentioned together; however, they have two different output goals. Artificial Intelligence attempts to emulate human cognition in decision making. Machine Learning uses algorithms to derive meaning from pools of data, sometimes in ways that aren't readily obvious. Both approaches require the application of computing resources across volumes of data. This topic will be discussed in greater detail in Chapter 10.

Deploying Analytic Workloads in Kubernetes

The original focus of Kubernetes was on scaling and orchestrating stateless applications. As you're learning in this book, Kubernetes is evolving to support stateful applications. The promise of moving more and more workloads into virtual datacenters has been highly motivating, and the world of analytics can take advantage of the progress made in stateless and stateful workloads. However, Kubernetes has some unique challenges in managing analytic workloads and many are still a work in progress. What more is required to complete the data picture and make analytic workloads on par with other parts of the stack like microservices and databases? Here are a few of the key considerations we'll examine in this chapter:

Orderly Execution
: An essential aspect of analytic workloads is the order of operations required to analyze large volumes of data. This involves far more than just making sure Pods are started with the proper storage and networking resources. It also includes a mapping of the application with the orderly execution run in each Pod. The Kubernetes component primarily responsible for this task is **kube-scheduler** (See Chapter 5), but the controllers for Jobs and CronJobs are involved as well. This is a particular area of attention for the Kubernetes communities focusing on analytics, which we will further cover in the chapter.

Storage Management
: Analytic workloads use ephemeral and persistent storage in different jobs that process data. The real trouble occurs when it comes to identifying and selecting the right storage per job. Many analytic workloads require ephemeral storage for short periods and more efficient (cheaper) persistent storage for long terms. As you learned in Chapter 2, Kubernetes storage has greatly increased maturity. Analytics projects that run on Kubernetes need to take advantage of the work already done with stateful workloads and continue to partner with the Kubernetes community for future enhancements in areas like storage classes and different access patterns.

Efficient use of resources
: There is an old saying that "everything counts in large amounts," and nothing makes that more evident than analytics. A job may require 1000 pods for 10 minutes, but what if it needs 10000? A challenging problem for the Kubernetes control plane. Another job might require terabytes of swap disk space that is only needed for the duration of a job. In a cloud native world, jobs should be able to quickly allocate the resources they need and release the resources when finished. Making these operations as efficient as possible saves time and more importantly, money. The fast and bursty nature of analytics has created some challenges for the Kubernetes API server and scheduler to keep up with all the Jobs that need to

www.dbooks.org

be run. Several of those challenges are already being addressed, as discussed later in the chapter, and some are still a work in progress.

Those are the challenges, but none of them are show-stoppers that will get in the way of our dream of a complete cloud native stack deployed as a single virtual data center in Kubernetes.

Sidebar: Analytics on Kubernetes is the next frontier

With Holden Karau, Open Source Engineer, Apache Spark PMC

Running analytic workloads has been the boss-level challenge for infrastructure engineers from day one. There is the challenge of massive volumes of needed resources, which in many cases are the most significant part of your infrastructure. Then, coordination is required to use all those resources efficiently, and this is where frameworks like Spark come into play. Projects like Yarn from Hadoop and then Mesos were developed to help with the container management game. Today infrastructure engineers everywhere are very happy with migrating the best aspects of those systems to Kubernetes.

Let's consider a few examples. Running multiple Spark tenants in Kubernetes provides better isolation between your workloads which is really important if you don't trust each workload to the same degree. Historically, the support for Python dependencies inside Spark jobs has been poor, but deploying on Kubernetes makes it possible to use leading edge Python libraries for machine learning and GPU usage. SREs can spend more time optimizing resources rather than chasing down obscure errors typical with large distributed systems.

Ultimately, we are still in the early days of learning how to run our analytic workloads in Kubernetes most effectively. The difficulty curve starts getting much steeper as data volumes increase. Once you start going over the tens or hundreds of terabytes, you will find yourself on the leading edge of Spark operations in Kubernetes. This is probably not a big surprise because that tends to be how infrastructure engineering works. The upper limit scale problems with Kubernetes are rooted in the early use cases it was designed for. The dynamic and elastic nature of Kubernetes works well for the use cases for which it was first designed, but it gets overwhelmed at the levels required to run Spark applications. This is not impossible to solve, and there are a few critical areas the Spark and Kubernetes communities are working to improve.

Kubernetes SREs love the idea of elastic workloads, but in Spark, that's been much more painful than it needs to be. Regular execution of a Spark job can cause a rapid increase in Pods while the job is processing, and then those resources are released when the job is complete. Scaling up is, unfortunately, much easier than scaling down. Spark can now make use of externalized resource allocation that opens the possibility for better solutions. Open source projects such as Apache YuniKorn and Volcano are already working on resource allocation solutions inside of Kubernetes, which the exe-

cution allocator and shuffle service can leverage to align more closely to the way Kubernetes works. Taken together, these efforts provide a significant path forward for more efficient resource usage in a dynamic Spark environment.

Resource-intensive applications like Spark can benefit from having more insight from the underlying systems to balance jobs as they are run. Today, Kubernetes and Spark do not share as much information as they need for the best outcomes, and you should adjust your expectations accordingly. A specific example is the way Kubernetes approaches storage quota enforcement, which can work against Spark in some cases. Ephemeral storage is a key part of Spark execution, but Kubernetes provides no system-level APIs that Spark could use to check the utilization against the storage quota, which can cause job failures. Previous applications that have been deployed on Kubernetes haven't needed this level of insight. Spark creates a compelling case for significant changes to expose additional system level APIs in Kubernetes, which will make it possible to build much more reliable analytic workloads.

If you are interested in solving these problems, these solutions will happen in open source projects. You can get involved today and help us move forward by participating in the larger community. Apache Spark has room for improvement and so does Kubernetes, but the future direction is clear. Kubernetes is where cloud native analytics will happen, and Spark will continue to evolve. There will be a lot of interesting work in this area for the next several years at least, and it's a pretty exciting community to be a part of.

Engineers can be their own worst enemies. Often when we go to solve one problem, it creates a few more that need to be solved. We can count this as progress, however, when it comes to managing data. Every step up we take, despite the challenges, allows for new solutions that were never available before. It's staggering to think of how much is possible with just a small number of people where not too long ago, it took massive teams to accomplish anything close to the analytics required today. See the quote about laziness at the beginning of the chapter. There is still work to be done and next, we will look at what tools are available for analyzing data in Kubernetes.

Introduction to Apache Spark™

Google changed the world of data analytics with the MapReduce algorithm simply by describing it to the world in an academic paper. Not long after the MapReduce paper got engineers talking, an open source implementation was created, the now-famous Apache Hadoop™. A massive ecosystem was built up around Hadoop with tooling and complementary projects such as Hadoop Distributed File System (HDFS). Growing pains from this fast-moving project opened the door for the next generation of tools that built on the lessons learned with Hadoop. One project that grew in popularity as an alternative to Hadoop was Apache Spark™. Spark addressed reliability and

www.dbooks.org

processing efficiency problems by introducing the Resilient Distributed Dataset (RDD) API and Directed Acyclic Graphs (DAG).

The RDD was a significant improvement over the forced linear processing patterns of MapReduce. This involved a lot of reading from disk, processing, and then writing back to disk only to be redone over and over. This put a burden on developers to reason through how data was processed. RDDs shifted the responsibility away from developers as an API that created a unified view of all data while abstracting the actual processing details. Those processing details were created in a workflow to perform each task expressed in a DAG. The DAG is nothing more than an optimized path that describes data and operations to be completed in an orderly fashion until the final result is produced. RDDs were eventually replaced with the DataSet and DataFrame APIs, which further enhanced developer productivity over large volumes of data.

Spark's operational complexity is greatly reduced compared to Hadoop, which notoriously tipped the scale with the infrastructure required even for basic jobs. Spark is an excellent example of one of the benefits of being a next-generation implementation with great hindsight. Much effort was put into simplifying Spark's architecture, leveraging distributed systems concepts. The result is the three familiar components you should be familiar with in a Spark Cluster shown in Figure 9-2.

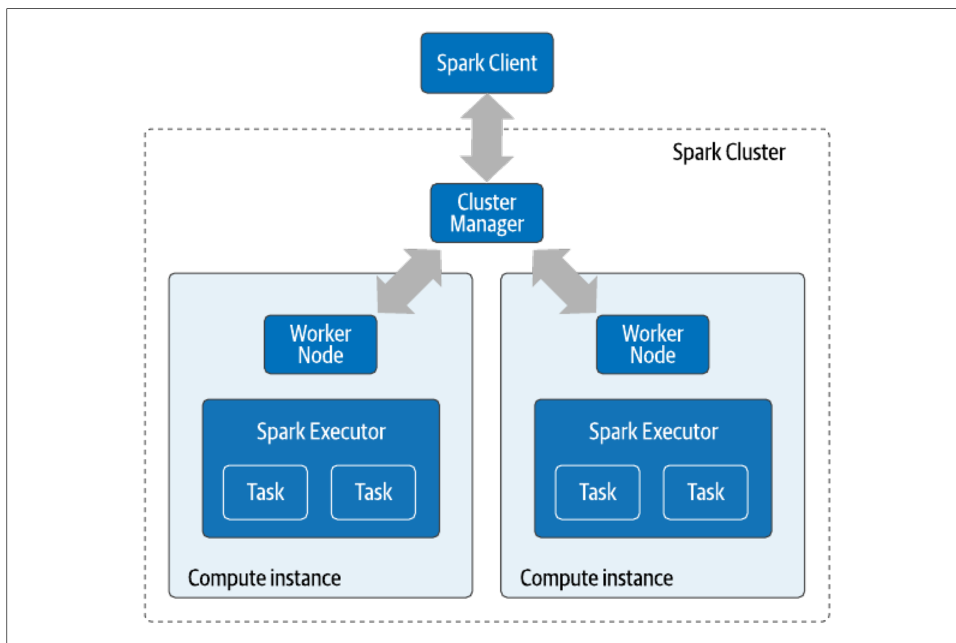


Figure 9-2. Components of a Spark Cluster

Let's review the responsibilities of each of these components:

Cluster Manager
: The Cluster Manager is the central hub for activity in the Spark cluster where new jobs are submitted for processing. The Cluster Manager also acquires the resources needed to complete the task submitted. Different versions of the Cluster Manager are primarily based on how resources are managed (Standalone, YARN, Mesos, and Kubernetes). The Cluster Manager is critical for deploying your Spark application using Kubernetes.

Worker Node
: When Spark jobs run, they are broken into manageable pieces by the Custer Manager and handed to the Worker Nodes to perform the processing. They serve as the local manager for hardware resources as a single point of contact. Worker Nodes invoke and manage Spark Executors.

Spark Executor
: Each application sent to a Worker Node will get its own Spark Executor. Each Executor is a stand-alone JVM process that operates independently and communicates back with the Worker Node. The tasks for the application are broken into threads that consume the compute resources allocated.

These are the traditional components of Spark as designed early in the project. What we'll see is that the need of deploying a cloud native version of Spark forced some architectural evolution. The fundamentals are the same, but the execution framework has adapted to take advantage of what Kubernetes provides and eliminate duplication in orchestration overhead. In the next section, we'll take a look at what those changes are and how to work with Spark in Kubernetes.

Deploying Apache Spark™ in Kubernetes

As of Apache Spark version 2.3, Kubernetes is one of the supported modes in the Cluster Manager. It would be easy to understate what that has meant for Spark as a cloud native analytics tool. Starting with Spark 3.1, Kubernetes mode is considered production-ready, continually adding steady improvements. When the Spark project looked at what it takes to run a clustered analytics system inside a cluster orchestration platform, there were a lot of overlaps that became obvious. Kubernetes already had the mechanisms in place for the lifecycle management of containers and the dynamic provisioning and de-provisioning of compute elements, so Spark lets Kubernetes take care of this work. The redundant parts were removed, and Spark is closer to how Kubernetes works as a result. The **spark-submit** command-line tool was extended to interface with Kubernetes clusters using the Kubernetes API, maintaining a familiar toolchain for developers and data engineers. These unique aspects of a Spark deployment in Kubernetes are shown in Figure 9-3.

www.dbooks.org

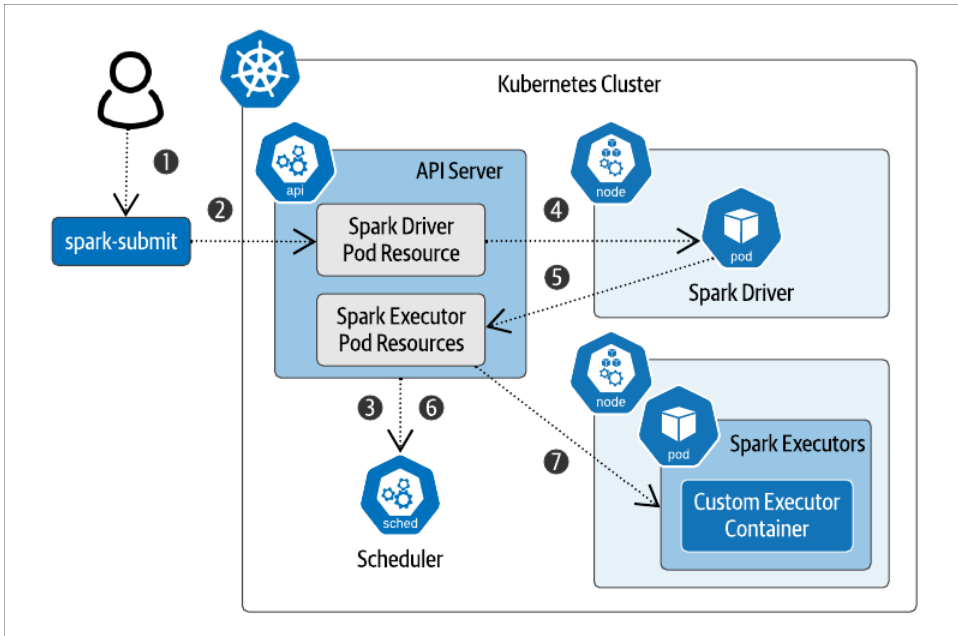


Figure 9-3. Spark on Kubernetes

Let's highlight a few of the differences:

Spark Driver
: The dedicated Cluster Manager of a standalone Spark cluster is replaced with native Kubernetes cluster management and the Spark Driver for Spark-specific management. The Spark Driver Pod is created when the Kubernetes API server receives a job from the **spark-submit** tool. It invokes the Spark Executor Pods to satisfy the job requirements. It is also responsible for cleaning up Executor Pods after the job, making it a crucial part of elastic workloads.

Spark Executor
: Like a stand-alone Spark cluster, Executors are where the work gets done and where the most compute resources are consumed. Invoked from the Spark Driver, they take job instructions passed by spark-submit with details such as CPU and memory limits, storage information, and security credentials. The containers used in Executor Pods are pre-created by the user.

Custom Executor Container
: Before a job is sent for processing using **spark-submit**, users must build a custom container image tailored to meet the application requirements. The Spark distribution download contains a Dockerfile that can be customized and used in conjunction with the **docker-image-tool.sh** script to build and upload the container

required when submitting a Spark job in Kubernetes. The custom container has everything it needs to work within a Kubernetes environment like a Spark Executor based on the Spark distribution version required.

The workflow for preparing and running Spark jobs when using Kubernetes and defaults can be relatively simple, only requiring a couple of steps. This is especially true if you are already familiar with and running Spark in production. You will need a running Kubernetes cluster and a download of Apache Spark in a local file path along with your Spark application source code.

Build your custom container

An executor container encapsulates your application and the runtime needed to act as an Executor Pod. The build script takes an argument for the source code repository and a tag assignment for the output image when pushed to your Docker registry.

```
$ ./bin/docker-image-tool.sh -r <repo> -t <tag> build
```

The output will be a Docker image with a jar file containing your application code. You will then need to push this image to your Docker registry.

```
$ ./bin/docker-image-tool.sh -r <repo> -t <tag> push
```

Docker image tags

Be mindful that your tag name is labeled and versioned correctly. Re-using the same tag name in production could have some unintended consequences, as some of us have learned from experience.

Submit and run your application

Once the Docker image is pushed to the repo, you use **spark-submit** to start the process of running the Spark application inside Kubernetes. This is the same **spark-submit** used for other modes, so many of the same arguments are used. This corresponds to Step 1 in Figure 9-3.

```
$ ./bin/spark-submit \
    --master k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port> \
    --deploy-mode cluster \
    --name <application-name> \
    --class <fully-qualified-class-name> \
    --conf spark.executor.instances=<instance-number> \
    --conf spark.kubernetes.container.image=<spark-image> \
    local:///path/to/application.jar
```

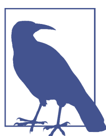
There are quite a few things happening here but the *most important* is in the --master parameter. To indicate this is for Kubernetes, the URL in the argument must start with a **k8s://** and point to the API server in the default Kubernetes cluster specified in

www.dbooks.org

your local .kubeconfig file. The **<spark-image>** is the Docker image you created in Step 1 and the application path refers to your application stored inside the image.

Next is Step 2, where **spark-submit** interacts with the Kubernetes cluster to schedule the Spark Driver Pod (Steps 3 and 4). The Spark Driver parses the job parameters and works with the Kubernetes Scheduler to set up Spark Executor pods (Steps 5, 6, and 7) to run the application code contained in the customer container image. The application will run to completion, and eventually, the Pod used will be terminated, and resources returned to the Kubernetes cluster in a process called garbage collection.

This is just an overview of how Spark natively works with Kubernetes. Please refer to the official documentation to go much further in-depth. There are many ways to customize the arguments and parameters to best fit your specific needs.



Security considerations when running Spark in Kubernetes

It should be clearly stated that security is **not** enabled by default when using Spark in Kubernetes. The first line of defense is authentication. Production Spark applications should use the built-in authentication in Spark to ensure the users and processes accessing your application are the ones you intended.

When creating a container for your application, the Spark documentation highly recommends changing the USER directive to an unprivileged UID and GID to mitigate against privilege escalation attacks. This can also be accomplished with a SecurityContext inside of the Pod Template file provided as a parameter to **spark-submit**.

Storage access should also be restricted with the Spark Driver and Spark Executor. Specifically, you should limit the paths that can be accessed by the running application to eliminate any accidental access in the event of a vulnerability. These can be set inside a Pod Security Policy, which the Spark Documentation recommends.

For optimal security of your Spark applications, use the security primitives Kubernetes provides and customize the defaults for your environment. The best security is the one you don't have to think about. If you are an SRE, this is one of the best things you can do for your developers and data engineers. Default secure!

Kubernetes Operator for Apache Spark

If Spark can run in Kubernetes via **spark-submit**, why do we need an operator? As you learned in previous chapters, Kubernetes operators give you more flexibility in managing applications and a more cloud native experience overall. Using **spark-submit** to run your Spark applications requires your production systems to be set up

with a local installation of Spark, including all dependencies. The Spark on Kubernetes Operator allows SREs and developers to manage park applications declaratively using Kubernetes tools such as Helm and **kubectl**. It also allows better observability on running jobs and exporting metrics to external systems like Prometheus. Finally, using the operator provides an experience much closer to running other applications in Kubernetes.

The first step is to install the operator into your Kubernetes cluster using Helm.

```
$ helm repo add spark-operator \
  https://googlecloudplatform.github.io/spark-on-k8s-operator
$ helm install my-release spark-operator/spark-operator \
  --namespace spark-operator --create-namespace
```

Once completed, you will have a SparkApplication controller running and looking for SparkApplication objects. This is the first big departure from spark-submit. Instead of a long list of command line arguments, you use the SparkApplication CRD to define the Spark job in a YAML file. Let's look at a config file from the official documentation.

```
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: spark-pi
  namespace: default
spec:
  type: Scala
  mode: cluster
  image: "gcr.io/spark-operator/spark:v3.1.1"
  imagePullPolicy: Always
  mainClass: org.apache.spark.examples.SparkPi
  mainApplicationFile: "local:///opt/spark/examples/jars/spark-
examples_2.12-3.1.1.jar"
  sparkVersion: "3.1.1"
  restartPolicy:
    type: Never
  volumes:
    - name: "test-volume"
      hostPath:
        path: "/tmp"
        type: Directory
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.1.1
    serviceAccount: spark
    volumeMounts:
      - name: "test-volume"
        mountPath: "/tmp"
```


www.dbooks.org

```
executor:
  cores: 1
  instances: 1
  memory: "512m"
  labels:
    version: 3.1.1
  volumeMounts:
    - name: "test-volume"
      mountPath: "/tmp"
```

The **spec:** section is similar to the parameters you passed in **spark-submit** with details about your application. The most important is the location of the container image. This example uses a default Spark container with the **spark-examples** jar file pre-installed. You will need to use the **docker-image-tool.sh** to build the image for your application as described in an earlier section, and modify the mainClass and mainApplicationFile as appropriate for your application.

Two other notable fields under **spec:** are **driver:** and **executor:**. These provide the specifications for the Spark Driver Pods and Spark Executor Pods that the Spark Operator will deploy. For **driver:** only one core is required but cpu and memory allocations need to be enough to maintain the number of executors you require. The number is set in the **executor:** section under **instances:**.

Minding your resources

For resource management, the requests you make under **driver:** and **spec:** need to be carefully considered for resource management. The number of instances plus their allocated CPU and memory could use up resources quickly. Jobs can hang indefinitely while waiting for resources to free up, which may never happen.

Now that your configuration yaml is ready, it's time to put it into action. For a walkthrough, refer to Figure 9-4.

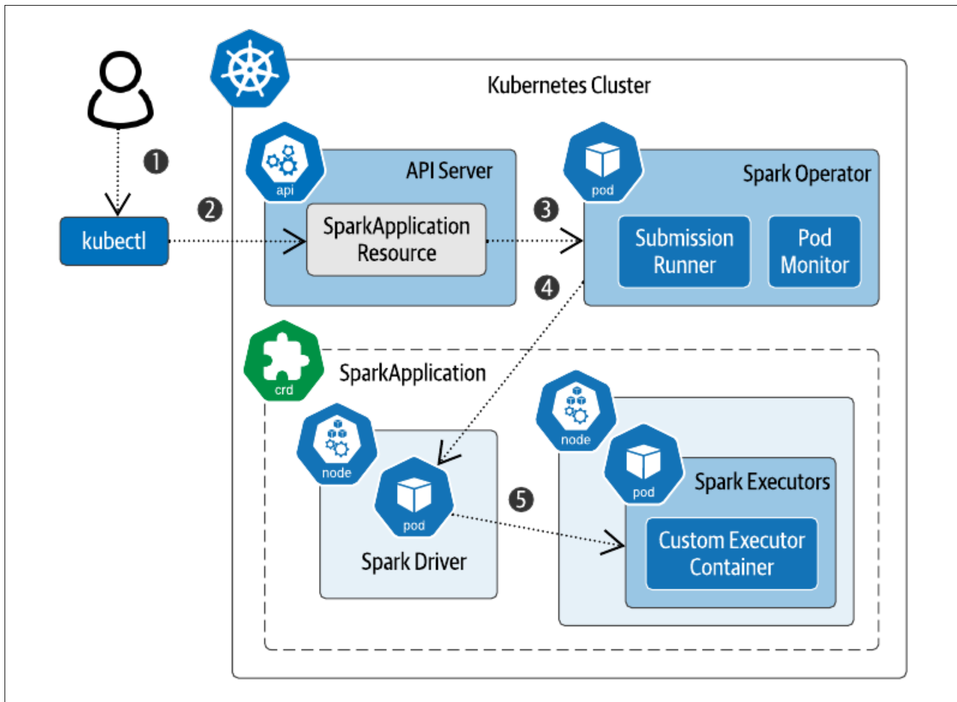


Figure 9-4. Spark on Kubernetes Operator

First, use '**kubectl apply -f <filename>**' (Step 1) to apply the SparkApplication into your running Kubernetes cluster (Step 2). The Spark Operator listens for new applications (Step 3) and when a new config object is applied, the Submission Runner controller begins the tasks of building out the required pods. From here the actions taken in the Kubernetes cluster are the same as if you used **spark-submit,** with all of the parameters being supplied in this case via the SparkApplication YAML. The Submission Runner starts the Spark Driver Pod (Step 4) which in turn directs the Spark Executor Pods (Step 5), which runs the application code to completion. The Pod Monitor included in the Spark Operator exports Spark metrics to observability tools such as Prometheus.

The Spark Operator fills in the gaps between the way **spark-submit** works versus how SREs and developers typically deploy applications into Kubernetes. This was a long answer to the question posed at the beginning of this section. We need an operator to make using Spark more cloud native, and, therefore, more manageable in the long run. The cloud native way of doing things includes taking a declarative approach to managing resources and making those resources observable.

www.dbooks.org

Alternative Schedulers for Kubernetes

As you learned in Chapter 5, the Kubernetes Scheduler has a basic but essential job: take requests for resources and assign the compute, network and storage to satisfy the requirements. Let's look at the default approach for this action as shown in Figure 9-5.

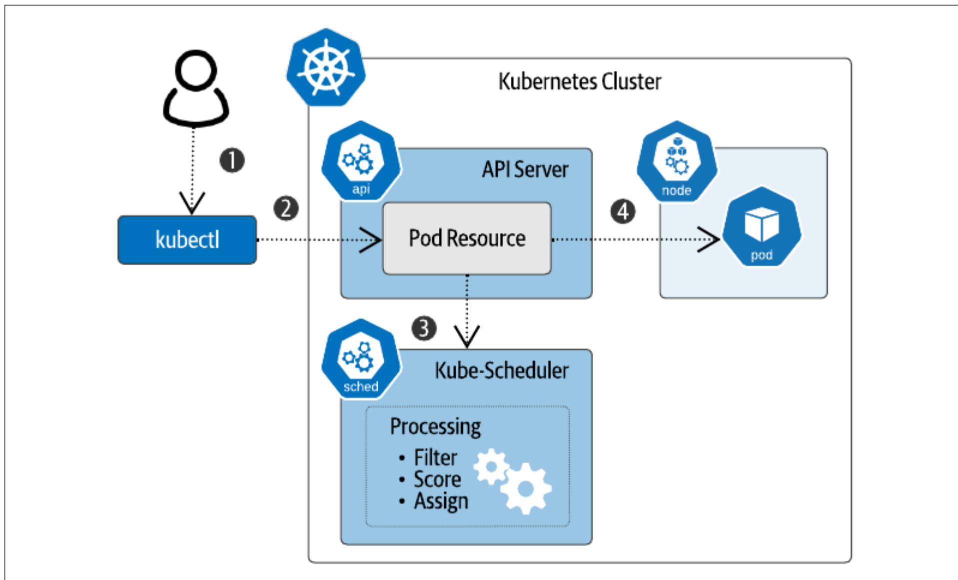


Figure 9-5. Typical Kubernetes Scheduling

A typical scheduling effort begins when you create a *deployment.yaml* file describing the resources required (Step 1), including which Pod resources are needed and how many. When the YAML file is submitted (Step 2) to the kubernetes cluster API server using **kubectl apply -f <deployment.yaml>**, the Pod resources are created with the supplied parameters and are ready for assignment to a Node. Nodes have the needed pool of resources, and it's the job of the kube-scheduler to be the matchmaker between Nodes and Pods. The Scheduler performs state matching whenever a new Pod resource is created (Step 3), and checks if the Pod has an assigned Node. If not, it makes the calculations needed to find an available Node. The Scheduler examines the requirements for the Pod, scores the available Nodes using an internal set of rules and selects a Node to run the Pod (Step 4). This is where the real work of container orchestration in Kubernetes gets done.

However, we have a problem with analytic workloads: the default Kubernetes scheduler was not designed for batch workloads. The design is just too basic to work the way that's needed for analytics. As mentioned by Holden Karau in the previous sidebar, Kubernetes was built for the needs of stateless workloads. These are long-running

processes that may expand or contract over time but tend to remain relatively static. Analytic applications such as Spark are different, requiring the scheduling of potentially thousands of short-lived jobs.

Thankfully, the developers of Kubernetes anticipated expanded requirements for future scheduling needs and made it possible for users to specify their scheduler in a configuration, bypassing the default scheduling approach.

The strong desire to manage the entire application stack with a common control plane has been an innovation driver. As we've shown above in Deploying Apache Spark™ in Kubernetes, Spark has been moving closer to Kubernetes. In this section, we'll look at how some teams have been bringing Kubernetes closer to Spark by building more appropriate schedulers. Two open source projects are leading the way in this effort: Volcano and Apache YuniKorn™. These schedulers share similar guiding principles that make them more appropriate for batch workloads by providing alternative features.

Multi-tenant Resource Management

The default Kubernetes Scheduler allocates Pods as requested until no more available resources match pod requirements. Both YuniKorn and Volcano.sh provide a wide variety of resourcing modes to match your application needs better, especially in multi-tenant environments. Fairness in resource management prevents one analytic job from starving out other jobs for required resources. As these jobs are scheduled, the entire resource pool is considered to balance utilization based on priority and throughput.

Gang scheduling adds another layer of intelligence. If a submitted job needs a certain amount of resources, it doesn't make sense to start the job if every Pod can't be started. The default scheduler will start Pods until the cluster runs out of resources, potentially stranding jobs as they wait for more Pods to come online. Gang scheduling implements an all or nothing approach, as jobs will only start when all resources needed are available for the complete job.

Job Queue Management

Smarter queue management can also lead to better resource management. If one job needs few resources and can be run while larger jobs are being run, the scheduler can fit the job in and, therefore, increase the Kubernetes cluster's overall throughput. In some cases, users need control over what jobs have priority and which can preempt or pause other running jobs as they are submitted. Queues can be reordered or reprioritized after jobs are submitted. Observability tooling provides queue insights that help determine total cluster health and resource usage.

If you are considering a production deployment of analytic workloads, you should avoid using the default scheduler, kube-scheduler. It wasn't designed for what you

www.dbooks.org

need in this case. Starting with a better scheduler lets you future-proof your Kubernetes experience. Let's examine some highlights of each scheduler.

Apache YuniKorn™

The YuniKorn project was built by engineers from Cloudera out of the operational frustration of working with analytic workloads in Spark. In the spirit of using Open Source to solve problems as a community, YuniKorn was donated to the Apache Software Foundation and accepted as an incubating project in 2020. The name comes directly from the two systems it supports, Yarn and Kubernetes. (Y uni-***fied*** K. YuniKorn. Get it?) It addresses the specific resource management and user control needs of analytic workloads from a Spark cluster administration point of view. YuniKorn also added support for TensorFlow and Flink jobs with the same level of resource control. No doubt, born of the same operation frustrations found in Spark.

YuniKorn is installed in Kubernetes using Helm. The goal of YuniKorn is to transform your Kubernetes Cluster into a place that is friendly to the resource requirements of batch jobs. A key part of that transformation is replacing the default kube-scheduler. To demonstrate how, let's use Figure 9-6 to walk through the components.

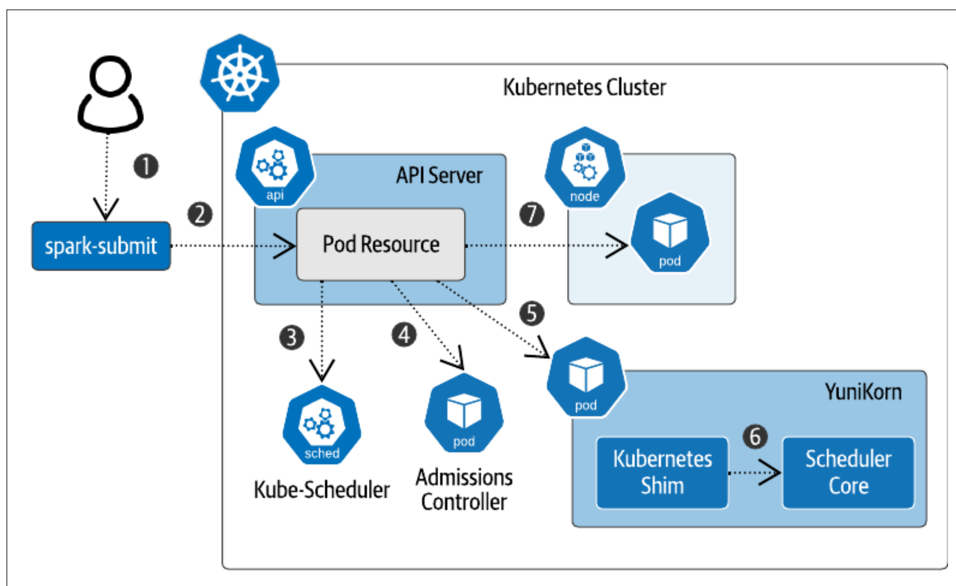


Figure 9-6. YuniKorn Architecture

YuniKorn is meant to be a drop-in scheduler replacement with minimal changes to your existing Spark workflow, so we will start there. When new resource requests (Step 1) are sent to the Kubernetes API server via spark-submit (Step 2), the default kube-scheduler (Step 3) is typically used to match Pods and Nodes. When YuniKorn

is deployed in your cluster, an admissions-controller Pod is created. The job of the admissions-controller is to listen for new resource requests (Step 4) and make a small change, adding schedulerName: yunikorn to the resource request. If you need more fine-grained control, you can disable the admissions-controller and enable YuniKorn on a per-job basis by manually adding the following line to the SparkApplication YAML.

```
spec:
schedulerName: yunikorn
```

All scheduling needs will now be handled by the YuniKorn Scheduler (Step 5). YuniKorn is built to run with multiple orchestration engines and provides an API translation layer called Kubernetes Shim to manage communication between Kubernetes and the YuniKorn Core (Step 6). YuniKorn Core extends the basic filter and score algorithm available in the default kube-scheduler by adding options appropriate for batch workloads such as Spark. These options range from simple resource-based queues to more advanced hierarchical queue management which allows for queues and resource pools to map to organizational structures. Hierarchical pooling can be helpful for those with a massive analytics footprint across many parts of a large enterprise and is critical for multi-tenant environments when running in a single Kubernetes cluster.

YuniKorn Core is configured using the queues.yaml file which contains all the details of how to YuniKorn will schedule Pods to Nodes which include:

Partitions
: One or more named configuration sections for different application requirements.

Queues
: Fine grained control over resources in a hierarchical arrangement to provide resource guarantees in a multi-tenant environment.

Node Sort Policy
: How Nodes are selected by available resources. Choices are FairnessPolicy and BinPackingPolicy.

Placement Rules
: Description and filters for Pod placement based on user or group membership.

Limits
: Definitions for fine grained resource limits on partitions or queues

New jobs are processed by YuniKorn Core by matching details and assigning the right queue. At this point the scheduler can make a decision to assign Pods to Nodes which are then brought online (Step 7).

www.dbooks.org

YuniKorn also ships with an observability web-based tool called Scheduler UI that provides insights into job and queue status. It can be used to monitor scheduler health and provide better insights to troubleshoot any Job issues.

Volcano

Volcano was developed as a general-purpose scheduler for running High Performance Computing (HPC) workloads in Kubernetes. Volcano supports a variety of workloads, including Spark, Flink, PyTorch TensorFlow, and specialized systems such as KubeGene for genome sequencing. Engineers built Volcano at Huawei, Tencent, and Baidu, to name a few of the long list of contributors. Donated to the Cloud Native Computing Foundation, it was accepted as a Sandbox project in 2020.

Volcano is installed using Helm and creates CRDs for Jobs and Queues, making the configuration a core part of your Kubernetes cluster as compared with YuniKorn, which is more of a bypass. This is a reflection of the general-purpose nature of Volcano. When installed, the Volcano Scheduler is available for any process needing advanced scheduling and queuing. Let's use Figure 9-7 to walk through how it works.

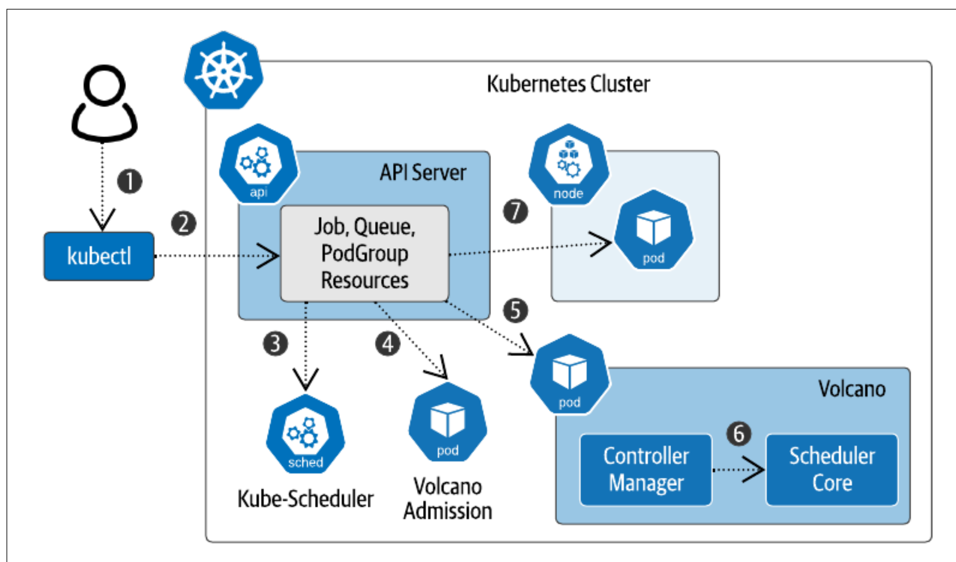


Figure 9-7. Volcano Architecture

To use Volcano with your batch jobs, you will need to explicitly add the scheduler configuration to your Job YAML file (1). If you are using Volcano for Spark, it is recommended by the Volcano project to use the Spark Operator for Kubernetes and add one field to your SparkApplication YAML:

```
spec:
batchScheduler: "volcano"
```

You can then use **kubectl apply -f** as normal to submit your Job (2). Without specifying the Volcano Scheduler, Kubernetes will match Pods and Nodes with the default kube-scheduler (3).

A Helm installation of Volcano will install the CRDs for Job, Queue, and PodGroup and create a new Pod called Volcano Admission. Volcano Admission (4) attaches to the API server and validates Volcano-specific CRD entries and Jobs asking for the Volcano Scheduler.

Job
: A Volcano-specific job with extended configuration for high performance computing.

Queue
: A collection of PodGroups to be managed as a First In, First Out (FIFO) resource group. Configuration dictates the behavior of the queue for different situations.

PodGroup
: A collection of Pods related to their purpose. Examples would be groups for Spark and TensorFlow with different properties for each.

When selected as the scheduler for a job (5), the Volcano Scheduler will take the CRDs and start to work (6). Incoming Jobs marked to use Volcano as the scheduler are matched with a PodGroup and Queue. Based on this assignment a final Node placement is made for each Pod (7).

The cluster specific configuration for the Volcano Scheduler core is stored in a ConfigMap named volcano-scheduler-configmap. This config file contains two main sections: actions and plugins. Actions are an ordered list of each step in the Node selection for each Job: enqueue, allocate, preempt, reclaim, and backfill. Each step is optional and can be re-ordered to match the type of work that needs to be performed.

Plugins are the algorithms used to match Pods with Nodes. Each has a different use case and purpose and can be combined as an ensemble:

Gang
: this plugin looks for higher priority tasks in the Queue and performs preemption and eviction if needed to free up resources for them.

Binpack
: a classic algorithm to find the best fit for using every resource available by mixing different size resource requests in the most efficient manner.

Conformance
: ignores any task in the namespace kube-system for eviction decisions.

www.dbooks.org

Dominant Resource Fairness (DRF)
: an algorithm to address issues of fairness across multiple resource types to ensure all Jobs have equal throughput.

Proportion
: a multi-tenant algorithm to allocate dedicated portions of cluster allocation for running Jobs.

Task-topology
: uses affinity to put network-intensive Jobs physically closer together for more efficient network use.

NodeOrder
: takes multiple user-defined dimensions to score every available Node before selection.

Predicates
: looks for certain predicates in Nodes for selection. Currently only supports GPUSharing predicate.

Priority
: chooses task priority based on user-supplied configuration in priorityClassName, createTime, and id.

Service Level Agreement (SLA)
: uses the parameter JobWaitingTime to allow individual Jobs the control over priority based on when they are needed.

Time Division Multiplexing (TDM)
: when Nodes are both used for Kubernetes and Yarn, TDM will Schedule Pods that share resources in this arrangement.

Numa-aware
: Provides CPU resource topology-aware scheduling for Pods.

Outside of the Kubernetes installation, Volcano also ships with a command-line tool called **vcctl**. Managing Volcano can be done solely through the use of kubectl. However, **vcctl** presents an interface for operators more familiar with job control systems.

As you can see from the list of features offered by YuniKorn and Volcano, having choices is a beautiful thing. Regardless of which project you choose, you'll have a better experience running analytic workloads in Kubernetes with one of these alternate schedulers.

Analytic Engines for Kubernetes

Spark is certainly a powerful tool that solves many use cases in analytics. Having just a single choice can be restrictive once that tool no longer works the way you do. When Google developed MapReduce, the real need was in data transformation, such as taking a pool of data and creating a count of the things in it. This is still a relevant problem given the volumes of data we create, which has been for some time. Even before MapReduce, massively parallel processing (MPP) was a popular approach for data analysis. These "supercomputers" consisted of rows and rows of individual computers presented as a single processing grid for researchers in fields such as physics and meteorology to run massive calculations that would take far too long on a single computer.

A similar computing need arises when tackling the machine learning and AI tasks in analytics: many processes need to analyze a large volume of data. Libraries such as TensorFlow require analytic tools beyond data transformation. With Kubernetes, data scientists and engineers can now create virtual datacenters quickly with commodity compute, network and storage to rival some of the supercomputers of the past. This combination of technologies brings a completely new and exciting future for developers building ML and AI based applications based on a self-service usage model without waiting for time on the very expensive supercomputer (yes, this was a thing).

Sidebar: The evolution of analytics for developers in a cloud native world

With Dean Wampler, Product Engineering Director for Accelerated Discovery, IBM Research

There has been speculation that streaming and batch analytics will somehow converge into a single, universal approach for every project or product. I believe with Kubernetes that vision becomes less ideal as developers determine what they need to be successful and it's all about choosing the best tool for the job. In fact, there are likely going to be more choices available for analyzing data that fit different use cases in and outside of Kubernetes. Developers and data engineers will need a variety of tools to overcome limitations and tradeoffs.

Let's suppose you have bought into the idea that you could do everything with streams. What does that actually mean in practice and where are the limitations? Suppose I wanted to know exactly how many items of all SKUs I sold in every store segmented into hour buckets. The challenge with that calculation is some uncertainty in knowing when all the data is delivered for each hour bucket. You can't start the job exactly on the hour because there might be data still in flight. Worst case, some data may be significantly delayed due to network partitions or other outages. Now it's up to the developers to build in the sophistication for provisional results, which might be

www.dbooks.org

calculated as quickly as possible, and then integrate corrections when late data arrives. With more diversity in data tooling, they can better solve the problem. For this example, maybe they would use Apache Flink for dashboards with some percentage of accuracy about what's happening immediately. Data that is captured later would be used in an overnight Apache Spark job to do the final accounting and produce the canonical results per-hourly bucket. You could argue this is a much more reasonable level of sophistication based on using the simplest, correct tool for each job, composable using Kubernetes.

At larger data scales, organizations still like using Spark for big analytics tasks. However, there is a growing trend where data scientists and data engineers are starting to recognize it's reasonable to have data in a database, rather than a data lake as a matter of choosing the right tool for the job. Some teams will use something like Cassandra because they don't want the complexity of keeping track of HDFS and Parquet files, and they want the benefits of indexing and queries. They accept a performance hit from scanning tables versus a file. These are teams experienced in making trade-off decisions for convenience or reliability, and sometimes even for less performance. Kubernetes can encourage a more extensive choice of alternatives by reducing the upfront costs of trying new things. It's still early days for cloud native analytics, but the leaders in this space make it work by being more agile with underlying data infrastructure.

A new generation of analytic tools is expanding the choices that can be made. With all its dominance, Apache Spark is still very dependent on the Java Virtual Machine (JVM) and that feature is becoming a trade-off consideration. Some data engineering shops don't want to deal with the JVM anymore, they just want to run Python, sometimes with C-based library kernels for high performance. This has created an opportunity for projects like Ray and Dask to cater directly to teams that want to use Python first. Developers gravitate to tools that help them go faster with fewer trade-offs. However, while the JVM might be a liability in some cases, Spark has enormous mindshare and years of continuous improvement. Keeping with the theme of choice, it's easy to see how Kubernetes can help create a place where Spark and Ray could be used in the same application. Cloud native analytics could eliminate the zero-sum game of all-in-one tools.

The analytics convergence that people have talked about will likely happen at the interface level, giving developers a single interface with access to the appropriate tooling underneath. The right mix of services for a cloud native world. Batch offline analytics with data warehouses augmented with online databases. Streaming analytics that provides real-time updates to the same data used for the batch jobs. All are provided as services deployed in Kubernetes. The most important factor is the easy access it provides. Citizen data scientists can use Excel to explore data. Visualization tools can connect to any underlying service with low or no code. Python is increasingly the language of choice for data engineers and scientists building pipelines. Support for SQL across streaming and batch analytics has remained universally popular, leveraging a data language that has been the standard for decades. Kubernetes will have to

support this by enabling the fine-grained concurrency within a single process that some data processing systems require while leveraging pod boundaries for big chunks of resources. It's a balance of trade-offs. The winner will be the developers and data scientists who no longer have to worry about making bad tool choices, allowing them to spend more time writing code that creates value.

Access via the right APIs and ability via the right infrastructure built on Kubernetes is a powerful combination that the data science and Python community has been working to make a reality. Two new projects are already making a mark: Dask and Ray. As pointed out by Dean, Python is the preferred language for data science. Both Ray and Dask provide a native Python interface for massively parallel processing both inside and outside of Kubernetes.

Dask

Dask is a Python-based clustering tool for large-scale processing that abstracts away the complicated setup steps. It can be used for anything that you can express in a Python program but has found a real home in data science with the countless libraries available. Scikit-Learn, NumPy, TensorFlow, Pandas are all mature data science libraries that can be used on a laptop then scaled to a massive cluster of computers thanks to Dask.

Dask integrates nicely with Kubernetes to provide the easy user experience that operators and developers have come to expect with Python. The Dask storage primitives Array, DataFrame, and Bag map to many cloud native storage choices. For example, you could map a DataFrame to a file stored in a PersistentVolume or an object bucket such as S3. Your storage scale is only limited to the underlying resources and your budget. As the Python code is working with your data, Dask manages the chunking across multiple workers seamlessly.

Deployment options include the manual Helm install we are now familiar with from Chapter 4, as you can see in this example:

```
$ helm repo add dask https://helm.dask.org/
$ helm repo update
$ helm install my-dask dask/dask
```

Or as an alternative, you can install a Dask cluster in Kubernetes with a Jupyter Notebook instance for working inside the cluster.

```
$ helm install my-dask dask/daskhub
```

Once your Dask cluster is running inside Kubernetes you can connect as a client and run your Python code across the compute nodes using the **HelmCluster** object. Connect using the name you gave your cluster given at the time of installation.

www.dbooks.org

```
from dask_kubernetes import HelmCluster
from dask.distributed import Client
# Connect to the name of the helm installation
cluster = HelmCluster(release_name="my-dask")
# specify the number of workers(pods) explicitly
cluster.scale(10)
# or dynamically scale based on current workload
cluster.adapt(minimum=1, maximum=100)
# Your Python code here
```

If that wasn't easy enough, you can completely skip the Helm installation and just let Dask do that part for you. The **KubeCluster** object takes an argument specifying the pod configuration either using a **make_pod_spec** method or specifying a YAML configuration file. It will connect to the default Kubernetes cluster accessible via **kubectl** and invoke the cluster creation inside your Kubernetes cluster as a part of the running Python program.

```
from dask.distributed import Client
from dask_kubernetes import KubeCluster, make_pod_spec
pod_spec = make_pod_spec(image='daskdev/dask:latest',
                         memory_limit='4G', memory_request='4G',
                         cpu_limit=1, cpu_request=1)
cluster = KubeCluster(pod_spec)
# specify the number of workers(pods) explicitly
cluster.scale(10)
# or dynamically scale based on current workload
cluster.adapt(minimum=1, maximum=100)
# Connect Dask to the cluster
client = Client(cluster)
# Your Python code here
```

Developer access to Kubernetes clusters for parallel computing couldn't get much easier and this is the appeal new tools like Dask can provide.

Ray

In a significant difference from the arbitrary Python code in Dask, Ray takes a different approach to Python clustering by operating as a parallel task manager that includes a distributed computing framework. For the end-user, Ray provides low-level C++ libraries to run distributed code purpose-built for compute-intensive workloads typical in data science. The base is Ray Core which does all the work of distributing workloads using the concept of a task. When developers write Python code using Ray each task is expressed as a remote function, as shown in this example from the Ray documentation:

```
# By adding the `@ray.remote` decorator, a regular Python function
# becomes a Ray remote function.
@ray.remote
```

```
def my_function():
    return 1
```

In this basic example you can see the difference in the approach Ray takes for distributing work. Developers have to be explicit in what work is distributed with Ray Core handling the compute management with the Cluster Manager.

A Ray deployment in Kubernetes is designed to leverage compute and network resource management within dynamic workloads. The Ray Operator includes a custom controller and CRD to deploy everything needed to attach code to a Ray cluster. A Helm chart is provided for easy installation. However, since the chart is unavailable in a public repository you must first download the entire Ray distribution to your local filesystem. An extensive configuration YAML file can be modified, but to get a simple Ray cluster working, the defaults are fine, as you can see in this example from the documentation:

```
$ cd ray/deploy/charts
$ helm -n ray install example-cluster --create-namespace ./ray
```

This results in the creation of two types of pods installed. The Head Node handles the communication and orchestration of running tasks in the cluster and the Worker Node where tasks execute their code. With a Ray cluster running inside a Kubernetes cluster, there are two ways to run a Ray job. Interactively with the Ray client or as a Job submitted via **kubectl**.

The Ray client is embedded into a python program file and initializes the connection to the Ray cluster. This requires the head service IP to be exposed either through Ingress or local port forwarding. Along with the remote function code, an initializer will establish the connection to the externalized Ray cluster host IP and port.

```
import ray
ray.init("ray://<host>:<port>")
@ray.remote
def my_function():
    return 1
```

Another option is to run your code inside the Kubernetes cluster and attach it to an internal service and port. You use **kubectl** to submit the job to run and pass a Job description YAML file that outlines the python program to use and pod resource information. Here is an example Job file from the Ray documentation:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: ray-test-job
spec:
  template:
    spec:
      restartPolicy: Never
```


www.dbooks.org

```
    containers:
      - name: ray
        image: rayproject/ray:latest
        imagePullPolicy: Always
        command: [ "/bin/bash", "-c", "--" ]
        args:
          - "wget <URL>/job_example.py &&
            python job_example.py"
        resources:
          requests:
            cpu: 100m
            memory: 512Mi
```

This file can then be submitted to the cluster using **kubectl**

```
$ kubectl -n ray create -f job-example.yaml
```

Inside the Python file submitted, we can use the DNS name of the Ray service head and let Kubernetes ensure the network path is routed.

```
ray.init("ray://example-cluster-ray-head:10001")
```

For both external and internal modes of running Ray programs, the Head Node utilizes the Kubernetes scheduler to manage the Worker Node Pod lifecycle to complete the submitted job. Ray provides a simple programming API for developers to utilize large-scale cluster computing without learning Kubernetes administration. SREs can create and manage Kubernetes clusters that can be easily used by data scientists using their preferred Python programming language.

Summary

This wraps up the tour of data components in your cloud native application stack. Adding analytics completes the total data picture by giving you the ability to find insights in larger volumes of data that can complement other parts of your application.

Analytics is at the frontier of cloud native data innovation and for this reason "Big Data" isn't something you should assume fits into Kubernetes in the same way as other data infrastructure. Two primary differences are the volumes of data involved and the bursty nature of the workloads. There are still improvements that are needed to make Apache Spark run more effectively on Kubernetes, especially in the areas of job management and storage APIs. aligned better with analytics. However, the knowledge is available to help you deploy with confidence today. Projects such as Apache YuniKorn and Volcano.sh are already leading the way in open source to give Kubernetes a better foundation for analytic workloads. Emerging analytic engines such as Dask and Ray may be a better choice for your use case, and they can be used in combination with other tools.

While analytic workloads may not have been in your original plans for deployment in Kubernetes, they can't be skipped if your goal is to build the complete picture of a virtual data center, purpose-designed to run your application.

www.dbooks.org

CHAPTER 10

Machine Learning and Other Emerging Use Cases for Data on Kubernetes

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 10th chapter of the final book. The GitHub repo is 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.

In previous chapters, we have covered traditional data infrastructure, including databases, streaming platforms, and analytic engines with a Kubernetes-centered focus. Now it's time to start looking beyond and exploring the projects and communities that are beginning to make cloud native their destination, especially concerning Artificial Intelligence and Machine Learning.

Any time multiple arrows start pointing in the same direction, it's worth paying attention. The directional arrows in data infrastructure all point to an overall macro trend of convergence on Kubernetes, supported by several interrelated trends:

- Common stacks are emerging for managing compute-intensive AI/ML workloads, including those that leverage specific hardware such as GPUs.

- Common data formats are helping to promote the efficient movement of data across compute, network, and storage resources.
- Object storage is becoming a common persistence layer for data infrastructure.

In this chapter, we will look at several emerging technologies that embody these trends, the use cases they enable, and how they contribute to helping you further manage the precious resources of compute, network, and storage.

The Cloud Native AI/ML Stack

As discussed in Chapter 9, Analytics, Artificial Intelligence, and Machine Learning (AI/ML) on Kubernetes is a topic worthy of more detailed examination. If you aren't familiar with this specialty in the world of data, it's an exciting domain that enhances our ability to produce real time, data-driven decisions at scale. While many of the core algorithms have existed for decades, the nature of this work has been changing rapidly over the past few years. Data science as a profession has traditionally been relegated to the back office, where volumes of historical data were gleaned for insight to find meaning and predict the future. Data scientists rarely had any direct involvement with end-user applications, and their work was disconnected from user-facing applications.

This began to change with the emergence of the data engineer role. Data engineers build the processing engines and pipelines to productionalize data science and break down silos between disciplines. As is typical for emerging fields in data infrastructure, the largest, most vocal organizations set the tempo for data engineering, and their tools and methods have become the mainstream.

The real-time nature of data in applications can't be left to just databases and streaming platforms. Products built by data scientists must be closer to the end-user to maximize their effectiveness in applications. Many organizations have recognized this as both a problem and an opportunity: how can we make data science another near real-time component of application deployments? True to form, when faced with a challenge, the community has risen to the occasion to build new projects and create new disciplines. As a result, a new category of data infrastructure on Kubernetes is emerging alongside the traditional categories of persistence, streaming, and analytics. This new stack consists of tools that support the real-time serving of data specific to AI and ML.

AI/ML Definitions

If you are new to the field of AI/ML, you can quickly become overwhelmed by the terminology. Before we look at a few cloud native technologies that solve different problems in the AI stack, let's spend some time understanding the new set of terms

www.dbooks.org

and concepts that are critical to understanding this specialty. If you are familiar with AI/ML, you can safely skip to the next section.

First, let's briefly overview some common terms used in AI/ML. These frequently appear in descriptions of projects and features, and you'll need to understand them to select the right tools and apply them effectively.

Algorithm
: The basic computational building block of Machine Learning is the algorithm. Algorithms are expressed in code as a set of instructions to analyze data. Common algorithms include linear regression, decision trees, K-means, and random forest. Data Scientists spend their time working with algorithms to gain insights from data. When the procedures and parameters are right, the final repeatable form is output into models.

Model
: Machine Learning aims to build systems that mimic how humans learn so that they can answer questions based on provided data without explicit programming. Example questions include identifying whether two objects are similar, the likelihood of occurrence of a particular event, or choosing the best option given multiple candidates. The answering system for these questions is described in a mathematical model, or model for short. A model acts as a function machine where data that describes a question goes in, and new data that represents an answer comes out.

Feature
: Features are the portions of a more extensive data set relevant to a specific use case. Features are used both to train models and to provide input to models in production. For example, if you wanted to predict the weather, you might select time, location, and temperature from a much larger data set, ignoring other data such as air quality. Feature selection is the process of determining what data you'll use, which can be an iterative process. When you hear the word feature, you can easily translate that to just data.

Training
: A model consists of an algorithm plus data (features) that apply that algorithm to a particular domain. To train a model, training data is passed through the algorithm to help refine the output to match the expected answer. This training data contains the same features that will be used in production use, with the key difference that the expected answer is known. Training is the most resource-intensive phase of machine learning.

Flow
: Flow is a shorthand term for workflow. An ML workflow describes the steps required to build a working model. The flow generally includes data collection,

pre-processing and cleaning, model training, validation, and performance testing. These are typically fully automated processes.

Vector
: The classic mathematical definition of a vector is a quantity that indicates direction and magnitude. Machine learning models are mathematical formulas that use numerical data. Since not all source data is represented as numbers, normalizing input data into vector representations is the key to using general-purpose algorithms in machine learning. Images and text are examples of data that can be vector encoded in the pre-processing step of the flow.

Prediction
: Prediction is the step of using the created model to produce a likely answer based on input data. For example, we might ask the expected temperature for a given location, date, and time using a weather model. The question being answered takes the form: "What will happen?"

Inference
: Inference models look for reasons by reversing the relationship of input data vs output data. Given an answer, what features contributed to arriving at this answer? Another weather example: based on rainfall, what are the most associated temperatures and barometric pressures? The question being answered is: "How did this happen?"

Drift
: Models are trained with snapshots of data from a point in time. Drift is a condition where the model loses accuracy due to conditions that have changed over time or are no longer relevant based on the original training data. When drift happens in a model, the solution is to refine the model with updated training data.

Bias
: Models are only as good as the algorithms used and how they are trained. Bias can be introduced at several points: in the algorithm itself, sample data that contains user prejudice or faulty measurement, or exclusion of data. In any case, the goal of machine learning is to be as accurate as possible, and bias is an accuracy measurement. Detecting bias in data is a complex problem and is easier to address early through good data governance and process rigor.

These are some of the key concepts that will help you understand the rest of this section. For a more complete introduction, consider Introduction to Machine Learning with Python (O'Reilly) or one of the many quality online courses available from your favorite learning platform.

www.dbooks.org

Defining an AI/ML Stack

Given these definitions, we can describe the elements of a cloud native AI stack and the purposes such a stack can serve. Emerging disciplines and communities have similar implementations with minor variations as various teams innovate to solve their own specific needs. We can identify some common patterns by looking at organizations that use AI/ML in production at scale and the trends around Kubernetes adoption. Figure 10-1 shows some of the typical elements found in architectures currently in production. Without being prescriptive, we'll use this as an example of the types of tools in the stack and how they might fit together to serve the real-time components of AI/ML.

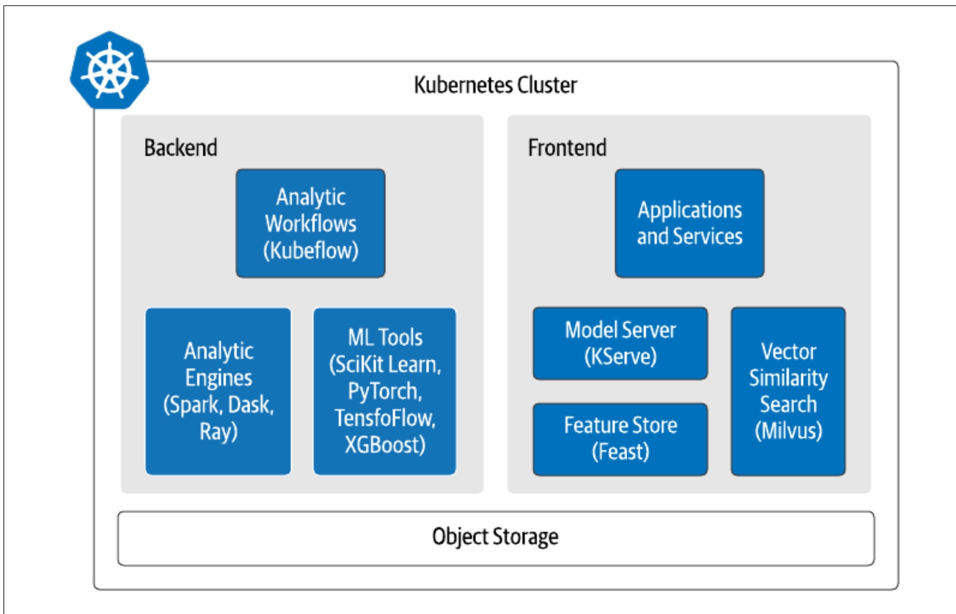


Figure 10-1. Common elements of a cloud native AI/ML stack

The goal of a cloud native AI/ML stack should be to get the insights produced by AI/ML as close to your users as possible, which means shortening the distance between backend analytic processes and making use of their output in frontend production systems. Data exploration happens using algorithms provided in libraries such as Scikit-Learn, PyTorch, Tensorflow, XGBoost. Python is the most commonly used language with Machine Learning libraries. The systems we discussed in Chapter 9, including Apache Spark, Dask, and Ray, are used to scale up the processing required to use Python libraries to build models. Kubeflow and similar tools allow data engineers to create ML workflows for model generation. The workflows output a model file to object storage, providing the bridge between the backend processes and front-end production use.

Models are meant to be used, and this is the role of real-time model serving tools such as KServe, Seldon, and BentoML. These tools perform predictions on behalf of applications using existing models from object storage and feature stores such as Feast. Feature stores perform full lifecycle management of feature data, storing new feature data in an online database such as Cassandra, training, and serving features to models.

Vector similarity search engines are a new but familiar addition to the real-time serving stack for applications. While traditional search engines such as Apache Solr provide convenient APIs for text searching, including fuzzy matching, vector similarity search is a more powerful algorithm, helping to answer the question "what is like the thing I currently have?". To do this, it uses relationships in the data instead of just the terms in your search query. Vector similarity supports many formats, including text, video, audio, and anything else that can be analyzed into a vector. Many open source tools implement vector similarity search, including Milvus, Weaviate, Qdrant, Vald, and Vearch.

Let's examine a few of the tools that support frontend ML usage by applications in more detail and learn how they are deployed in Kubernetes: KServe, Feast, and Milvus.

Real-time Model Serving with KServe

The "last mile" problem in AI/ML of real-time access to analytic products is one that Kubernetes is well poised to solve. Consider the architecture of modern web applications: HTTP servers that seem to simply serve a web page often have much more complexity behind them. The reality is that application logic and data infrastructure are combined to hide the complexity. Much like the HTTP server that listens for requests and serves a web page, a model server hides the complexity of loading and executing models. It focuses on the developer experience after the data science is done.

KServe is a Kubernetes native model server that makes it easy to provide prediction capabilities to applications in production environments. Let's learn more about the origins and functionality of KServe from one of the project founders.

Sidebar: Operationalizing Machine Learning Models with KServe

with Theofilos Papapanagiotou, Data Science Architect at Prosus

Google is well known for its contributions to the machine learning community with projects such as TensorFlow. Based on the framework they used to run TensorFlow internally, Google also created KubeFlow, an open source project to help data scientists and engineers use TensorFlow in production. KubeFlow contains multiple subprojects for different aspects of deploying machine learning workflows. One sub-

www.dbooks.org

project which addressed the externalization of models was called KubeFlow Serving or KFServing. Initially, it was only built for TensorFlow, but new contributors joined in and added support for other models such as PyTorch, SciKit Learn, and XGBoost. In 2021 KFServing became an independent project from KubeFlow and was renamed KServe.

The core function of KServe is to provide an API endpoint for deploying previously built machine learning models in Kubernetes. Deploying each model involves multiple steps. KServe handles the fetching of the model from an object store, loading it into memory, and determining if the model needs to use CPU or GPU. When GPU is required, KServe manages the copying of the model from CPU memory to GPU memory. This behavior can be specified with just a few lines of YAML, which eliminates a lot of the toil when working with Machine Learning in production environments. For SREs, there is additional integration with KNative eventing to manage the scale-out, and observability features like metrics and logging. These are expected behaviors of an HTTP API and important aspects of putting machine learning models in production.

There are many contributors to KServe, and they are all driven by a similar mission: operationalizing machine learning to be used by as many people as possible. Data is significant intellectual property to your organization, and data scientists are tasked to build models that make efficient use of that data. The real treasure for an organization is the ability to take those models and apply them to data to make predictions that can be used in your products, which in turn creates added value for your customers. KServe emphasizes using data in real-time over pushing it to a data lake where it might be forgotten. For this reason, KServe does not provide a general purpose data store, it's simply a hosting system for models. It functions as a microservice in your cloud native application, accepting inference requests containing a list of features. The data returned is a prediction based on the input and it has to happen quickly, efficiently, and securely.

Bloomberg is one of the top contributors to KServe, and their use case is an excellent example of how KServe adds value. Bloomberg News is a real-time news feed that has a diminishing time value for its users, so articles it provides must be timely and relevant. Bloomberg uses a massive collection of Natural Language Processing (NLP) models to score incoming news articles from a variety of sources. Each article is labeled, classified, and provided to users through a service they call Terminal. This processing isn't a back office problem that can be done later, and the inferencing must be updated dynamically. Fortunately, KServe allows the models to be updated on the fly. This sort of problem is common in many mobile applications and SaaS products and the ease of integration is key.

Beyond just serving models, KServe also helps manage the lifecycle of machine learning models. One feature, called an explainer, provides further information about each prediction. For example, it can offer insight into why a decision was made to approve or reject a loan application. KServe does this by providing feature importance and

highlighting features in the model that led to the loan decision outcome, such as income level or credit history. Knowing more than just a binary yes or no decision helps build trust in the application. For Machine Learning operations (MLOps) you can use feature importance to detect model drift by integrating KServe with other services to compare results with training data to see if the production model is diverging. You can even include bias detection with AI Fairness, which is now a Linux Foundation incubating project. These features help KServe reduce the effort involved in MLOps.

Machine learning affects all our lives, from food delivery to entertainment. Serving models dynamically in a Kubernetes environment is a crucial step toward integrating machine learning and AI in more and more applications, and KServe will play a large role in making that happen.

Figure 10-2 shows how KServe is deployed on Kubernetes. The control plane consists of the KServe controller, which manages custom resources known as InferenceServices. Each InferenceService instance contains two microservices, a Transformer Service and a Predictor Service, each consisting of a Deployment and a Service. The KNative framework is used for request processing, treating these as serverless microservices that can scale to zero when they are not being used for maximum efficiency.

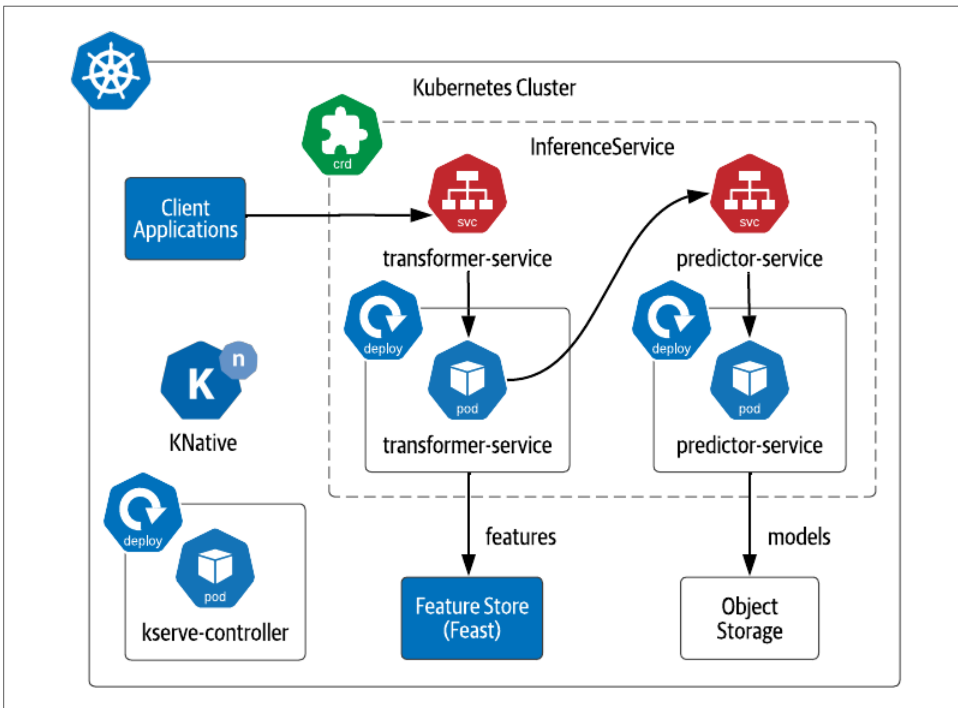


Figure 10-2. Deploying KServe in Kubernetes

www.dbooks.org

The Transformer Service provides the endpoint for prediction requests from client applications. It also implements a three-stage process: preprocessing, prediction, and post-processing:

- **Preprocessing**: the Transformer Service converts the incoming data into a usable form for the model. For example, you may have a model that predicts if a hot dog is in a picture. The Transformer Service will convert an incoming picture to a vector before passing it to the inference service. During preprocessing, the Transformer Service also loads feature data from a feature store such as Feast.
- **Prediction**: the Transformer Service delegates the work of prediction to the Predictor Service, which is responsible for loading the model from object storage and executing it using the provided feature data.
- **Post-processing**: the Transformer Service receives the prediction result and performs any needed post processing to prepare the response to the client application.

If you are familiar with traditional web serving, you can see the helpful analog that model serving creates. Instead of serving HTML pages, KServe covers the modern application needs for serving AI/ML workloads. As a Kuberentes native project it fits seamlessly into your cloud native datacenter and application stack.

Full-Lifecycle Feature Management with Feast

Lifecycle management is a common theme in any data architecture, encompassing how data is added, updated, and deleted over time. Feature stores serve a helpful coordination role by managing the lifecycle of features used by ML models from discovery to their use in production systems, eliminating the versioning and coordination issues that can arise when different teams are involved. How did Feast come to exist?

Sidebar: Bridging Machine Learning Models and Data with Feast

with Willem Pinear, Principal Engineer, Tecton

The Feast project was born from the experiences of the machine learning platform team at GoJek. After building out the core ML tooling, we realized our data scientists were struggling to get models into production. We needed a different kind of tooling to enable the data scientists to help themselves. The same operational rigor we applied to the deployment of traditional data infrastructure was also needed for ML infrastructure. These realizations led to the creation of what we now know as the Feast project. After observing emerging tools from other teams, especially what the Uber team had been doing with Michaelangelo, the idea of a feature store became a first-class priority for us.

To help understand what a feature store does, consider the problem space. GoJek had hundreds of millions of users using a variety of services like ride-hailing, food delivery, or digital payments. Each service had some element of ML which required many steps to go from the back office data science team to production. We used tools like Flink to help with the large-scale SQL batch transformation and stream processing required for model creation, and systems like Redis and Cassandra to serve data, but there were remaining problems to operationalize our ML models. What was needed was a framework on top of those data systems unifying offline and online access, and so the concept of the feature store emerged.

Feature stores serve as a layer to give models a consistent way to access data, effectively providing a bridge between ML models and data within your organization. In production ML models there are two stages: the training and the online phases. Whether the data is coming from a stream, request data, or a data warehouse, your model can't have different copies of data in different environments in each stage. During the training phase, a feature store manages scale requirements for data processing when computing data for model export, similar to other big data tools like Spark. In the online phase, the feature store provides low latency real-time access to models and in some cases, derives features in real-time, also known as on-demand features. Feast ensures the consistency of data for both phases, and it meets both online and offline requirements. Traditional database systems can only provide a subset of those features. For example, Cassandra supports many of the online features, but not offline scalability or specialized features like point-in-time correctness.

Feast began as a place to store computed features, but as we got further into the problem, we also needed to serve those features in production against our models in a consistent way, as an integrated part of our job flow. As the Feast project grew, Google became a key collaborator and within a few months, we had the first working parts of the project. The KubeFlow team at Google suggested we open source the project to make it available to a larger community. With the support of our management, we released a minimum viable product very quickly. So fast, in fact, we released Feast without a lot of things needed to help new users get started, like documentation!

Despite the minimal state, it became clear that Feast met a huge need as a community quickly formed around the project. Teams having similar issues with ML flows were coming to the same realization that they needed an assistant or data platform for operationalizing ML.

In the early days of the project, deploying Feast in Kubernetes included a big stack of components. Today, Feast has evolved to be more lightweight; many of the extra components have been stripped away, making it more efficient and easier to manage. The best approach to building ML platforms on Kubernetes is to make your processing components as stateless as possible and store state externally. The registry doesn't even have to be in Kubernetes since it can just be a file in an object store. Feast is frequently deployed alongside Redis or Cassandra inside Kubernetes and connected externally to data warehouses like BigQuery and Redshift. Providing external access

www.dbooks.org

to Feast is an important aspect. This is typically done using an Ingress to access the API server directly. In other cases KServe is used as an intermediate serving layer to provide a scalable solution when a highly used ML model is used by external services.

The future for Feast is to be more cloud native and fully integrated with Kubernetes. There are quite a few challenges to be solved in deploying ML in Kubernetes, with the biggest being operational maturity. It still takes quite a bit of work for engineers to install many of the components and the day two maintenance is more demanding than it should be. More community involvement will help grow the maturity of machine learning as an emerging part of the Kubernetes data stack.

As Willem noted, the deployment of Feast on Kubernetes is at a basic state of maturity. As no operator or custom resources are defined, you install Feast using a Helm chart. Figure 10-3 shows a sample installation using the example documented on the Feast website, which consists of the Feature Server and other supporting services.

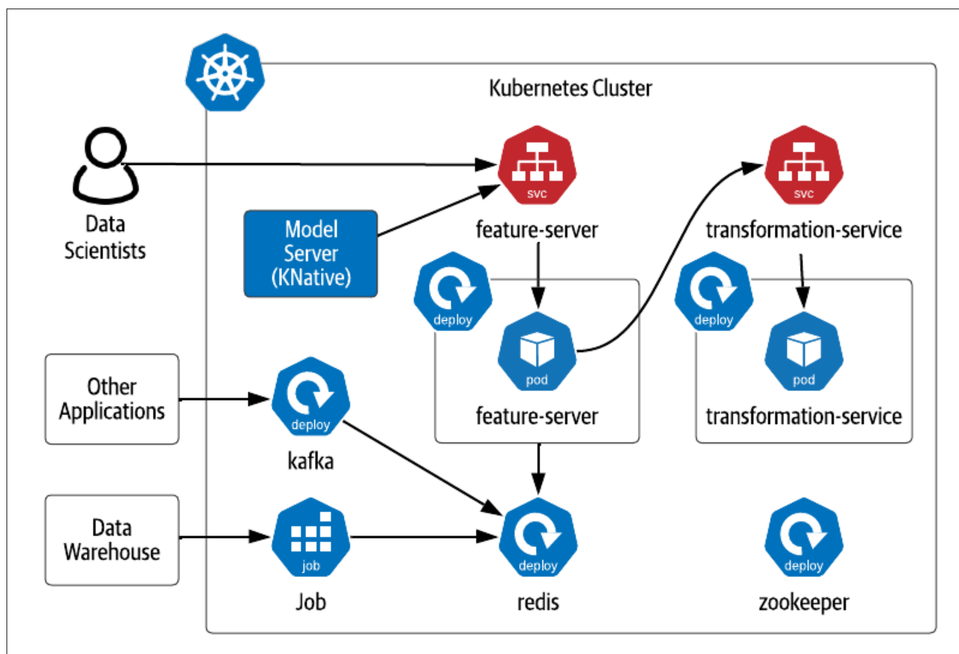


Figure 10-3. Deploying Feast in Kubernetes

Let's examine these components and how they interact. Data Scientists identify features from existing data sources in a process called feature engineering and create features using an interface exposed by the Feature Server. The user can either provide feature data at the time of creating the feature or can connect to various backend services so that the data can be updated continuously. Feast can consume data pub-

lished to Kafka topics, or through Kubernetes jobs which pull data from an external source such as a data warehouse. The feature data is stored in an online database such as Redis or Cassandra so that it can be easily served to production applications. Zookeeper is used to coordinate metadata and service discovery. The Helm chart also supports the ability to deploy Grafana for visualization of metrics. This may sound familiar to you, because the reuse of common building blocks like Redis, Zookeeper, and Grafana is a pattern we've seen used in several other examples in this book.

When model serving tools like KServe are asked to make predictions, they use the features stored in Feast as a record of truth. Any updated training by data scientists is done using the same feature store, eliminating the need for multiple sources of data. The Transformation Service provides an optional capability to generate new features on demand by performing transformations on existing feature data.

KServe and Feast are often used together to create a complete realtime model serving stack. Feast performs the dynamic part of feature management, working with online and offline data storage as new features arrive through streaming and data warehouses. KServe handles the dynamic provisioning for the model serving by using the serverless capabilities of KNative. This means that when not in use, KServe can scale to zero and react when new requests arrive, saving valuable resources in your Kubernetes based AI/ML stack by only using what you need.

Vector Similarity Search with Milvus

Now that we've looked at tools that enable you to use ML models and features in production systems, let's switch gears and look at a different type of AI/ML tool - vector similarity search (VSS). As discussed earlier in the chapter, a vector is a number object representing direction and magnitude. VSS is an application of vector mathematics in machine learning. The K-nearest neighbor (KNN) algorithm is a way to find how "close" two things are next to each other. There are many variations of this algorithm but they all rely on expressing data as a vector. The data to be searched is vectorized using a CPU intensive KNN type algorithm; typically this is more of a backend process. VSS servers can then index the vector data for less CPU-intensive searching and provide a query mechanism that allows end users to provide a vector and find things that are close to it.

Milvus is one of many servers designed around the emerging field of VSS. Let's learn how Milvus came to exist and why it's a great fit for Kubernetes.

Sidebar: A New Era of Search for Kubernetes Applications

with Xiaofan Luan Director of Engineering Zilliz, Milvus Maintainer

There is a growing community around the newly emerging field of VSS, most notably in the use of libraries such as Facebook AI Similarity Search (FAISS) and Hierarchical

www.dbooks.org

Navigable Small World (HNSW). These libraries are used to take the output of computationally expensive Machine Learning algorithms and create end-user applications. Algorithms like Convoluted Neural Networks (CNN) can take data including images and generate vectors that are simply a list of numbers. The real value of the analysis comes from what you do with that list of numbers.

Structured data searching has been a standard feature of traditional RDBMSs in which all the values in columns are indexed for fast lookup. Projects like Apache Lucene built on this, making text search a new kind of competency for unstructured data. Users can provide all or part of the text they are searching for and get back multiple results with varying confidence values. Lucene is the engine for higher-level systems such as Apache Solr and ElasticSearch. Combined, they create a data server that is used in almost every kind of application now.

Milvus was designed to fulfill a similar purpose as Solr and ElasticSearch. However, instead of working with only text, Milvus exposes a general purpose VSS capability. It provides a top-level operational server for users that want more than just a library and need a system that can handle important details like durability, failure, and recovery. Milvus is a system that can be deployed and managed in Kubernetes to manage storage and helper features like computation disaggregation. Most importantly, it provides the Milvus API interface for application developers to use VSS in their code to do things that aren't possible with previous databases.

To give an example of how this works, imagine a library containing photos of meals. Using an image analysis tool such as YOLO, the objects in the images are separated into main dishes such as a sandwich and various side dishes like french fries. The next step is to process each object using ResNet to extract its dimensions. The output of ResNet is a 256-dimensional vector for each item, which is then loaded into Milvus and assigned a unique ID. Milvus indexes the different objects so they can be accessed via its search interface. User-facing applications can provide a picture of a hamburger and fries and ask for similar meals based on the indexed images and the similarities.

Let's compare this example to the experience of using a text search engine like ElasticSearch. To start, you would need a text description of each meal, index that description using Lucene, and then you would be able to search for the words "French Fries." Similar to how ElasticSearch makes searching text easier, Milvus does the same thing to enable the searching of vectorized video, audio, and even natural language text.

Milvus 1.0 was deployed as a single node for storing, indexing, and serving data. This worked for anyone needing a simple package, but it wasn't cloud native or Kubernetes friendly. For the 2.0 release, we decided that Milvus needed to change into a distributed architecture and become more cloud native, separating the compute elements from storage. Our goal was to make Milvus scale horizontally by independent function with an additional benefit of disaster recovery. There are four layers deployed in a Milvus cluster. The access layer, coordinator service, worker node, and storage nodes. Breaking up Milvus into something similar to micro-services reduces the reliance on state to only the storage nodes. The access layer, coordinator service,

and worker nodes are stateless, making the system much easier to scale up and down and eliminating single points of failure. One of the essential features for the Milvus operator in the 2.0 release was the change to object storage and away from StatefulSets. With these updates, Kubernetes is now the preferred way to deploy Milvus.

Milvus is now a graduated project under the governance of the LF AI & Data Foundation. The projects in this foundation are all looking toward a cloud native future for data and the emergence of AI and ML as a core part of every application. The focus for Milvus post 2.0 is performance. Applications based on AI/ML require fast responses and search is a speed dependent operation. Code improvements are a more traditional way of gaining performance, but in the AI/ML world, hardware plays a big part as well. Taking advantage of GPUs or custom FPGA applications will again help developers take advantage of AI/ML performance advances using a simple API. Overall we want to provide an easy path for people building cloud native applications to go from the leading edge to mainstream with a great experience.

As Xiaofan mentioned, Milvus supports both standalone and clustered deployments, using the four layers described above. Both models are supported in Kubernetes via Helm, with the clustered deployment shown in Figure 10-4.

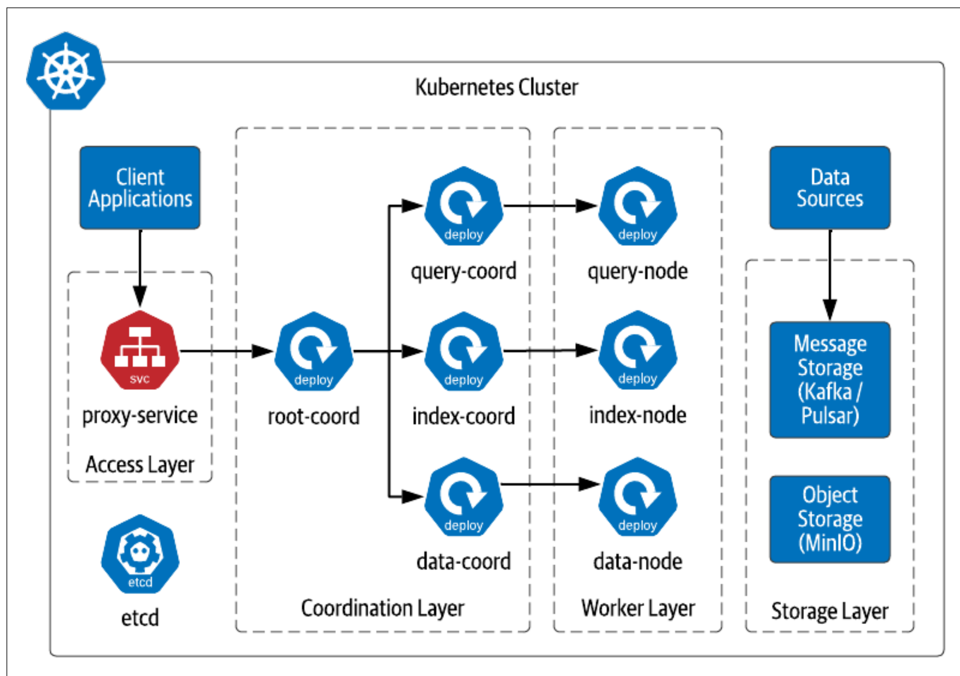


Figure 10-4. Deploying Milvus in Kubernetes

www.dbooks.org

The Access Layer contains the Proxy Service, which uses a Kubernetes Load Balancer to route requests from client applications. The services in the Coordination Layer handle incoming search and index queries, routing them to the core server components in the Worker Layer that handle queries and manage data storage and indexing. The Data Nodes manage persistence via files in object storage. The Message Storage uses Apache Pulsar or Apache Kafka to store the stream of incoming data that is then passed to Data Nodes.

As you can see, Milvus is designed to be Kubernetes native, with a horizontally scalable architecture that makes it well poised to scale up to massive data sets including billions or even trillions of vectors.

Efficient Data Movement with Apache Arrow

Now that we have explored an AI/ML Kubernetes stack that helps you manage compute resources more efficiently, you might be wondering what can be done with network resources? The "Fallacies of distributed computing" we discussed in Chapter 1. include two important points: the fallacies of believing that bandwidth is infinite and that transport cost is zero. Even when compute and storage resources seem much more finite, it's easy to forget how easily you can run out of bandwidth. The deeper you get into deploying your data infrastructure into Kubernetes, the more likely it is you will find out. Early adopters of Apache Hadoop often shared that as their clusters grew, their network switches needed to be replaced with the best that could be purchased at the time. Just consider what it takes to sort 10 terabytes of data. How about 1 petabyte? You get the idea.

Apache Arrow is a project that addresses the problem of bandwidth utilization by providing a more efficient format. This actually isn't an unknown approach in the history of computer science. IBM introduced EBCDIC character encoding to create efficiency for the preferred transport of the time: the punch card. Arrow attacks the problem of efficiency from the ground up in order to avoid the endless upgrading to add more resources, proving that the solution to a control problem is never "add more power." Let's hear from some experts to learn how this works.

Sidebar: Efficient data movement with Apache Arrow

with Josh Patterson, CEO, Voltron, and Keith Kraus, VP of Product, Voltron

As big data technologies like Spark, Kudu, and Cassandra made it possible to move larger amounts of data between systems, it became clear that the computational and performance cost of serializing and de-serializing data was getting too high. Wes McKinney and Jacques Nadeau, along with others, made a bid to address this problem with a project called Apache Arrow. Arrow provides a standard way to represent the layout of data so systems can share that data with fewer headaches.

Arrow uses an in-memory columnar format; that is, data arranged in a tabular format of rows and columns. In traditional relational databases, each record is represented as a row with multiple columns. Arrow pivots this arrangement: data is organized in sequentially ordered columns. This provides significant advantages when searching and processing large amounts of data, especially because of how it aligns with modern CPU architectures.

Arrow Flight is a subproject to bring the same efficiency we see in processing to network communications. Highly connected distributed systems consume network resources quickly and any efficiency gains quickly make a big difference at high volumes. Flight is a Remote Procedure Call (RPC) layer that drastically reduces resource utilization for communications between data services by eliminating serialization costs. Arrow Flight uses gRPC for network efficiency which enables it to send data in parallel using multiple channels. Using Arrow in all of your Kubernetes native analytics stack reduces resource usage and therefore cost.

Arrow doesn't just provide benefits for network utilization, it also has promise for more efficient compute processing for AI/ML workloads. Arrow provides a fast access pattern for data analytics and tabular data that Kubernetes applications can take advantage of. Arrow was one of the first projects in the data analytics space to encourage users to think carefully about the usage of memory and processing hardware, and this timing has coincided nicely with the rise of deep learning. Kubernetes native analytics workloads powered by Arrow will help keep costs low while allowing higher processing volumes.

In fact, Kubernetes was a key driver that moved the Arrow project forward. As the GPU accelerated stack was being defined around 2018, Kubernetes was emerging as an industry standard, replacing Hadoop Yarn as the leading resource management tool for big data processing. The Kubernetes community was developing key features more rapidly, like support for the Remote Direct Memory Access (RDMA) protocol and topology awareness of nodes containing GPUs. Kubernetes also supported faster SLAs for cluster operations. With modern GPUs offering 50 times faster processing times, the job of analyzing dozens of terabytes might take 5 minutes, while scheduling and provisioning the machines with Yarn to perform that job could take 10 minutes. The auto-scaling abilities in Kubernetes offered the right reaction time to match these cyclical workloads. New advanced schedulers such as Yunikorn and Volcano now make those operations even faster and more efficient.

Finding ways to take advantage of new hardware technology is a critical part of the battle we have to keep up with the ever-increasing volumes of data created. The trend toward using GPUs for big data processing is already increasing, and adopting Arrow will only make this easier. In fact, the effect on the community has already reached a tipping point. With the momentum of GPUs adopting the Arrow format, data tools have started adopting Arrow for compatibility, helping to cement Arrow as a data interoperability standard. Arrow could be more than just a language-agnostic con-

www.dbooks.org

nector, but a hardware connector. We've come to believe that an increasing number of systems will become Arrow native in the near future.

The data and analytics ecosystem will continue to drive the future of Kubernetes and Arrow. Frameworks like Dask, and Ray use Python as their underlying compute library, with Arrow used as the format within Pandas data frames sent over the wire between workers. Getting your tabular data efficiently over the wire is a huge benefit and Arrow provides an easy-to-implement standard that is completely interchangeable, open, and widely adopted. It allows future tool developers to focus on the special thing they are building and less on optimizing interconnect.

The Arrow community has become a center of gravity attracting large and innovative projects. The data and analytics community has a pattern of rebuilding new infrastructure about every ten years. This time the revolution is defined not by starting over, but refining the things that we have, biased toward optimizing the primitives. Arrow provides a modular building block that can be used, optimized, extended, and composed with multiple other systems. The groundwork for the next ten years of data infrastructure can start on a sound foundation learned from the mistakes of the past decade. Then we can focus on problems like improving Parquet, using SIMD vectorization, or building storage that could be compacted tightly and quickly. Arrow can be a big part of these solutions because it touches so many systems. Even though its focus on how we represent data is simple, minor improvements here can have massive ripple effects on our cloud native future.

Using Apache Arrow enabled projects enables you to share data efficiently, reducing your resource usage across compute, network and storage. Example usage of Arrow with Apache Spark is shown in Figure 10-5.

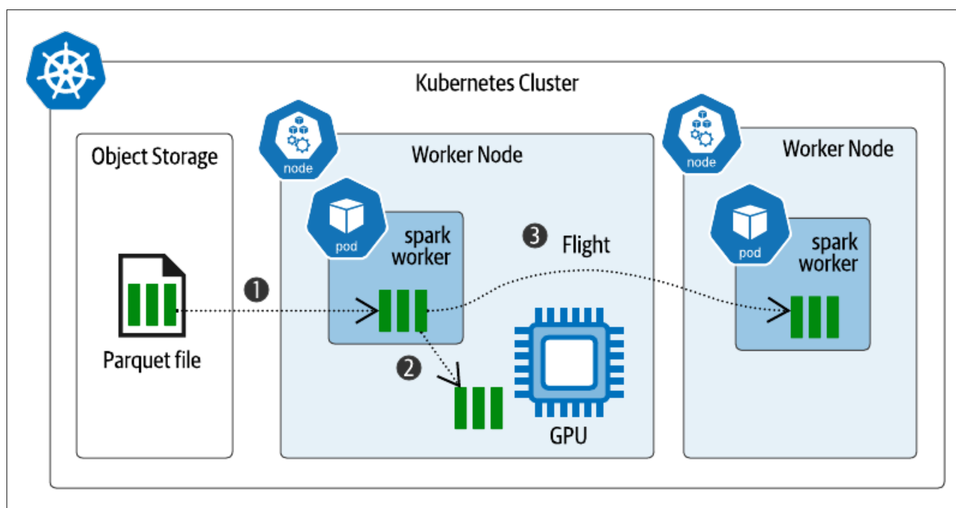


Figure 10-5. Moving Data with Apache Arrow

Parquet data files containing Arrow formatted data persisted to object storage can be easily loaded without a de-serialization step (1). The data can then be analyzed by a Spark application (2), including loading directly into a GPU for processing where available. The same efficiency level is maintained when passing data between worker nodes using Arrow Flight (3). The Arrow record batch is sent without any intermediate memory copying or serialization, and the receiver can reconstruct the Arrow record without memory copy or deserialization. The efficient relationship between the remote processes eliminates two things: processing overhead for sending data and the efficient Arrow record format that eliminates wasted bandwidth.

At the scale common in Spark applications, the effect on network latency and bandwidth can add up quickly. The network transport savings really keep your data moving, even when volumes reach into terabytes and petabytes. Research performed by TU Delft showed a 20 to 30 times efficiency gain using Arrow Flight to move large volumes of data.

Versioned Object Storage with LakeFS

Object storage is becoming the standard for cloud native data persistence. It lowers the complexity for services but also points to a different way of thinking about data mutability. Instead of opening a file and providing random access, file storage is precomputed, written once and read many times. Instead of updating a data file, you write a new one, but how do you distinguish which data files are current? For this reason, object storage presents issues with disk space management. Since there is no concept of managing an entire filesystem, each file is an object in a virtually infinite resource.

Object storage APIs are fairly basic with few frills, but data teams need more than just the basics for their use cases. LakeFS and Nessie are two projects trying to make object storage a better fit for emerging workloads on Kubernetes. Let's examine how LakeFS extends the functionality of object storage for cloud native applications.

Sidebar: Data Integrity to Let You Sleep at Night

Adi Polak, VP of Developer Experience, Treeverse

In working as a full time engineer building big data infrastructure, there were many times I had to manually change data in object storage in our production environment. This task became even more challenging when using complex data formats, such as Parquet. On one occasion, I needed to delete some data files to resolve a production issue. Unfortunately, I accidentally deleted the wrong data. That meant 20 hours in the office with a very grumpy DevOps team trying to recover the data from backup because, of course, it was customer data. At least in this case, we were aware of the

www.dbooks.org

issue. What's even more concerning are the silent failures that impact data products without us even being aware.

These problems occur frequently today because our systems are too delicate. We are biased towards action, but we're human and therefore have a tendency to make mistakes. The result is that bad things happen to good data.

How does LakeFS help with situations like the one described above? The simplest way to describe LakeFS is that it enables Git-like capabilities for object storage. It allows engineers to perform familiar actions like branch, commit, merge and revert. This creates new options for how you use data and enhances workflows.

For example, a typical use case for using LakeFS is Continuous Integration / Continuous Deployment (CI/CD) flows. Data engineers frequently need to reproduce some portion of a data pipeline over different versions of the data, which requires branching. When running on Kubernetes, multiple containers can potentially run the same code over different versions of data. Branching data on the object store creates an isolated environment for experimentation. If there is a mistake in the branch, you can simply revert. This provides the ability to experiment at low cost without harming the original data, which builds trust and allows teams to move faster with safety.

Another example is trying out a new application to see how it fits into the bigger data flow. Git semantics on data can make a massive difference in complicated scenarios that are typically hard to test. Holden Karau has spoken in Chapter 9 about the difficulty in testing big data applications. It's almost impossible to mimic production flows in development or staging environments because of the variety and production data volume. With LakeFS, you can use branching to test with multiple data versions, duplicating the variety and volume seen in production and building confidence in what is being built.

To integrate with your environment, lakeFS exposes an S3 compatible API endpoint through a stateless service. However, it doesn't actually serve as storage. LakeFS forks data commands from your application. Loading data to lakeFS is a metadata operation that creates your main branch in lakeFS by creating pointers to the physical data in your S3 bucket. Any additional branch is an atomic metadata operation pointing to the same data as main when created from it. The metadata created by lakeFS is saved to your S3. As long as the user application is using the lakeFS API endpoint all Git functionality is available. If users want to stop using lakeFS at any point, the original data storage is unaltered and can be used directly by changing the endpoint address used by your application. To roll up any changes while using LakeFS, an offboarding script is available to synchronize any changes before taking LakeFS out of the path. This makes it easy for users to try lakeFS and then adopt or move on without the need to move existing data. The design of lakeFS enables seamless integrations with other parts of your data infrastructure such as Iceberg, Hudi, or Delta Lake, providing the added features of branch, commit, rollback, and merge.

LakeFS addresses the lack of ACID transactions across multiple systems by providing the ability to have versioned object storage. The consistency level guarantees pass-through from the originating application. However, when multi-table operations are performed on an isolated branch, the merge function across all tables is atomic, achieving cross-table consistency.

We are at an interesting intersection point for data workloads on Kubernetes. Many developers who have been working with distributed data workloads for years think in terms of the Hadoop ecosystem. Now we're actually bringing in a different type of developer: the application developer that works with Kubernetes. There is a potential for more friction and errors since these developers are not always aware of the infrastructure and how things have traditionally worked in the big data world.

Kubernetes is now being used to orchestrate the systems that process data and turn it into products for sale. If the data is not protected, your business is at risk. Organizations need to be able to audit, save and deliver data reliably, even if it is at a lower SLA. LakeFS is a great fit for Kubernetes deployment because it assumes that the complexity of distributed systems and distributed data will lead to many mistakes around data. That assumption is met with the assurance that any mistake is easily fixed and never devastating, leading to a great night of sleep for your DevOps teams.

Using LakeFS in Kubernetes is a great fit because of its stateless design and declarative deployment. A Helm deployment consists of configuring the LakeFS service, which then serves as a communication gateway to and from other services.

www.dbooks.org

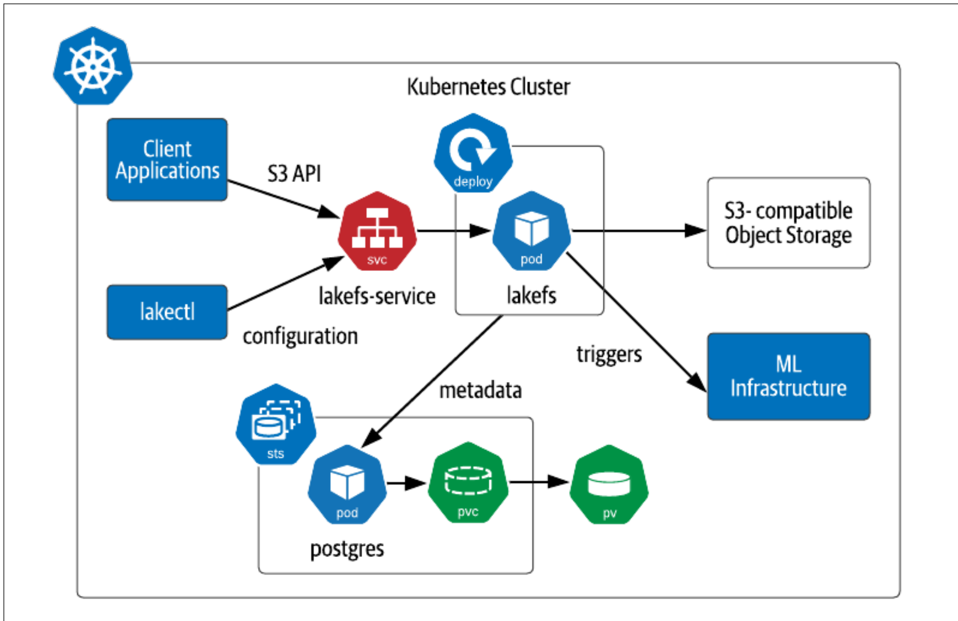


Figure 10-6. Deploying LakeFS in Kubernetes

Communications into the server emulate AWS S3 object storage, enabling interaction with any data store that supports the S3 API. Incoming communication is bound as a ClusterIP to serve HTTP traffic across one or more stateless LakeFS server Pods managed by a Deployment.

LakeFS uses PostgreSQL to manage metadata, so users can either provide the endpoint for a running system, as shown in Figure 10-6, or LakeFS can run an embedded PostgreSQL server inside the LakeFS pod for its exclusive use. PostgreSQL is the state management for the stateless LakeFS servers when deployed as a cluster.

The most important connection is to the object storage endpoints that will store the actual data. When users persist data to LakeFS, the actual data file will pass through to the backend object storage, and versioning metadata is stored in PostgreSQL.

The additional outbound connection is for providing orchestration with other machine learning infrastructure. Webhooks allow for triggers on action that alert downstream systems when something such as a commit is issued. These triggers serve as a key ingredient to automated ML workflows and other applications.

Summary

As you can see, the pipeline of new and exciting ways to work with data in Kubernetes extends well into the future. New projects are addressing the challenges of

advanced data workloads according to the cloud native principles of elasticity, scalability, and self-healing.

These tools give you the ability to manage the critical resources of compute, network and storage. You can better manage compute-intensive workloads such as AI/ML with KServe for the delivery, Feast for model management and Milvus to operationalize new search methods. Network resources are ruled by the simple laws of volume and speed, and at the volumes of data we can create, every little bit helps. Apache Arrow reduces this volume by creating a common reference frame across applications. Unifying around object storage provides further efficiencies, with tools like LakeFS making object storage easier to consume in ways that are sympathetic to application data storage needs.

At this point, we've examined data infrastructure on Kubernetes from mature areas like storage all the way out to cutting-edge projects for managing AI/ML artifacts such as models and features. Now it's time to take all the knowledge you've gained so far and plan to put it into practice.

www.dbooks.org