O'REILLY®

Production Kubernetes

Building Successful Application Platforms

Josh Rosso, Rich Lander,
Alexander Brand & John Harris

O'REILLY®

Production Kubernetes

Although Kubernetes has become the dominant container orchestrator, many organizations relatively new to this system struggle to run actual production workloads. In this practical book, software engineers Josh Rosso, Rich Lander, Alexander Brand, and John Harris from VMware share their experiences running Kubernetes in production and provide insight on key challenges and best practices.

The brilliance of Kubernetes is how configurable and extensible it is, from pluggable runtimes to storage integrations. For platform engineers, software developers, infosec folks, network engineers, storage engineers, and others, this book examines how the path to success with Kubernetes involves a variety of technology, pattern, and abstraction considerations.

With this guide, you will:

- Understand the capabilities required to build a robust Kubernetes-based platform
- Learn from real-world experience to avoid the pitfalls that often occur when building a Kubernetes-based platform
- Discover how Kubernetes's architecture and design enables extensible platform development
- Analyze the concerns of internal and external users to develop a platform that's fit for purpose
- Control the complexity of your Kubernetes platform by making reasoned decisions about tooling and abstractions
- Examine the path to production with Kubernetes and learn about common tooling options and trade-offs

"So much good advice. I wish I'd had this book when designing clusters for the first time."

—**Michael Goodness**
Principal DevOps Engineer, MLB

"This book is a must-read if you are tasked with replatforming or evaluating the effort involved in adopting Kubernetes for infrastructure teams."

—**Duffie Cooley**
CNCF Ambassador

Josh Rosso is an engineer who's worked with Kubernetes at CoreOS (Red Hat), Heptio, and VMware.

Rich Lander is a VMware field engineer who helps enterprises adopt Kubernetes and cloud native technology.

Alexander Brand is a software engineer focused on Kubernetes and cloud native technologies.

John Harris is a staff engineer who's worked on cloud native tooling, platforms, and patterns at VMware (Heptio) and Docker.

KUBERNETES

US $69.99 CAN $92.99
ISBN: 978-1-492-09230-8

Twitter: @oreillymedia
facebook.com/oreilly

Production Kubernetes

Building Successful Application Platforms

*Josh Rosso, Rich Lander,
Alexander Brand, and John Harris*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Production Kubernetes

by Josh Rosso, Rich Lander, Alexander Brand, and John Harris

Copyright © 2021 Josh Rosso, Rich Lander, Alexander Brand, and John Harris. 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: John Devins
Development Editor: Jeff Bleiel
Production Editor: Christopher Faucher
Copyeditor: Kim Cofer
Proofreader: Piper Editorial Consulting, LLC
Indexer: Ellen Troutman
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

March 2021: First Edition

Revision History for the First Edition

2021-03-16: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492092308* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Production 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 VMware Tanzu. See our statement of editorial independence.

978-1-492-09230-8

[LSI]

Table of Contents

Foreword xiii

Preface xv

1. A Path to Production 1
Defining Kubernetes 1
The Core Components 2
Beyond Orchestration—Extended Functionality 4
Kubernetes Interfaces 5
Summarizing Kubernetes 7
Defining Application Platforms 7
The Spectrum of Approaches 8
Aligning Your Organizational Needs 10
Summarizing Application Platforms 11
Building Application Platforms on Kubernetes 12
Starting from the Bottom 13
The Abstraction Spectrum 15
Determining Platform Services 16
The Building Blocks 17
Summary 21

2. Deployment Models 23
Managed Service Versus Roll Your Own 24
Managed Services 24
Roll Your Own 24
Making the Decision 25
Automation 26
Prebuilt Installer 26

Custom Automation 27
Architecture and Topology 28
etcd Deployment Models 28
Cluster Tiers 29
Node Pools 31
Cluster Federation 32
Infrastructure 35
Bare Metal Versus Virtualized 36
Cluster Sizing 39
Compute Infrastructure 41
Networking Infrastructure 42
Automation Strategies 44
Machine Installations 46
Configuration Management 46
Machine Images 46
What to Install 47
Containerized Components 49
Add-ons 50
Upgrades 52
Platform Versioning 52
Plan to Fail 53
Integration Testing 54
Strategies 55
Triggering Mechanisms 60
Summary 61

3. Container Runtime. 63
The Advent of Containers 64
The Open Container Initiative 65
OCI Runtime Specification 65
OCI Image Specification 67
The Container Runtime Interface 69
Starting a Pod 70
Choosing a Runtime 72
Docker 73
containerd 74
CRI-O 75
Kata Containers 76
Virtual Kubelet 77
Summary 78

4. Container Storage. 79
Storage Considerations 80
Access Modes 80
Volume Expansion 81
Volume Provisioning 81
Backup and Recovery 81
Block Devices and File and Object Storage 82
Ephemeral Data 83
Choosing a Storage Provider 83
Kubernetes Storage Primitives 83
Persistent Volumes and Claims 83
Storage Classes 86
The Container Storage Interface (CSI) 87
CSI Controller 88
CSI Node 89
Implementing Storage as a Service 89
Installation 90
Exposing Storage Options 92
Consuming Storage 94
Resizing 96
Snapshots 97
Summary 99

5. Pod Networking. 101
Networking Considerations 102
IP Address Management 102
Routing Protocols 104
Encapsulation and Tunneling 106
Workload Routability 108
IPv4 and IPv6 109
Encrypted Workload Traffic 109
Network Policy 110
Summary: Networking Considerations 112
The Container Networking Interface (CNI) 112
CNI Installation 114
CNI Plug-ins 116
Calico 117
Cilium 120
AWS VPC CNI 123
Multus 125
Additional Plug-ins 126
Summary 126

6. Service Routing. 127
Kubernetes Services 128
The Service Abstraction 128
Endpoints 135
Service Implementation Details 138
Service Discovery 148
DNS Service Performance 151
Ingress 152
The Case for Ingress 153
The Ingress API 154
Ingress Controllers and How They Work 156
Ingress Traffic Patterns 157
Choosing an Ingress Controller 161
Ingress Controller Deployment Considerations 162
DNS and Its Role in Ingress 165
Handling TLS Certificates 166
Service Mesh 169
When (Not) to Use a Service Mesh 169
The Service Mesh Interface (SMI) 170
The Data Plane Proxy 173
Service Mesh on Kubernetes 175
Data Plane Architecture 179
Adopting a Service Mesh 181
Summary 184

7. Secret Management. 187
Defense in Depth 188
Disk Encryption 189
Transport Security 190
Application Encryption 190
The Kubernetes Secret API 191
Secret Consumption Models 193
Secret Data in etcd 196
Static-Key Encryption 198
Envelope Encryption 201
External Providers 203
Vault 203
Cyberark 203
Injection Integration 204
CSI Integration 208
Secrets in the Declarative World 210
Sealing Secrets 211

Sealed Secrets Controller 211
Key Renewal 214
Multicluster Models 215
Best Practices for Secrets 215
Always Audit Secret Interaction 215
Don't Leak Secrets 216
Prefer Volumes Over Environment Variables 216
Make Secret Store Providers Unknown to Your Application 216
Summary 217

8. Admission Control. 219
The Kubernetes Admission Chain 220
In-Tree Admission Controllers 222
Webhooks 223
Configuring Webhook Admission Controllers 225
Webhook Design Considerations 227
Writing a Mutating Webhook 228
Plain HTTPS Handler 229
Controller Runtime 231
Centralized Policy Systems 234
Summary 241

9. Observability. 243
Logging Mechanics 244
Container Log Processing 244
Kubernetes Audit Logs 247
Kubernetes Events 249
Alerting on Logs 250
Security Implications 251
Metrics 251
Prometheus 251
Long-Term Storage 253
Pushing Metrics 253
Custom Metrics 253
Organization and Federation 254
Alerts 255
Showback and Chargeback 257
Metrics Components 260
Distributed Tracing 269
OpenTracing and OpenTelemetry 269
Tracing Components 270
Application Instrumentation 272

Service Meshes 272
Summary 272

10. **Identity** 273
User Identity 274
Authentication Methods 275
Implementing Least Privilege Permissions for Users 285
Application/Workload Identity 288
Shared Secrets 289
Network Identity 289
Service Account Tokens (SAT) 293
Projected Service Account Tokens (PSAT) 297
Platform Mediated Node Identity 299
Summary 311

11. **Building Platform Services** 313
Points of Extension 314
Plug-in Extensions 314
Webhook Extensions 315
Operator Extensions 316
The Operator Pattern 317
Kubernetes Controllers 317
Custom Resources 318
Operator Use Cases 323
Platform Utilities 323
General-Purpose Workload Operators 324
App-Specific Operators 324
Developing Operators 325
Operator Development Tooling 325
Data Model Design 329
Logic Implementation 331
Extending the Scheduler 347
Predicates and Priorities 348
Scheduling Policies 348
Scheduling Profiles 350
Multiple Schedulers 350
Custom Scheduler 350
Summary 351

12. **Multitenancy** 353
Degrees of Isolation 354
Single-Tenant Clusters 354

Multitenant Clusters 355
The Namespace Boundary 357
Multitenancy in Kubernetes 358
Role-Based Access Control (RBAC) 358
Resource Quotas 360
Admission Webhooks 361
Resource Requests and Limits 363
Network Policies 368
Pod Security Policies 370
Multitenant Platform Services 374
Summary 375

13. Autoscaling. 377
Types of Scaling 378
Application Architecture 379
Workload Autoscaling 380
Horizontal Pod Autoscaler 380
Vertical Pod Autoscaler 384
Autoscaling with Custom Metrics 387
Cluster Proportional Autoscaler 388
Custom Autoscaling 389
Cluster Autoscaling 389
Cluster Overprovisioning 393
Summary 395

14. Application Considerations. 397
Deploying Applications to Kubernetes 398
Templating Deployment Manifests 398
Packaging Applications for Kubernetes 399
Ingesting Configuration and Secrets 400
Kubernetes ConfigMaps and Secrets 400
Obtaining Configuration from External Systems 403
Handling Rescheduling Events 404
Pre-stop Container Life Cycle Hook 404
Graceful Container Shutdown 405
Satisfying Availability Requirements 407
State Probes 408
Liveness Probes 409
Readiness Probes 410
Startup Probes 411
Implementing Probes 412
Pod Resource Requests and Limits 413

Resource Requests 413
Resource Limits 414
Application Logs 415
What to Log 415
Unstructured Versus Structured Logs 416
Contextual Information in Logs 416
Exposing Metrics 416
Instrumenting Applications 417
USE Method 419
RED Method 419
The Four Golden Signals 419
App-Specific Metrics 419
Instrumenting Services for Distributed Tracing 420
Initializing the Tracer 420
Creating Spans 421
Propagate Context 422
Summary 423

15. Software Supply Chain. 425
Building Container Images 426
The Golden Base Images Antipattern 428
Choosing a Base Image 429
Runtime User 430
Pinning Package Versions 430
Build Versus Runtime Image 431
Cloud Native Buildpacks 432
Image Registries 434
Vulnerability Scanning 435
Quarantine Workflow 437
Image Signing 438
Continuous Delivery 439
Integrating Builds into a Pipeline 440
Push-Based Deployments 443
Rollout Patterns 445
GitOps 446
Summary 448

16. Platform Abstractions. 449
Platform Exposure 450
Self-Service Onboarding 451
The Spectrum of Abstraction 453
Command-Line Tooling 454

Abstraction Through Templating 455
Abstracting Kubernetes Primitives 458
Making Kubernetes Invisible 462
Summary 464

Index. 465

Foreword

It has been more than six years since we publicly released Kubernetes. I was there at the start and actually submitted the first commit to the Kubernetes project. (That isn't as impressive as it sounds! It was a maintenance task as part of creating a clean repo for public release.) I can confidently say that the success Kubernetes has seen is something we had hoped for but didn't really expect. That success is based on a large community of dedicated and welcoming contributors along with a set of practitioners who bridge the gap to the real world.

I'm lucky enough to have worked with the authors of *Production Kubernetes* at the startup (Heptio) that I cofounded with the mission to bring Kubernetes to typical enterprises. The success of Heptio is, in large part, due to my colleagues' efforts in creating a direct connection with real users of Kubernetes who are solving real problems. I'm grateful to each one of them. This book captures that on-the-ground experience to give teams the tools they need to really make Kubernetes work in a production environment.

My entire professional career has been based on building systems aimed at application teams and developers. It started with Microsoft Internet Explorer and then continued with Windows Presentation Foundation and then moved to cloud with Google Compute Engine and Kubernetes. Again and again I've seen those building platforms suffer from what I call "The Platform Builder's Curse." The people who are building the platforms are focused on a longer time horizon and the challenge of building a foundation that will, hopefully, last decades. But that focus creates a blind spot to the problems that users are having *right now*. Oftentimes we are so busy building a thing we don't have the time and problems that lead us to actually use the thing we are building.

The only way to defeat the platform builder's curse is to actively seek information from outside our platform-builder bubble. This is what the Heptio Field Engineering team (and later the VMware Kubernetes Architecture Team—KAT) did for me. Beyond helping a wide variety of customers across industries be successful with Kubernetes, the team is a critical window into the reality of how the "theory" of our platform is applied.

This problem is only exacerbated by the thriving ecosystem that has been built up around Kubernetes and the Cloud Native Computing Foundation (CNCF). This includes both projects that are part of the CNCF and those that are in the larger orbit. I describe this ecosystem as "beautiful chaos." It is a rainforest of projects with varying degrees of overlap and maturity. This is what innovation looks like! But, just like exploring a rainforest, exploring this ecosystem requires dedication and time, and it comes with risks. New users to the world of Kubernetes often don't have the time or capacity to become experts in the larger ecosystem.

Production Kubernetes maps out the parts of that ecosystem, when individual tools and projects are appropriate, and demonstrates how to evaluate the right tool for the problems the reader is facing. This advice goes beyond just telling readers to use a particular tool. It is a larger framework for understanding the problem a class of tools solves, knowing whether you have that problem, being familiar with the strengths and weaknesses to different approaches, and offering practical advice for getting going. For those looking to take Kubernetes into production, this information is gold!

In conclusion, I'd like to send a big "Thank You" to Josh, Rich, Alex, and John. Their experience has made many customers directly successful, has taught me a lot about the thing that we started more than six years ago, and now, through this book, will provide critical advice to countless more users.

— Joe Beda
Principal Engineer for VMware Tanzu,
Cocreator of Kubernetes,
Seattle, January 2021

Preface

Kubernetes is a remarkably powerful technology and has achieved a meteoric rise in popularity. It has formed the basis for genuine advances in the way we manage software deployments. API-driven software and distributed systems were well established, if not widely adopted, when Kubernetes emerged. It delivered excellent renditions of these principles, which are foundational to its success, but it also delivered something else that is vital. In the recent past, software that autonomously converged on declared, desired state was possible only in giant technology companies with the most talented engineering teams. Now, highly available, self-healing, autoscaling software deployments are within reach of every organization, thanks to the Kubernetes project. There is a future in front of us where software systems accept broad, high-level directives from us and execute upon them to deliver desired outcomes by discovering conditions, navigating changing obstacles, and repairing problems without our intervention. Furthermore, these systems will do it faster and more reliably than we ever could with manual operations. Kubernetes has brought us all much closer to that future. However, that power and capability comes at the cost of some additional complexity. The desire to share our experiences helping others navigate that complexity is why we decided to write this book.

You should read this book if you want to use Kubernetes to build a production-grade application platform. If you are looking for a book to help you get started with Kubernetes, or a text on how Kubernetes works, this is not the right book. There is a wealth of information on these subjects in other books, in the official documentation, and in countless blog posts and the source code itself. We recommend pairing the consumption of this book with your own research and testing for the solutions we discuss, so we rarely dive deeply into *step-by-step* tutorial style examples. We try to cover as much theory as necessary and leave most of the *implementation* as an exercise to the reader.

Throughout this book, you'll find guidance in the form of options, tooling, patterns, and practices. It's important to read this guidance with an understanding of how the authors view the practice of building application platforms. We are engineers and architects who get deployed across many Fortune 500 companies to help them take their platform aspirations from idea to production. We have been using Kubernetes as the foundation for getting there since as early as 2015, when Kubernetes reached `1.0`. We have tried as much as possible to focus on patterns and philosophy rather than on tools, as new tooling appears quicker than we can write! However, we inevitably have to demonstrate those patterns with the most appropriate tool du jour.

We have had major successes guiding teams through their cloud native journey to completely transform how they build and deliver software. That said, we have also had our doses of failure. A common reason for failure is an organization's misconception of what Kubernetes will solve for. This is why we dive so deep into the concept early on. Over this time we've found several areas to be especially interesting for our customers. Conversations that help customers get further on their path to production, or even help them define it, have become routine. These conversations became so common that we decided maybe it's time to write a book!

While we've made this journey to production with organizations time and time again, there is only one key consistency across them. This is that the road *never* looks the same, no matter how badly we sometimes want it to. With this in mind, we want to set the expectation that if you're going into this book looking for the "5-step program" for getting to production or the "10 things every Kubernetes user should know," you're going to be frustrated. We're here to talk about the many decision points and the traps we've seen, and to back it up with concrete examples and anecdotes when appropriate. Best practices exist but must always be viewed through the lens of pragmatism. There is no one-size-fits-all approach, and "It depends" is an entirely valid answer to many of the questions you'll inevitably confront on the journey.

That said, we highly encourage you to *challenge* this book! When working with clients we're always encouraging them to challenge and augment our guidance. Knowledge is fluid, and we are always updating our approaches based on new features, information, and constraints. You should continue that trend; as the cloud native space continues to evolve, you'll certainly decide to take alternative roads from what we recommended. We're here to tell you about the ones we've been down so you can weigh our perspective against your own.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

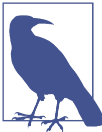
`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Kubernetes kinds are capitalized, as in Pod, Service, and StatefulSet.

This element signifies a tip or suggestion.



This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download and discussion at *https://github.com/production-kubernetes*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Production Kubernetes* by Josh Rosso, Rich Lander, Alexander Brand, and John Harris (O'Reilly). Copyright 2021 Josh Rosso, Rich Lander, Alexander Brand, and John Harris, 978-1-492-09231-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/production-kubernetes*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

The authors would like to thank Katie Gamanji, Michael Goodness, Jim Weber, Jed Salazar, Tony Scully, Monica Rodriguez, Kris Dockery, Ralph Bankston, Steve Sloka, Aaron Miller, Tunde Olu-Isa, Alex Withrow, Scott Lowe, Ryan Chapple, and Kenan Dervisevic for their reviews and feedback on the manuscript. Thanks to Paul Lundin for encouraging the development of this book and for building the incredible Field Engineering team at Heptio. Everyone on the team has contributed in some way by collaborating on and developing many of the ideas and experiences we cover over the next 450 pages. Thanks also to Joe Beda, Scott Buchanan, Danielle Burrow, and Tim Coventry-Cox at VMware for their support as we initiated and developed this project. Finally, thanks to John Devins, Jeff Bleiel, and Christopher Faucher at O'Reilly for their ongoing support and feedback.

The authors would also like to personally thank the following people:

Josh: I would like to thank Jessica Appelbaum for her absurd levels of support, specifically blueberry pancakes, while I dedicated my time to this book. I'd also like to thank my mom, Angela, and dad, Joe, for being my foundation growing up.

Rich: I would like to thank my wife, Taylor, and children, Raina, Jasmine, Max, and John, for their support and understanding while I took time to work on this book. I would also like to thank my Mum, Jenny, and my Dad, Norm, for being great role models.

Alexander: My love and thanks to my amazing wife, Anais, who was incredibly supportive as I dedicated time to writing this book. I also thank my family, friends, and colleagues who have helped me become who I am today.

John: I'd like to thank my beautiful wife, Christina, for her love and patience during my work on this book. Also thanks to my close friends and family for their ongoing support and encouragement over the years.

CHAPTER 1
A Path to Production

Over the years, the world has experienced wide adoption of Kubernetes within organizations. Its popularity has unquestionably been accelerated by the proliferation of containerized workloads and microservices. As operations, infrastructure, and development teams arrive at this inflection point of needing to build, run, and support these workloads, several are turning to Kubernetes as part of the solution. Kubernetes is a fairly young project relative to other, massive, open source projects such as Linux. Evidenced by many of the clients we work with, it is still early days for most users of Kubernetes. While many organizations have an existing Kubernetes footprint, there are far fewer that have reached production and even less operating at scale. In this chapter, we are going to set the stage for the journey many engineering teams are on with Kubernetes. Specifically, we are going to chart out some key considerations we look at when defining a path to production.

Defining Kubernetes

Is Kubernetes a platform? Infrastructure? An application? There is no shortage of thought leaders who can provide you their precise definition of what Kubernetes is. Instead of adding to this pile of opinions, let's put our energy into clarifying the problems Kubernetes solves. Once defined, we will explore how to build atop this feature set in a way that moves us toward production outcomes. The ideal state of "Production Kubernetes" implies that we have reached a state where workloads are successfully serving production traffic.

The name *Kubernetes* can be a bit of an umbrella term. A quick browse on GitHub reveals the `kubernetes` organization contains (at the time of this writing) 69 repositories. Then there is `kubernetes-sigs`, which holds around 107 projects. And don't get us started on the hundreds of Cloud Native Compute Foundation (CNCF) projects that play in this landscape! For the sake of this book, *Kubernetes* will refer exclusively

to the core project. So, what is the core? The core project is contained in the kubernetes/kubernetes repository. This is the location for the key components we find in most Kubernetes clusters. When running a cluster with these components, we can expect the following functionality:

- Scheduling workloads across many hosts
- Exposing a declarative, extensible, API for interacting with the system
- Providing a CLI, `kubectl`, for humans to interact with the API server
- Reconciliation from current state of objects to desired state
- Providing a basic service abstraction to aid in routing requests to and from workloads
- Exposing multiple interfaces to support pluggable networking, storage, and more

These capabilities create what the project itself claims to be, a *production-grade container orchestrator*. In simpler terms, Kubernetes provides a way for us to run and schedule containerized workloads on multiple hosts. Keep this primary capability in mind as we dive deeper. Over time, we hope to prove how this capability, while foundational, is only part of our journey to production.

The Core Components

What are the components that provide the functionality we have covered? As we have mentioned, core components reside in the `kubernetes/kubernetes` repository. Many of us consume these components in different ways. For example, those running managed services such as Google Kubernetes Engine (GKE) are likely to find each component present on hosts. Others may be downloading binaries from repositories or getting signed versions from a vendor. Regardless, anyone can download a Kubernetes release from the `kubernetes/kubernetes` repository. After downloading and unpacking a release, binaries may be retrieved using the `cluster/get-kube-binaries.sh` command. This will auto-detect your target architecture and download server and client components. Let's take a look at this in the following code, and then explore the key components:

```
$ ./cluster/get-kube-binaries.sh

Kubernetes release: v1.18.6
Server: linux/amd64  (to override, set KUBERNETES_SERVER_ARCH)
Client: linux/amd64  (autodetected)

Will download kubernetes-server-linux-amd64.tar.gz from https://dl.k8s.io/v1.18.6
Will download and extract kubernetes-client-linux-amd64.tar.gz
Is this ok? [Y]/n
```

Inside the downloaded server components, likely saved to *server/kubernetes-server-${ARCH}.tar.gz*, you'll find the key items that compose a Kubernetes cluster:

API Server
: The primary interaction point for all Kubernetes components and users. This is where we get, add, delete, and mutate objects. The API server delegates state to a backend, which is most commonly etcd.

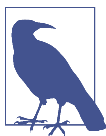
kubelet
: The on-host agent that communicates with the API server to report the status of a node and understand what workloads should be scheduled on it. It communicates with the host's container runtime, such as Docker, to ensure workloads scheduled for the node are started and healthy.

Controller Manager
: A set of controllers, bundled in a single binary, that handle reconciliation of many core objects in Kubernetes. When desired state is declared, e.g., three replicas in a Deployment, a controller within handles the creation of new Pods to satisfy this state.

Scheduler
: Determines where workloads should run based on what it thinks is the optimal node. It uses filtering and scoring to make this decision.

Kube Proxy
: Implements Kubernetes services providing virtual IPs that can route to backend Pods. This is accomplished using a packet filtering mechanism on a host such as `iptables` or `ipvs`.

While not an exhaustive list, these are the primary components that make up the core functionality we have discussed. Architecturally, Figure 1-1 shows how these components play together.



Kubernetes architectures have many variations. For example, many clusters run kube-apiserver, kube-scheduler, and kube-controller-manager as containers. This means the control-plane may also run a container-runtime, kubelet, and kube-proxy. These kinds of deployment considerations will be covered in the next chapter.

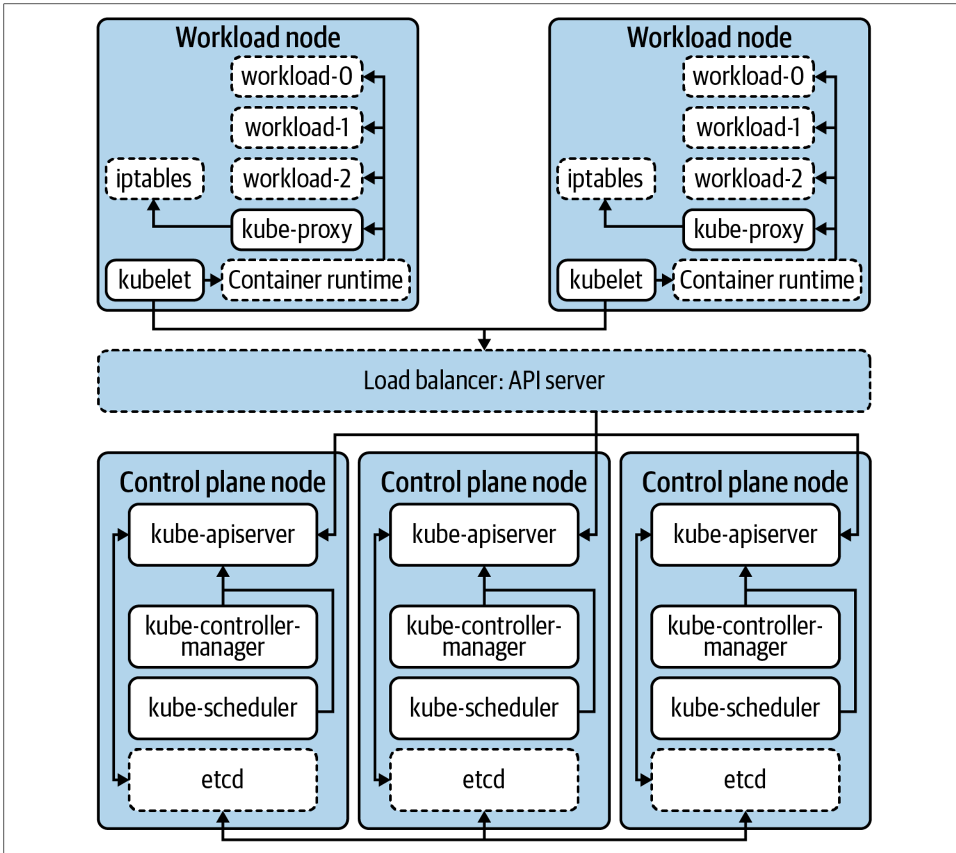


Figure 1-1. The primary components that make up the Kubernetes cluster. Dashed borders represent components that are not part of core Kubernetes.

Beyond Orchestration—Extended Functionality

There are areas where Kubernetes does more than just orchestrate workloads. As mentioned, the component kube-proxy programs hosts to provide a virtual IP (VIP) experience for workloads. As a result, internal IP addresses are established and route to one or many underlying Pods. This concern certainly goes beyond running and scheduling containerized workloads. In theory, rather than implementing this as part of core Kubernetes, the project could have defined a Service API and required a plug-in to implement the Service abstraction. This approach would require users to choose between a variety of plug-ins in the ecosystem rather than including it as core functionality.

This is the model many Kubernetes APIs, such as Ingress and NetworkPolicy, take. For example, creation of an Ingress object in a Kubernetes cluster does not guarantee

action is taken. In other words, while the API exists, it is not core functionality. Teams must consider what technology they'd like to plug in to implement this API. For Ingress, many use a controller such as ingress-nginx, which runs in the cluster. It implements the API by reading Ingress objects and creating NGINX configurations for NGINX instances pointed at Pods. However, `ingress-nginx` is one of many options. Project Contour implements the same Ingress API but instead programs instances of envoy, the proxy that underlies Contour. Thanks to this pluggable model, there are a variety of options available to teams.

Kubernetes Interfaces

Expanding on this idea of adding functionality, we should now explore interfaces. Kubernetes interfaces enable us to customize and build on the core functionality. We consider an interface to be a definition or contract on how something can be interacted with. In software development, this parallels the idea of defining functionality, which classes or structs may implement. In systems like Kubernetes, we deploy plug-ins that satisfy these interfaces, providing functionality such as networking.

A specific example of this interface/plug-in relationship is the Container Runtime Interface (CRI). In the early days of Kubernetes, there was a single container runtime supported, Docker. While Docker is still present in many clusters today, there is growing interest in using alternatives such as containerd or CRI-O. Figure 1-2 demonstrates this relationship with these two container runtimes.

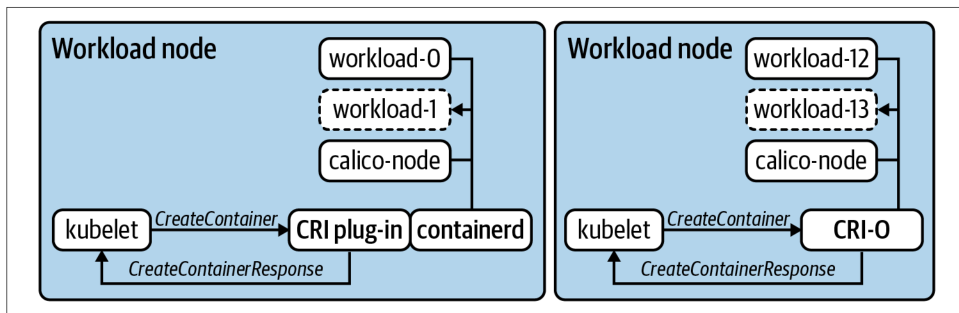


Figure 1-2. Two workload nodes running two different container runtimes. The kubelet sends commands defined in the CRI such as `CreateContainer` and expects the runtime to satisfy the request and respond.

In many interfaces, commands, such as `CreateContainerRequest` or `PortForwardRequest`, are issued as remote procedure calls (RPCs). In the case of CRI, the communication happens over GRPC and the kubelet expects responses such as `CreateContainerResponse` and `PortForwardResponse`. In Figure 1-2, you'll also notice two different models for satisfying CRI. CRI-O was built from the ground up as an implementation of CRI. Thus the kubelet issues these commands directly to it.

containerd supports a plug-in that acts as a shim between the kubelet and its own interfaces. Regardless of the exact architecture, the key is getting the container runtime to execute, without the kubelet needing to have operational knowledge of how this occurs for *every possible* runtime. This concept is what makes interfaces so powerful in how we architect, build, and deploy Kubernetes clusters.

Over time, we've even seen some functionality removed from the core project in favor of this plug-in model. These are things that historically existed "in-tree," meaning within the `kubernetes/kubernetes` code base. An example of this is cloud-provider integrations (CPIs). Most CPIs were traditionally baked into components such as the kube-controller-manager and the kubelet. These integrations typically handled concerns such as provisioning load balancers or exposing cloud provider metadata. Sometimes, especially prior to the creation of the Container Storage Interface (CSI), these providers provisioned block storage and made it available to the workloads running in Kubernetes. That's a lot of functionality to live in Kubernetes, not to mention it needs to be re-implemented for every possible provider! As a better solution, support was moved into its own interface model, e.g., kubernetes/cloud-provider, that can be implemented by multiple projects or vendors. Along with minimizing sprawl in the Kubernetes code base, this enables CPI functionality to be managed out of band of the core Kubernetes clusters. This includes common procedures such as upgrades or patching vulnerabilities.

Today, there are several interfaces that enable customization and additional functionality in Kubernetes. What follows is a high-level list, which we'll expand on throughout chapters in this book:

- The Container Networking Interface (CNI) enables networking providers to define how they do things from IPAM to actual packet routing.
- The Container Storage Interface (CSI) enables storage providers to satisfy intra-cluster workload requests. Commonly implemented for technologies such as ceph, vSAN, and EBS.
- The Container Runtime Interface (CRI) enables a variety of runtimes, common ones including Docker, containerd, and CRI-O. It also has enabled a proliferation of less traditional runtimes, such as firecracker, which leverages KVM to provision a minimal VM.
- The Service Mesh Interface (SMI) is one of the newer interfaces to hit the Kubernetes ecosystem. It hopes to drive consistency when defining things such as traffic policy, telemetry, and management.
- The Cloud Provider Interface (CPI) enables providers such as VMware, AWS, Azure, and more to write integration points for their cloud services with Kubernetes clusters.

- The Open Container Initiative Runtime Spec. (OCI) standardizes image formats ensuring that a container image built from one tool, when compliant, can be run in any OCI-compliant container runtime. This is not directly tied to Kubernetes but has been an ancillary help in driving the desire to have pluggable container runtimes (CRI).

Summarizing Kubernetes

Now we have focused in on the scope of Kubernetes. It is a *container orchestrator*, with a couple extra features here and there. It also has the ability to be extended and customized by leveraging plug-ins to interfaces. Kubernetes can be foundational for many organizations looking for an elegant means of running their applications. However, let's take a step back for a moment. If we were to take the current systems used to run applications in your organization and replace them with Kubernetes, would that be enough? For many of us, there is much more involved in the components and machinery that make up our current "application platform."

Historically, we have witnessed a lot of pain when organizations hold the view of having a "Kubernetes" strategy—or when they assume that Kubernetes will be an adequate forcing function for modernizing how they build and run software. Kubernetes is a technology, a great one, but it really should not be the focal point of where you're headed in the modern infrastructure, platform, and/or software realm. We apologize if this seems obvious, but you'd be surprised how many executive or higher-level architects we talk to who believe that Kubernetes, by itself, is the answer to problems, when in actuality their problems revolve around application delivery, software development, or organizational/people issues. Kubernetes is best thought of as a piece of your puzzle, one that enables you to deliver platforms for your applications. We have been dancing around this idea of an application platform, which we'll explore next.

Defining Application Platforms

In our path to production, it is key that we consider the idea of an application platform. We define an application platform as a viable place to run workloads. Like most definitions in this book, how that's satisfied will vary from organization to organization. Targeted outcomes will be vast and desirable to different parts of the business—for example, happy developers, reduction of operational costs, and quicker feedback loops in delivering software are a few. The application platform is often where we find ourselves at the intersection of apps and infrastructure. Concerns such as developer experience (devx) are typically a key tenet in this area.

Application platforms come in many shapes and sizes. Some largely abstract underlying concerns such as the IaaS (e.g., AWS) or orchestrator (e.g., Kubernetes). Heroku is a great example of this model. With it you can easily take a project written in languages like Java, PHP, or Go and, using one command, deploy them to production.

Alongside your app runs many platform services you'd otherwise need to operate yourself. Things like metrics collection, data services, and continuous delivery (CD). It also gives you primitives to run highly available workloads that can easily scale. Does Heroku use Kubernetes? Does it run its own datacenters or run atop AWS? Who cares? For Heroku users, these details aren't important. What's important is delegating these concerns to a provider or platform that enables developers to spend more time solving business problems. This approach is not unique to cloud services. RedHat's OpenShift follows a similar model, where Kubernetes is more of an implementation detail and developers and platform operators interact with a set of abstractions on top.

Why not stop here? If platforms like Cloud Foundry, OpenShift, and Heroku have solved these problems for us, why bother with Kubernetes? A major trade-off to many prebuilt application platforms is the need to conform to their view of the world. Delegating ownership of the underlying system takes a significant operational weight off your shoulders. At the same time, if how the platform approaches concerns like service discovery or secret management does not satisfy your organizational requirements, you may not have the control required to work around that issue. Additionally, there is the notion of vendor or opinion lock-in. With abstractions come opinions on how your applications should be architected, packaged, and deployed. This means that moving to another system may not be trivial. For example, it's significantly easier to move workloads between Google Kubernetes Engine (GKE) and Amazon Elastic Kubernetes Engine (EKS) than it is between EKS and Cloud Foundry.

The Spectrum of Approaches

At this point, it is clear there are several approaches to establishing a successful application platform. Let's make some big assumptions for the sake of demonstration and evaluate theoretical trade-offs between approaches. For the average company we work with, say a mid to large enterprise, Figure 1-3 shows an arbitrary evaluation of approaches.

In the bottom-left quadrant, we see deploying Kubernetes clusters themselves, which has a relatively low engineering effort involved, especially when managed services such as EKS are handling the control plane for you. These are lower on production readiness because most organizations will find that more work needs to be done on top of Kubernetes. However, there are use cases, such as teams that use dedicated cluster(s) for their workloads, that may suffice with just Kubernetes.

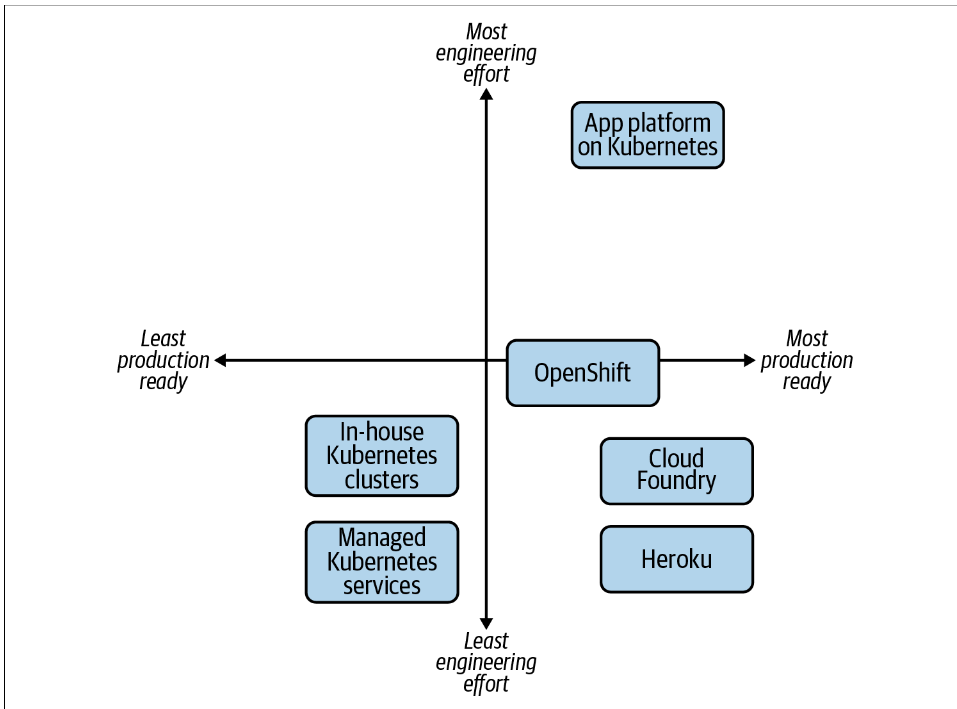


Figure 1-3. The multitude of options available to provide an application platform to developers.

In the bottom right, we have the more established platforms, ones that provide an end-to-end developer experience out of the box. Cloud Foundry is a great example of a project that solves many of the application platform concerns. Running software in Cloud Foundry is more about ensuring the software fits within its opinions. OpenShift, on the other hand, which for most is far more production-ready than just Kubernetes, has more decision points and considerations for how you set it up. Is this flexibility a benefit or a nuisance? That's a key consideration for you.

Lastly, in the top right, we have building an application platform on top of Kubernetes. Relative to the others, this unquestionably requires the most engineering effort, at least from a platform perspective. However, taking advantage of Kubernetes extensibility means you can create something that lines up with your developer, infrastructure, and business needs.

Aligning Your Organizational Needs

What's missing from the graph in Figure 1-3 is a third dimension, a z-axis that demonstrates how aligned the approach is with your requirements. Let's examine another visual representation. Figure 1-4 maps out how this might look when considering platform alignment with organizational needs.

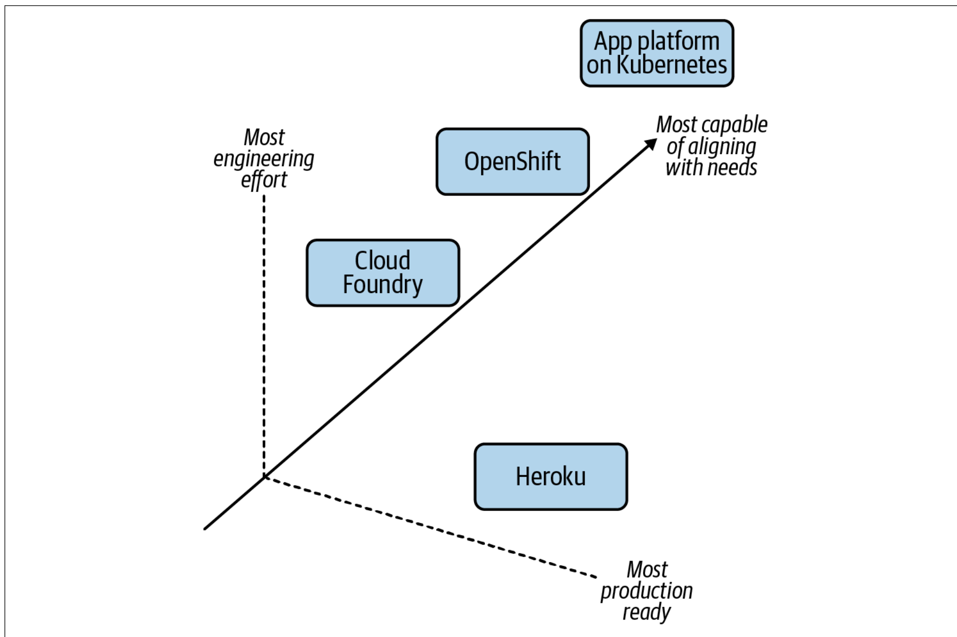


Figure 1-4. The added complexity of the alignment of these options with your organizational needs, the z-axis.

In terms of requirements, features, and behaviors you'd expect out of a platform, building a platform is almost always going to be the most aligned. Or at least the most capable of aligning. This is because you can build anything! If you wanted to re-implement Heroku in-house, on top of Kubernetes, with minor adjustments to its capabilities, it is technically possible. However, the cost/reward should be weighed out with the other axes (x and y). Let's make this exercise more concrete by considering the following needs in a next-generation platform:

- Regulations require you to run mostly on-premise
- Need to support your baremetal fleet along with your vSphere-enabled datacenter
- Want to support growing demand for developers to package applications in containers

- Need ways to build self-service API mechanisms that move you away from "ticket-based" infrastructure provisioning
- Want to ensure APIs you're building atop of are vendor agnostic and not going to cause lock-in because it has cost you millions in the past to migrate off these types of systems
- Are open to paying enterprise support for a variety of products in the stack, but unwilling to commit to models where the entire stack is licensed per node, core, or application instance

We must understand our engineering maturity, appetite for building and empowering teams, and available resources to qualify whether building an application platform is a sensible undertaking.

Summarizing Application Platforms

Admittedly, what constitutes an application platform remains fairly gray. We've focused on a variety of platforms that we believe bring an experience to teams far beyond just workload orchestration. We have also articulated that Kubernetes can be customized and extended to achieve similar outcomes. By advancing our thinking beyond "How do I get a Kubernetes" into concerns such as "What is the current developer workflow, pain points, and desires?" platform and infrastructure teams will be more successful with what they build. With a focus on the latter, we'd argue, you are far more likely to chart a proper path to production and achieve nontrivial adoption. At the end of the day, we want to meet infrastructure, security, and developer requirements to ensure our customers—typically developers—are provided a solution that meets their needs. Often we do not want to simply provide a "powerful" engine that every developer must build their own platform atop of, as jokingly depicted in Figure 1-5.

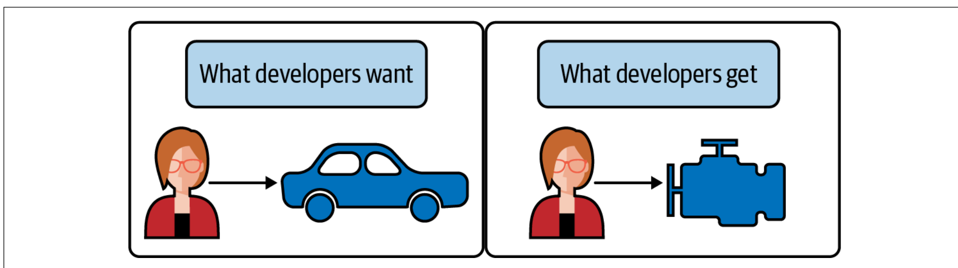


Figure 1-5. When developers desire an end-to-end experience (e.g., a driveable car), do not expect an engine without a frame, wheels, and more to suffice.

Building Application Platforms on Kubernetes

Now we've identified Kubernetes as one piece of the puzzle in our path to production. With this, it would be reasonable to wonder "Isn't Kubernetes just missing stuff then?" The Unix philosophy's principle of "make each program do one thing well" is a compelling aspiration for the Kubernetes project. We believe its best features are largely the ones it does not have! Especially after being burned with one-size-fits-all platforms that try to solve the world's problems for you. Kubernetes has brilliantly focused on being a great orchestrator while defining clear interfaces for how it can be built on top of. This can be likened to the foundation of a home.

A good foundation should be structurally sound, able to be built on top of, and provide appropriate interfaces for routing utilities to the home. While important, a foundation alone is rarely a habitable place for our applications to live. Typically, we need some form of home to exist on top of the foundation. Before discussing *building* on top of a foundation such as Kubernetes, let's consider a pre-furnished apartment as shown in Figure 1-6.

Figure 1-6. An apartment that is move-in ready. Similar to platform as a service options like Heroku. Illustration by Jessica Appelbaum.

This option, similar to our examples such as Heroku, is habitable with no additional work. There are certainly opportunities to customize the experience inside; however, many concerns are solved for us. As long as we are comfortable with the price of rent and are willing to conform to the nonnegotiable opinions within, we can be successful on day one.

Circling back to Kubernetes, which we have likened to a foundation, we can now look to build that habitable home on top of it, as depicted in Figure 1-7.

Figure 1-7. Building a house. Similar to establishing an application platform, which Kubernetes is foundational to. Illustration by Jessica Appelbaum.

At the cost of planning, engineering, and maintaining, we can build remarkable platforms to run workloads throughout organizations. This means we're in complete control of every element in the output. The house can and should be tailored to the needs of the future tenants (our applications). Let's now break down the various layers and considerations that make this possible.

Starting from the Bottom

First we must start at the bottom, which includes the technology Kubernetes expects to run. This is commonly a datacenter or cloud provider, which offers compute, storage, and networking. Once established, Kubernetes can be bootstrapped on top. Within minutes you can have clusters living atop the underlying infrastructure. There are several means of bootstrapping Kubernetes, and we'll cover them in depth in Chapter 2.

From the point of Kubernetes clusters existing, we next need to look at a conceptual flow to determine what we should build on top. The key junctures are represented in Figure 1-8.

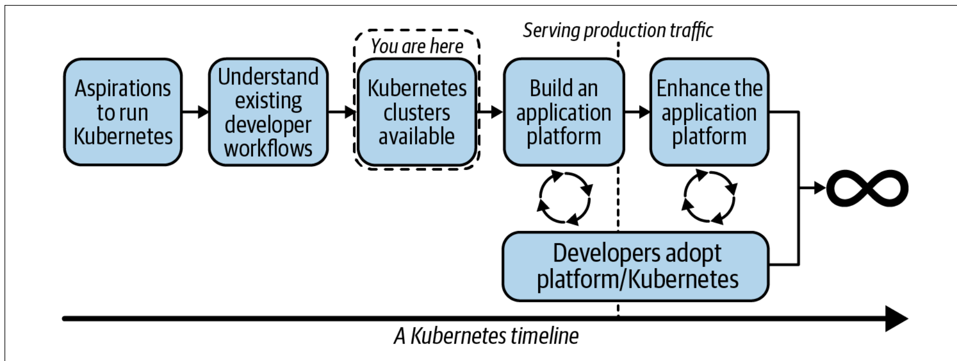


Figure 1-8. A flow our teams may go through in their path to production with Kubernetes.

From the point of Kubernetes existing, you can expect to quickly be receiving questions such as:

- "How do I ensure workload-to-workload traffic is fully encrypted?"
- "How do I ensure egress traffic goes through a gateway guaranteeing a consistent source CIDR?"
- "How do I provide self-service tracing and dashboards to applications?"
- "How do I let developers onboard without being concerned about them becoming Kubernetes experts?"

This list can be endless. It is often incumbent on us to determine which requirements to solve at a platform level and which to solve at an application level. The key here is to deeply understand exiting workflows to ensure what we build lines up with current expectations. If we cannot meet that feature set, what impact will it have on the development teams? Next we can start the building of a platform on top of Kubernetes. In doing so, it is key we stay paired with development teams willing to onboard early and understand the experience to make informed decisions based on quick feedback. After reaching production, this flow should not stop. Platform teams should not expect what is delivered to be a static environment that developers will use for decades. In order to be successful, we must constantly be in tune with our development groups to understand where there are issues or potential missing features that could increase development velocity. A good place to start is considering what level of interaction with Kubernetes we should expect from our developers. This is the idea of how much, or how little, we should abstract.

The Abstraction Spectrum

In the past, we've heard posturing like, "If your application developers know they're using Kubernetes, you've failed!" This can be a decent way to look at interaction with Kubernetes, especially if you're building products or services where the underlying orchestration technology is meaningless to the end user. Perhaps you're building a database management system (DBMS) that supports multiple database technologies. Whether shards or instances of a database run via Kubernetes, Bosh, or Mesos probably doesn't matter to your developers! However, taking this philosophy wholesale from a tweet into your team's success criteria is a dangerous thing to do. As we layer pieces on top of Kubernetes and build platform services to better serve our customers, we'll be faced with many points of decision to determine what appropriate abstractions looks like. Figure 1-9 provides a visualization of this spectrum.

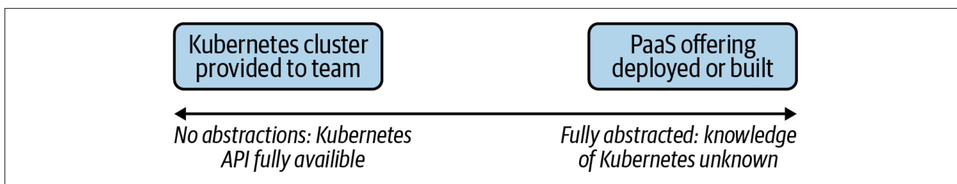


Figure 1-9. The various ends of the spectrum. Starting with giving each team its own Kubernetes cluster to entirely abstracting Kubernetes from your users, via a platform as a service (PaaS) offering.

This can be a question that keeps platform teams up at night. There's a lot of merit in providing abstractions. Projects like Cloud Foundry provide a fully baked developer experience—an example being that in the context of a single `cf push` we can take an application, build it, deploy it, and have it serving production traffic. With this goal and experience as a primary focus, as Cloud Foundry furthers its support for running on top of Kubernetes, we expect to see this transition as more of an implementation detail than a change in feature set. Another pattern we see is the desire to offer more than Kubernetes at a company, but not make developers explicitly choose between technologies. For example, some companies have a Mesos footprint alongside a Kubernetes footprint. They then build an abstraction enabling transparent selection of where workloads land without putting that onus on application developers. It also prevents them from technology lock-in. A trade-off to this approach includes building abstractions on top of two systems that operate differently. This requires significant engineering effort and maturity. Additionally, while developers are eased of the burden around knowing how to interact with Kubernetes or Mesos, they instead need to understand how to use an abstracted company-specific system. In the modern era of open source, developers from all over the stack are less enthused about learning systems that don't translate between organizations. Lastly, a pitfall we've seen is an obsession with abstraction causing an inability to expose key features of Kubernetes.

Over time this can become a cat-and-mouse game of trying to keep up with the project and potentially making your abstraction as complicated as the system it's abstracting.

On the other end of the spectrum are platform groups that wish to offer self-service clusters to development teams. This can also be a great model. It does put the responsibility of Kubernetes maturity on the development teams. Do they understand how Deployments, ReplicaSets, Pods, Services, and Ingress APIs work? Do they have a sense for setting millicpus and how overcommit of resources works? Do they know how to ensure that workloads configured with more than one replica are always scheduled on different nodes? If yes, this is a perfect opportunity to avoid over-engineering an application platform and instead let application teams take it from the Kubernetes layer up.

This model of development teams owning their own clusters is a little less common. Even with a team of humans that have a Kubernetes background, it's unlikely that they want to take time away from shipping features to determine how to manage the life cycle of their Kubernetes cluster when it comes time to upgrade. There's so much power in all the knobs Kubernetes exposes, but for many development teams, expecting them to become Kubernetes experts on top of shipping software is unrealistic. As you'll find in the coming chapters, abstraction does not have to be a binary decision. At a variety of points we'll be able to make informed decisions on where abstractions make sense. We'll be determining where we can provide developers the right amount of flexibility while still streamlining their ability to get things done.

Determining Platform Services

When building on top of Kubernetes, a key determination is what features should be built into the platform relative to solved at the application level. Generally this is something that should be evaluated at a case-by-case basis. For example, let's assume every Java microservice implements a library that facilitates mutual TLS (mTLS) between services. This provides applications a construct for identity of workloads and encryption of data over the network. As a platform team, we need to deeply understand this usage to determine whether it is something we should offer or implement at a platform level. Many teams look to solve this by potentially implementing a technology called a service mesh into the cluster. An exercise in trade-offs would reveal the following considerations.

Pros to introducing a service mesh:

- Java apps no longer need to bundle libraries to facilitate mTLS.
- Non-Java applications can take part in the same mTLS/encryption system.
- Lessened complexity for application teams to solve for.

Cons to introducing a service mesh:

- Running a service mesh is not a trivial task. It is another distributed system with operational complexity.
- Service meshes often introduce features far beyond identity and encryption.
- The mesh's identity API might not integrate with the same backend system as used by the existing applications.

Weighing these pros and cons, we can come to the conclusion as to whether solving this problem at a platform level is worth the effort. The key is we don't need to, and should not strive to, solve every application concern in our new platform. This is another balancing act to consider as you proceed through the many chapters in this book. Several recommendations, best practices, and guidance will be shared, but like anything, you should assess each based on the priorities of your business needs.

The Building Blocks

Let's wrap up this chapter by concretely identifying key building blocks you will have available as you build a platform. This includes everything from the foundational components to optional platform services you may wish to implement.

The components in Figure 1-10 have differing importance to differing audiences.

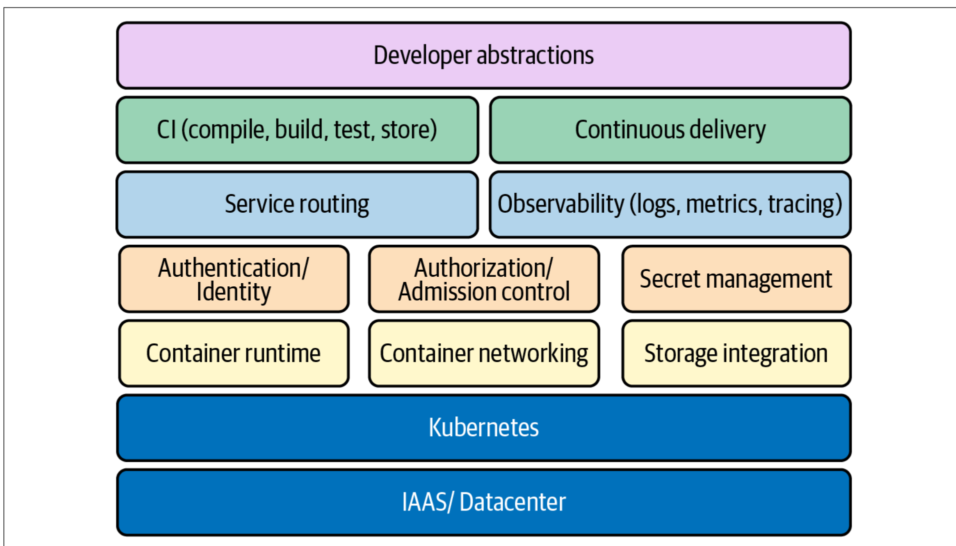


Figure 1-10. Many of the key building blocks involved in establishing an application platform.

Some components such as container networking and container runtime are required for every cluster, considering that a Kubernetes cluster that can't run workloads or allow them to communicate would not be very successful. You are likely to find some components to have variance in whether they should be implemented at all. For example, secret management might not be a platform service you intend to implement if applications already get their secrets from an external secret management solution.

Some areas, such as security, are clearly missing from Figure 1-10. This is because security is not a feature but more so a result of how you implement everything from the IAAS layer up. Let's explore these key areas at a high level, with the understanding that we'll dive much deeper into them throughout this book.

IAAS/datacenter and Kubernetes

IAAS/datacenter and Kubernetes form the foundational layer we have called out many times in this chapter. We don't mean to trivialize this layer because its stability will directly correlate to that of our platform. However, in modern environments, we spend much less time determining the architecture of our racks to support Kubernetes and a lot more time deciding between a variety of deployment options and topologies. Essentially we need to assess how we are going to provision and make available Kubernetes clusters.

Container runtime

The container runtime will faciliate the life cycle management of our workloads on each host. This is commonly implemented using a technology that can manage containers, such as CRI-O, containerd, and Docker. The ability to choose between these different implementations is thanks to the Container Runtime Interface (CRI). Along with these common examples, there are specialized runtimes that support unique requirements, such as the desire to run a workload in a micro-vm.

Container networking

Our choice of container networking will commonly address IP address management (IPAM) of workloads and routing protocols to facilitate communication. Common technology choices include Calico or Cilium, which is thanks to the Container Networking Interface (CNI). By plugging a container networking technology into the cluster, the kubelet can request IP addresses for the workloads it starts. Some plug-ins go as far as implementing service abstractions on top of the Pod network.

Storage integration

Storage integration covers what we do when the on-host disk storage just won't cut it. In modern Kubernetes, more and more organizations are shipping stateful workloads to their clusters. These workloads require some degree of certainty that the state will be resilient to application failure or rescheduling events. Storage can be supplied by common systems such as vSAN, EBS, Ceph, and many more. The ability to choose between various backends is facilitated by the Container Storage Interface (CSI). Similar to CNI and CRI, we are able to deploy a plug-in to our cluster that understands how to satisfy the storage needs requested by the application.

Service routing

Service routing is the facilitation of traffic to and from the workloads we run in Kubernetes. Kubernetes offers a Service API, but this is typically a stepping stone for support of more feature-rich routing capabilities. Service routing builds on container networking and creates higher-level features such as layer 7 routing, traffic patterns, and much more. Many times these are implemented using a technology called an Ingress controller. At the deeper side of service routing comes a variety of service meshes. This technology is fully featured with mechanisms such as service-to-service mTLS, observability, and support for applications mechanisms such as circuit breaking.

Secret management

Secret management covers the management and distribution of sensitive data needed by workloads. Kubernetes offers a Secrets API where sensitive data can be interacted with. However, out of the box, many clusters don't have robust enough secret management and encryption capabilities demanded by several enterprises. This is largely a conversation around defense in depth. At a simple level, we can ensure data is encrypted before it is stored (encryption at rest). At a more advanced level, we can provide integration with various technologies focused on secret management, such as Vault or Cyberark.

Identity

Identity covers the authentication of humans and workloads. A common initial ask of cluster administrators is how to authenticate users against a system such as LDAP or a cloud provider's IAM system. Beyond humans, workloads may wish to identify themselves to support zero-trust networking models where impersonation of workloads is far more challenging. This can be facilitated by integrating an identity provider and using mechanisms such as mTLS to verify a workload.

Authorization/admission control

Authorization is the next step after we can verify the identity of a human or workload. When users or workloads interact with the API server, how do we grant or deny their access to resources? Kubernetes offers an RBAC feature with resource/verb-level controls, but what about custom logic specific to authorization inside our organization? Admission control is where we can take this a step further by building out validation logic that can be as simple as looking over a static list of rules to dynamically calling other systems to determine the correct authorization response.

Software supply chain

The software supply chain covers the entire life cycle of getting software in source code to runtime. This involves the common concerns around continuous integration (CI) and continuous delivery (CD). Many times, developers' primary interaction point is the pipelines they establish in these systems. Getting the CI/CD systems working well with Kubernetes can be paramount to your platform's success. Beyond CI/CD are concerns around the storage of artifacts, their safety from a vulnerability standpoint, and ensuring integrity of images that will be run in your cluster.

Observability

Observability is the umbrella term for all things that help us understand what's happening with our clusters. This includes at the system and application layers. Typically, we think of observability to cover three key areas. These are logs, metrics, and tracing. Logging typically involves forwarding log data from workloads on the host to a target backend system. From this system we can aggregate and analyze logs in a consumable way. Metrics involves capturing data that represents some state at a point in time. We often aggregate, or scrape, this data into some system for analysis. Tracing has largely grown in popularity out of the need to understand the interactions between the various services that make up our application stack. As trace data is collected, it can be brought up to an aggregate system where the life of a request or response is shown via some form of context or correlation ID.

Developer abstractions

Developer abstractions are the tools and platform services we put in place to make developers successful in our platform. As discussed earlier, abstraction approaches live on a spectrum. Some organizations will choose to make the usage of Kubernetes completely transparent to the development teams. Other shops will choose to expose many of the powerful knobs Kubernetes offers and give significant flexibility to every developer. Solutions also tend to focus on the developer onboarding experience, ensuring they can be given access and secure control of an environment they can utilize in the platform.

Summary

In this chapter, we have explored ideas spanning Kubernetes, application platforms, and even building application platforms on Kubernetes. Hopefully this has gotten you thinking about the variety of areas you can jump into in order to better understand how to build on top of this great workload orchestrator. For the remainder of the book we are going to dive into these key areas and provide insight, anecdotes, and recommendations that will further build your perspective on platform building. Let's jump in and start down this path to production!

CHAPTER 2

Deployment Models

The first step to using Kubernetes in production is obvious: make Kubernetes exist. This includes installing systems to provision Kubernetes clusters and to manage future upgrades. Being that Kubernetes is a distributed software system, deploying Kubernetes largely boils down to a software installation exercise. The important difference compared with most other software installs is that Kubernetes is intrinsically tied to the infrastructure. As such, the software installation and the infrastructure it's being installed on need to be simultaneously solved for.

In this chapter we will first address preliminary questions around deploying Kubernetes clusters and how much you should leverage managed services and existing products or projects. For those that heavily leverage existing services, products, and projects, most of this chapter may not be of interest because about 90% of the content in this chapter covers how to approach custom automation. This chapter can still be of interest if you are evaluating tools for deploying Kubernetes so that you can reason about the different approaches available. For those in the uncommon position of having to build custom automation for deploying Kubernetes, we will address overarching architectural concerns, including special considerations for etcd as well as how to manage the various clusters under management. We will also look at useful patterns for managing the various software installations as well as the infrastructure dependencies and will break down the various cluster components and demystify how they fit together. We'll also look at ways to manage the add-ons you install to the base Kubernetes cluster as well as strategies for upgrading Kubernetes and the add-on components that make up your application platform.

Managed Service Versus Roll Your Own

Before we get further into the topic of deployment models for Kubernetes, we should address the idea of whether you should even *have* a full deployment model for Kubernetes. Cloud providers offer managed Kubernetes services that mostly alleviate the deployment concerns. You should still develop reliable, declarative systems for provisioning these managed Kubernetes clusters, but it may be advantageous to abstract away most of the details of *how* the cluster is brought up.

Managed Services

The case for using managed Kubernetes services boils down to savings in engineering effort. There is considerable technical design and implementation in properly managing the deployment and life cycle of Kubernetes. And remember, Kubernetes is just one component of your application platform—the container orchestrator.

In essence, with a managed service you get a Kubernetes control plane that you can attach worker nodes to at will. The obligation to scale, ensure availability, and manage the control plane is alleviated. These are each significant concerns. Furthermore, if you already use a cloud provider's existing services you get a leg up. For example, if you are in Amazon Web Services (AWS) and already use Fargate for serverless compute, Identity and Access Management (IAM) for role-based access control, and CloudWatch for observability, you can leverage these with their Elastic Kubernetes Service (EKS) and solve for several concerns in your app platform.

It is not unlike using a managed database service. If your core concern is an application that serves your business needs, and that app requires a relational database, but you cannot justify having a dedicated database admin on staff, paying a cloud provider to supply you with a database can be a huge boost. You can get up and running faster. The managed service provider will manage availability, take backups, and perform upgrades on your behalf. In many cases this is a clear benefit. But, as always, there is a trade-off.

Roll Your Own

The savings available in using a managed Kubernetes service come with a price tag. You pay with a lack of flexibility and freedom. Part of this is the threat of vendor lock-in. The managed services are generally offered by cloud infrastructure providers. If you invest heavily in using a particular vendor for your infrastructure, it is highly likely that you will design systems and leverage services that will not be vendor neutral. The concern is that if they raise their prices or let their service quality slip in the future, you may find yourself painted into a corner. Those experts you paid to handle concerns you didn't have time for may now wield dangerous power over your destiny.

Of course, you can diversify by using managed services from multiple providers, but there will be deltas between the way they expose features of Kubernetes, and which features are exposed could become an awkward inconsistency to overcome.

For this reason, you may prefer to roll your own Kubernetes. There is a vast array of knobs and levers to adjust on Kubernetes. This configurability makes it wonderfully flexible and powerful. If you invest in understanding and managing Kubernetes itself, the app platform world is your oyster. There will be no feature you cannot implement, no requirement you cannot meet. And you will be able to implement that seamlessly across infrastructure providers, whether they be public cloud providers, or your own servers in a private datacenter. Once the different infrastructure inconsistencies are accounted for, the Kubernetes features that are exposed in your platform will be consistent. And the developers that use your platform will not care—and may not even know—who is providing the underlying infrastructure.

Just keep in mind that developers will care only about the features of the platform, not the underlying infrastructure or who provides it. If you are in control of the features available, and the features you deliver are consistent across infrastructure providers, you have the freedom to deliver a superior experience to your devs. You will have control of the Kubernetes version you use. You will have access to all the flags and features of the control plane components. You will have access to the underlying machines and the software that is installed on them as well as the static Pod manifests that are written to disk there. You will have a powerful and dangerous tool to use in the effort to win over your developers. But never ignore the obligation you have to learn the tool well. A failure to do so risks injuring yourself and others with it.

Making the Decision

The path to glory is rarely clear when you begin the journey. If you are deciding between a managed Kubernetes service or rolling your own clusters, you are much closer to the beginning of your journey with Kubernetes than the glorious final conclusion. And the decision of managed service versus roll your own is fundamental enough that it will have long-lasting implications for your business. So here are some guiding principles to aid the process.

You should lean toward a managed service if:

- The idea of understanding Kubernetes sounds terribly arduous
- The responsibility for managing a distributed software system that is critical to the success of your business sounds dangerous
- The inconveniences of restrictions imposed by vendor-provided features seem manageable

- You have faith in your managed service vendor to respond to your needs and be a good business partner

You should lean toward rolling your own Kubernetes if:

- The vendor-imposed restrictions make you uneasy
- You have little or no faith in the corporate behemoths that provide cloud compute infrastructure
- You are excited by the power of the platform you can build around Kubernetes
- You relish the opportunity to leverage this amazing container orchestrator to provide a delightful experience to your devs

If you decide to use a managed service, consider skipping most of the remainder of this chapter. "Add-ons" on page 50 and "Triggering Mechanisms" on page 60 are still applicable to your use case, but the other sections in this chapter will not apply. If, on the other hand, you are looking to manage your own clusters, read on! Next we'll dig more into the deployment models and tools you should consider.

Automation

If you are to undertake designing a deployment model for your Kubernetes clusters, the topic of automation is of the utmost importance. Any deployment model will need to keep this as a guiding principle. Removing human toil is critical to reduce cost and improve stability. Humans are costly. Paying the salary for engineers to execute routine, tedious operations is money not spent on innovation. Furthermore, humans are unreliable. They make mistakes. Just one error in a series of steps may introduce instability or even prevent the system from working at all. The upfront engineering investment to automate deployments using software systems will pay dividends in saved toil and troubleshooting in the future.

If you decide to manage your own cluster life cycle, you must formulate your strategy for doing this. You have a choice between using a prebuilt Kubernetes installer or developing your own custom automation from the ground up. This decision has parallels with the decision between managed services versus roll your own. One path gives you great power, control, and flexibility but at the cost of engineering effort.

Prebuilt Installer

There are now countless open source and enterprise-supported Kubernetes installers available. Case in point: there are currently 180 Kubernetes Certified Service Providers listed on the CNCF's website. Some you will need to pay money for and will be accompanied by experienced field engineers to help get you up and running as well as support staff you can call on in times of need. Others will require research and

experimentation to understand and use. Some installers—usually the ones you pay money for—will get you from zero to Kubernetes with the push of a button. If you fit the prescriptions provided and options available, and your budget can accommodate the expense, this installer method could be a great fit. At the time of this writing, using prebuilt installers is the approach we see most commonly in the field.

Custom Automation

Some amount of custom automation is commonly required even if using a prebuilt installer. This is usually in the form of integration with a team's existing systems. However, in this section we're talking about developing a custom Kubernetes installer.

If you are beginning your journey with Kubernetes or changing direction with your Kubernetes strategy, the homegrown automation route is likely your choice only if all of the following apply:

- You have more than just one or two engineers to devote to the effort
- You have engineers on staff with deep Kubernetes experience
- You have specialized requirements that no managed service or prebuilt installer satisfies well

Most of the remainder of this chapter is for you if one of the following applies:

- You fit the use case for building custom automation
- You are evaluating installers and want to gain a deeper insight into what good patterns look like

This brings us to details of building custom automation to install and manage Kubernetes clusters. Underpinning all these concerns should be a clear understanding of your platform requirements. These should be driven primarily by the requirements of your app development teams, particularly those that will be the earliest adopters. Do not fall into the trap of building a platform in a vacuum without close collaboration with the consumers of the platform. Make early prerelease versions of your platform available to dev teams for testing. Cultivate a productive feedback loop for fixing bugs and adding features. The successful adoption of your platform depends upon this.

Next we will cover architecture concerns that should be considered before any implementation begins. This includes deployment models for etcd, separating deployment environments into tiers, tackling challenges with managing large numbers of clusters, and what types of node pools you might use to host your workloads. After that, we'll get into Kubernetes installation details, first for the infrastructure dependencies, then, the software that is installed on the clusters' virtual or physical machines, and finally for the containerized components that constitute the control plane of a Kubernetes cluster.

Architecture and Topology

This section covers the architectural decisions that have broad implications for the systems you use to provision and manage your Kubernetes clusters. They include the deployment model for etcd and the unique considerations you must take into account for that component of the platform. Among these topics is how you organize the various clusters under management into tiers based on the service level objectives (SLOs) for them. We will also look at the concept of node pools and how they can be used for different purposes within a given cluster. And, lastly, we will address the methods you can use for federated management of your clusters and the software you deploy to them.

etcd Deployment Models

As the database for the objects in a Kubernetes cluster, etcd deserves special consideration. etcd is a distributed data store that uses a consensus algorithm to maintain a copy of the your cluster's state on multiple machines. This introduces network considerations for the nodes in an etcd cluster so they can reliably maintain that consensus over their network connections. It has unique network latency requirements that we need to design for when considering network topology. We'll cover that topic in this section and also look at the two primary architectural choices to make in the deployment model for etcd: dedicated versus colocated and whether to run in a container or install directly on the host.

Network considerations

The default settings in etcd are designed for the latency in a single datacenter. If you distribute etcd across multiple datacenters, you should test the average round-trip between members and tune the heartbeat interval and election timeout for etcd if need be. We strongly discourage the use of etcd clusters distributed across different regions. If using multiple datacenters for improved availability, they should at least be in close proximity within a region.

Dedicated versus colocated

A very common question we get about how to deploy is whether to give etcd its own dedicated machines or to colocate them on the control plane machines with the API server, scheduler, controller manager, etc. The first thing to consider is the size of clusters you will be managing, i.e., the number of worker nodes you will run per cluster. The trade-offs around cluster sizes will be discussed later in the chapter. Where you land on that subject will largely inform whether you dedicate machines to etcd. Obviously etcd is crucial. If etcd performance is compromised, your ability to control the resources in your cluster will be compromised. As long as your workloads don't

have dependencies on the Kubernetes API, they should not suffer, but keeping your control plane healthy is still very important.

If you are driving a car down the street and the steering wheel stops working, it is little comfort that the car is still driving down the road. In fact, it may be terribly dangerous. For this reason, if you are going to be placing the read and write demands on etcd that come with larger clusters, it is wise to dedicate machines to them to eliminate resource contention with other control plane components. In this context, a "large" cluster is dependent upon the size of the control plane machines in use but should be at least a topic of consideration with anything above 50 worker nodes. If planning for clusters with over 200 workers, it's best to just plan for dedicated etcd clusters. If you do plan smaller clusters, save yourself the management overhead and infrastructure costs—go with colocated etcd. Kubeadm is a popular Kubernetes bootstrapping tool that you will likely be using; it supports this model and will take care of the associated concerns.

Containerized versus on host

The next common question revolves around whether to install etcd on the machine or to run it in a container. Let's tackle the easy answer first: if you're running etcd in a colocated manner, run it in a container. When leveraging kubeadm for Kubernetes bootstrapping, this configuration is supported and well tested. It is your best option. If, on the other hand, you opt for running etcd on dedicated machines your options are as follows: you can install etcd on the host, which gives you the opportunity to bake it into machine images and eliminate the additional concerns of having a container runtime on the host. Alternatively, if you run in a container, the most useful pattern is to install a container runtime and kubelet on the machines and use a static manifest to spin up etcd. This has the advantage of following the same patterns and install methods as the other control plane components. Using repeated patterns in complex systems is useful, but this question is largely a question of preference.

Cluster Tiers

Organizing your clusters according to tiers is an almost universal pattern we see in the field. These tiers often include testing, development, staging, and production. Some teams refer to these as different "environments." However, this is a broad term that can have different meanings and implications. We will use the term *tier* here to specifically address the different types of clusters. In particular, we're talking about the SLOs and SLAs that may be associated with the cluster, as well as the purpose for the cluster, and where the cluster sits in the path to production for an application, if at all. What exactly these tiers will look like for different organizations varies, but there are common themes and we will describe what each of these four tiers commonly mean. Use the same cluster deployment and life cycle management systems across all

tiers. The heavy use of these systems in the lower tiers will help ensure they will work as expected when applied to critical production clusters:

Testing
: Clusters in the testing tier are single-tenant, ephemeral clusters that often have a time-to-live (TTL) applied such that they are automatically destroyed after a specified amount of time, usually less than a week. These are spun up very commonly by platform engineers for the purpose of testing particular components or platform features they are developing. Testing tiers may also be used by developers when a local cluster is inadequate for local development, or as a subsequent step to testing on a local cluster. This is more common when an app dev team is initially containerizing and testing their application on Kubernetes. There is no SLO or SLA for these clusters. These clusters would use the latest version of a platform, or perhaps optionally a pre-alpha release.

Development
: Development tier clusters are generally "permanent" clusters without a TTL. They are multitenant (where applicable) and have all the features of a production cluster. They are used for the first round of integration tests for applications and are used to test the compatibility of application workloads with alpha versions of the platform. Development tiers are also used for general testing and development for the app dev teams. These clusters normally have an SLO but not a formal agreement associated with them. The availability objectives will often be near production-level, at least during business hours, because outages will impact developer productivity. In contrast, the applications have zero SLO or SLA when running on dev clusters and are very frequently updated and in constant flux. These clusters will run the officially released alpha and/or beta version of the platform.

Staging
: Like those in the development tier, clusters in the staging tier are also permanent clusters and are commonly used by multiple tenants. They are used for final integration testing and approval before rolling out to live production. They are used by stakeholders that are not actively developing the software running there. This would include project managers, product owners, and executives. This may also include customers or external stakeholders who need access to prerelease versions of software. They will often have a similar SLO to development clusters. Staging tiers may have a formal SLA associated with them if external stakeholders or paying customers are accessing workloads on the cluster. These clusters will run the officially released beta version of the platform if strict backward compatibility is followed by the platform team. If backward compatibility cannot be guaranteed, the staging cluster should run the same stable release of the platform as used in production.

Production

Production tier clusters are the money-makers. These are used for customer-facing, revenue-producing applications and websites. Only approved, production-ready, stable releases of software are run here. And only the fully tested and approved stable release of the platform is used. Detailed well-defined SLOs are used and tracked. Often, legally binding SLAs apply.

Node Pools

Node pools are a way to group together types of nodes within a single Kubernetes cluster. These types of nodes may be grouped together by way of their unique characteristics or by way of the role they play. It's important to understand the trade-offs of using node pools before we get into details. The trade-off often revolves around the choice between using multiple node pools within a single cluster versus provisioning separate, distinct clusters. If you use node pools, you will need to use Node selectors on your workloads to make sure they end up in the appropriate node pool. You will also likely need to use Node taints to prevent workloads without Node selectors from inadvertently landing where they shouldn't. Additionally, the scaling of nodes within your cluster becomes more complicated because your systems have to monitor distinct pools and scale each separately. If, on the other hand, you use distinct clusters you displace these concerns into cluster management and software federation concerns. You will need more clusters. And you will need to properly target your workloads to the right clusters. Table 2-1 summarizes these pros and cons of using node pools.

Table 2-1. Node pool pros and cons

Pros	Cons
Reduced number of clusters under management	Node selectors for workloads will often be needed
Smaller number of target clusters for workloads	Node taints will need to be applied and managed
	More complicated cluster scaling operations

A characteristic-based node pool is one that consists of nodes that have components or attributes that are required by, or suited to, some particular workloads. An example of this is the presence of a specialized device like a graphics processing unit (GPU). Another example of a characteristic may be the type of network interface it uses. One more could be the ratio of memory to CPU on the machine. We will discuss the reasons you may use nodes with different ratios of these resources in more depth later on in "Infrastructure" on page 35. Suffice to say for now, all these characteristics lend themselves to different types of workloads, and if you run them collectively in the same cluster, you'll need to group them into pools to manage where different Pods land.

A role-based node pool is one that has a particular function and that you often want to insulate from resource contention. The nodes sliced out into a role-based pool don't necessarily have peculiar characteristics, just a different function. A common example is to dedicate a node pool to the ingress layer in your cluster. In the example of an ingress pool, the dedicated pool not only insulates the workloads from resource contention (particularly important in this case since resource requests and limits are not currently available for network usage) but also simplifies the networking model and the specific nodes that are exposed to traffic from sources outside the cluster. In contrast to the characteristic-based node pool, these roles are often not a concern you can displace into distinct clusters because the machines play an important role in the function of a particular cluster. That said, do ensure you are slicing off nodes into a pool for good reason. Don't create pools indiscriminately. Kubernetes clusters are complex enough. Don't complicate your life more than you need to.

Keep in mind that you will most likely need to solve the multicluster management problems that many distinct clusters bring, regardless of whether you use node pools. There are very few enterprises that use Kubernetes that don't accumulate a large number of distinct clusters. There are a large variety of reasons for this. So if you are tempted to introduce characteristic-based node pools, consider investing the engineering effort into developing and refining your multicluster management. Then you unlock the opportunity to seamlessly use distinct clusters for the different machine characteristics you need to provide.

Cluster Federation

Cluster federation broadly refers to how to centrally manage all the clusters under your control. Kubernetes is like a guilty pleasure. When you discover how much you enjoy it, you can't have just one. But, similarly, if you don't keep that habit under control, it can become messy. Federation strategies are ways for enterprises to manage their software dependencies so they don't spiral into costly, destructive addictions.

A common, useful approach is to federate regionally and then globally. This lessens the blast radius of, and reduces the computational load for, these federation clusters. When you first begin federation efforts, you may not have the global presence or volume of infrastructure to justify a multilevel federation approach, but keep it in mind as a design principle in case it becomes a future requirement.

Let's discuss some important related subjects in this area. In this section, we'll look at how management clusters can help with consolidating and centralizing regional services. We'll consider how we can consolidate the metrics for workloads in various clusters. And we'll discuss how this impacts the managing workloads that are deployed across different clusters in a centrally managed way.

Management clusters

Management clusters are what they sound like: Kubernetes clusters that manage other clusters. Organizations are finding that, as their usage expands and as the number of clusters under management increases, they need to leverage software systems for smooth operation. And, as you might expect, they often use Kubernetes-based platforms to run this software. Cluster API has become a popular project for accomplishing this. It is a set of Kubernetes operators that use custom resources such as Cluster and Machine resources to represent other Kubernetes clusters and their components. A common pattern used is to deploy the Cluster API components to a management cluster for deploying and managing the infrastructure for other workload clusters.

Using a management cluster in this manner does have flaws, however. It is usually prudent to strictly separate concerns between your production tier and other tiers. Therefore, organizations will often have a management cluster dedicated to production. This further increases the management cluster overhead. Another problem is with cluster autoscaling, which is a method of adding and removing worker nodes in response to the scaling of workloads. The Cluster Autoscaler typically runs in the cluster that it scales so as to watch for conditions that require scaling events. But the management cluster contains the controller that manages the provisioning and decommissioning of those worker nodes. This introduces an external dependency on the management cluster for any workload cluster that uses Cluster Autoscaler, as illustrated in Figure 2-1. What if the management cluster becomes unavailable at a busy time that your cluster needs to scale out to meet demand?

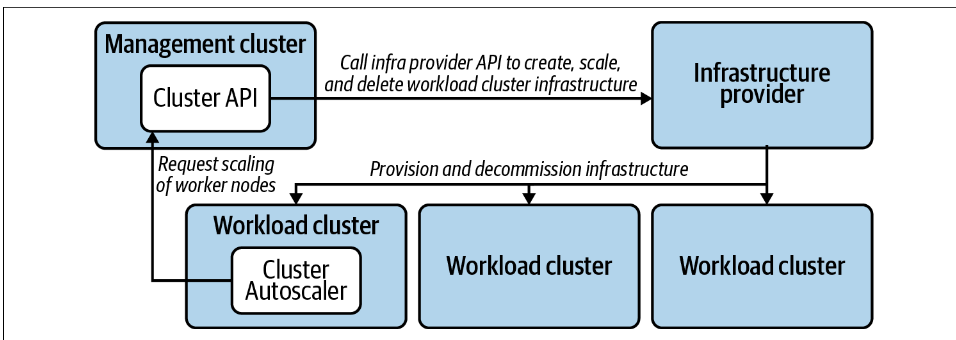


Figure 2-1. Cluster Autoscaler accessing a management cluster to trigger scaling events.

One strategy to remedy this is to run the Cluster API components in the workload cluster in a self-contained manner. In this case, the Cluster and Machine resources will also live there in the workload cluster. You can still use the management cluster for creation and deletion of clusters, but the workload cluster becomes largely autonomous and free from the external dependency on the management cluster for routine operations, such as autoscaling, as shown in Figure 2-2.

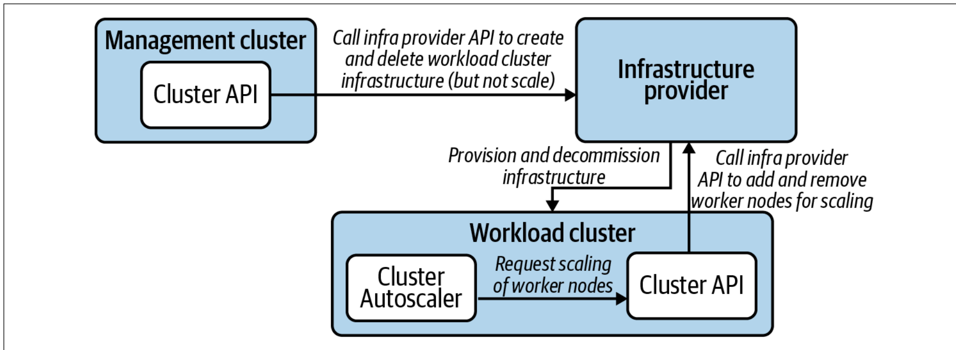


Figure 2-2. Cluster Autoscaler accessing a local Cluster API component to perform scaling events.

This pattern also has the distinct advantage that if any other controllers or workloads in the cluster have a need for metadata or attributes contained in the Cluster API custom resources, they can access them by reading the resource through the local API. There is no need to access the management cluster API. For example, if you have a Namespace controller that changes its behavior based on whether it is in a development or production cluster, that is information that can already be contained in the Cluster resource that represents the cluster in which it lives.

Additionally, management clusters also often host shared or regional services that are accessed by systems in various other clusters. These are not so much *management* functions. Management clusters are often just a logical place to run these shared services. Examples of these shared services include CI/CD systems and container registries.

Observability

When managing a large number of clusters, one of the challenges that arises is the collection of metrics from across this infrastructure and bringing them—or a subset thereof—into a central location. High-level measurable data points that give you a clear picture of the health of the clusters and workloads under management is a critical concern of cluster federation. Prometheus is a mature open source project that many organizations use to gather and store metrics. Whether you use it or not, the model it uses for federation is very useful and worth looking at so as to replicate with the tools you use, if possible. It supports the regional approach to federation by allowing federated Prometheus servers to scrape subsets of metrics from other, lower-level Prometheus servers. So it will accommodate any federation strategy you employ. Chapter 9 explores this topic in more depth.

Federated software deployment

Another important concern when managing various, remote clusters is how to manage deployment of software to those clusters. It's one thing to be able to manage the clusters themselves, but it's another entirely to organize the deployment of end-user workloads to these clusters. These are, after all, the point of having all these clusters. Perhaps you have critical, high-value workloads that must be deployed to multiple regions for availability purposes. Or maybe you just need to organize where workloads get deployed based on characteristics of different clusters. How you make these determinations is a challenging problem, as evidenced by the relative lack of consensus around a good solution to the problem.

The Kubernetes community has attempted to tackle this problem in a way that is broadly applicable for some time. The most recent incarnation is KubeFed. It also addresses cluster configurations, but here we're concerned more with the definitions of workloads that are destined for multiple clusters. One of the useful design concepts that has emerged is the ability to federate any API type that is used in Kubernetes. For example, you can use federated versions of Namespace and Deployment types and for declaring how resources should be applied to multiple clusters. This is a powerful notion in that you can centrally create a FederatedDeployment resource in one management cluster and have that manifest as multiple remote Deployment objects being created in other clusters. However, we expect to see more advances in this area in the future. At this time, the most common way we still see in the field to manage this concern is with CI/CD tools that are configured to target different clusters when workloads are deployed.

Now that we've covered the broad architectural concerns that will frame how your fleet of clusters is organized and managed, let's dig into the infrastructure concerns in detail.

Infrastructure

Kubernetes deployment is a software installation process with a dependency on IT infrastructure. A Kubernetes cluster can be spun up on one's laptop using virtual machines or Docker containers. But this is merely a simulation for testing purposes. For production use, various infrastructure components need to be present, and they are often provisioned as a part of the Kubernetes deployment itself.

A useful production-ready Kubernetes cluster needs some number of computers connected to a network to run on. To keep our terminology consistent, we'll use the term *machines* for these computers. Those machines may be virtual or physical. The important issue is that you are able to provision these machines, and a primary concern is the method used to bring them online.

You may have to purchase hardware and install them in a datacenter. Or you may be able to simply request the needed resources from a cloud provider to spin up machines as needed. Whatever the process, you need machines as well as properly configured networking, and this needs to be accounted for in your deployment model.

As an important part of your automation efforts, give careful consideration to the automation of infrastructure management. Lean away from manual operations such as clicking through forms in an online wizard. Lean toward using declarative systems that instead call an API to bring about the same result. This automation model requires the ability to provision servers, networking, and related resources on demand, as with a cloud provider such as Amazon Web Services, Microsoft Azure, or Google Cloud Platform. However, not all environments have an API or web user interface to spin up infrastructure. Vast production workloads run in datacenters filled with servers that are purchased and installed by the company that will use them. This needs to happen well before the Kubernetes software components are installed and run. It's important we draw this distinction and identify the models and patterns that apply usefully in each case.

The next section will address the challenges of running Kubernetes on bare metal in contrast to using virtual machines for the nodes in your Kubernetes clusters. We will then discuss cluster sizing trade-offs and the implications that has for your cluster life cycle management. Subsequently, we will go over the concerns you should take into account for the compute and networking infrastructure. And, finally, this will lead us to some specific strategies for automating the infrastructure management for your Kubernetes clusters.

Bare Metal Versus Virtualized

When exploring Kubernetes, many ponder whether the relevance of the virtual machine layer is necessary. Don't containers largely do the same thing? Would you essentially be running two layers of virtualization? The answer is, not necessarily. Kubernetes initiatives can be wildly successful atop bare metal or virtualized environments. Choosing the right medium to deploy to is critical and should be done through the lens of problems solved by various technologies and your team's maturity in these technologies.

The virtualization revolution changed how the world provisions and manages infrastructure. Historically, infrastructure teams used methodologies such as PXE booting hosts, managing server configurations, and making ancillary hardware, such as storage, available to servers. Modern virtualized environments abstract all of this behind APIs, where resources can be provisioned, mutated, and deleted at will without knowing what the underlying hardware looks like. This model has been proven throughout datacenters with vendors such as VMware and in the cloud where the

majority of compute is running atop some sort of hypervisor. Thanks to these advancements, many newcomers operating infrastructure in the cloud native world may never know about some of those underlying hardware concerns. The diagram in Figure 2-3 is not an exhaustive representation of the difference between virtualization and bare metal, but more so how the interaction points tend to differ.

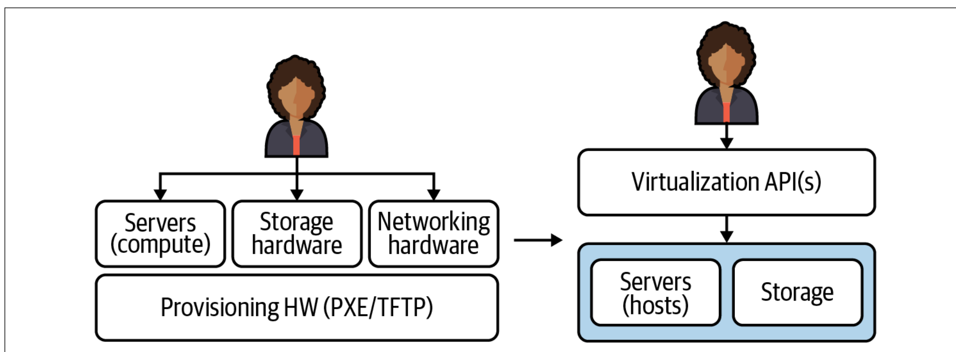


Figure 2-3. Comparison of administrator interactions when provisioning and managing bare metal compute infrastructure versus virtual machines.

The benefits of the virtualized models go far beyond having a unified API to interact with. In virtualized environments, we have the benefit of building many virtual servers within our hardware server, enabling us to slice each computer into fully isolated machines where we can:

- Easily create and clone machines and machine images
- Run many operating systems on the same server
- Optimize server usage by dedicating variant amounts of resources based on application needs
- Change resource settings without disrupting the server
- Programmatically control what hardware servers have access to, e.g., NICs
- Run unique networking and routing configurations per server

This flexibility also enables us to scope operational concerns on a smaller basis. For example, we can now upgrade one host without impacting all others running on the hardware. Additionally, with many of the mechanics available in virtualized environments, the creating and destroying of servers is typically more efficient. Virtualization has its own set of trade-offs. There is, typically, overhead incurred when running further away from the metal. Many hyper-performance–sensitive applications, such as trading applications, may prefer running on bare metal. There is also overhead in running the virtualization stack itself. In edge computing, for cases such as telcos running their 5G networks, they may desire running against the hardware.

Now that we've completed a brief review of the virtualization revolution, let's examine how this has impacted us when using Kubernetes and container abstractions because these force our point of interaction even higher up the stack. Figure 2-4 illustrates what this looks like through an operator's eyes at the Kubernetes layer. The underlying computers are viewed as a "sea of compute" where workloads can define what resources they need and will be scheduled appropriately.

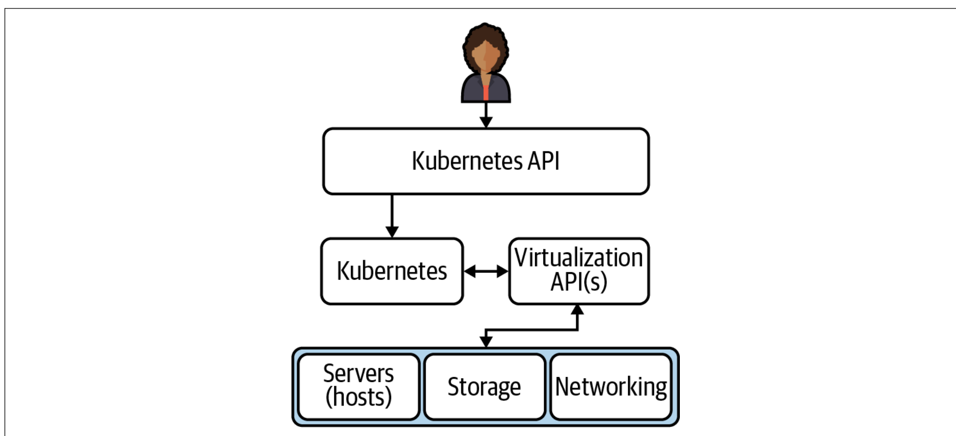


Figure 2-4. Operator interactions when using Kubernetes.

It's important to note that Kubernetes clusters have several integration points with the underlying infrastructure. For example, many use CSI-drivers to integrate with storage providers. There are multiple infra management projects that enable requesting new hosts from the provider and joining the cluster. And, most commonly, organizations rely on Cloud Provider Integrations (CPIs), which do additional work, such as provisioning load balancers outside of the cluster to route traffic within.

In essence, there are a lot of conveniences infrastructure teams lose when leaving virtualization behind—things that Kubernetes *does not* inherently solve. However, there are several projects and integration points with bare metal that make this space evermore promising. Bare metal options are becoming available through major cloud providers, and bare metal–exclusive IaaS services like Packet (recently acquired by Equinix Metal) are gaining market share. Success with bare metal is not without its challenges, including:

Significantly larger nodes
: Larger nodes cause (typically) more Pods per node. When thousands of Pods per node are needed to make good use of your hardware, operations can become more complicated. For example, when doing in-place upgrades, needing to drain a node to upgrade it means you may trigger 1000+ rescheduling events.

Dynamic scaling
: How to get new nodes up quickly based on workload or traffic needs.

Image provisioning
: Quickly baking and distributing machine images to keep cluster nodes as immutable as possible.

Lack of load balancer API
: Need to provide ingress routing from outside of the cluster to the Pod network within.

Less sophisticated storage integration
: Solving for getting network-attached storage to Pods.

Multitenant security concerns
: When hypervisors are in play, we have the luxury of ensuring security-sensitive containers run on dedicated hypervisors. Specifically we can slice up a physical server in any arbitrary way and make container scheduling decisions based on that.

These challenges are absolutely solvable. For example, the lack of load balancer integration can be solved with projects like kube-vip or metallb. Storage integration can be solved by integrating with a ceph cluster. However, the key is that containers aren't a new-age virtualization technology. Under the hood, containers are (in most implementations) using Linux kernel primitives to make processes feel isolated from others on a host. There's an endless number of trade-offs to continue unpacking, but in essence, our guidance when choosing between cloud providers (virtualization), on-prem virtualization, and bare metal is to consider what option makes the most sense based on your technical requirements and your organization's operational experience. If Kubernetes is being considered a replacement for a virtualization stack, reconsider exactly what Kubernetes solves for. Remember that learning to operate Kubernetes and enabling teams to operate Kubernetes is already an undertaking. Adding the complexity of completely changing how you manage your infrastructure underneath it significantly grows your engineering effort and risk.

Cluster Sizing

Integral to the design of your Kubernetes deployment model and the planning for infrastructure is the cluster sizes you plan to use. We're often asked, "How many worker nodes should be in production clusters?" This is a distinct question from, "How many worker nodes are needed to satisfy workloads?" If you plan to use one, single production cluster to rule them all, the answer to both questions will be the same. However, that is a unicorn we never see in the wild. Just as a Kubernetes cluster allows you to treat server machines as cattle, modern Kubernetes installation methods

and cloud providers allow you to treat the clusters themselves as cattle. And every enterprise that uses Kubernetes has at least a small herd.

Larger clusters offer the following benefits:

Better resource utilization
: Each cluster comes with a control plane overhead cost. This includes etcd and components such as the API server. Additionally, you'll run a variety of platform services in each cluster; for example, proxies via Ingress controllers. These components add overhead. A larger cluster minimizes replication of these services.

Fewer cluster deployments
: If you run your own bare metal compute infrastructure, as opposed to provisioning it on-demand from a cloud provider or on-prem virtualization platform, spinning clusters up and down as needed, scaling those clusters as demands dictate becomes less feasible. Your cluster deployment strategy can afford to be less automated if you execute that deployment strategy less often. It is entirely possible the engineering effort to fully automate cluster deployments would be greater than the engineering effort to manage a less automated strategy.

Simpler cluster and workload management profile
: If you have fewer production clusters, the systems you use to allocate, federate, and manage these concerns need not be as streamlined and sophisticated. Federated cluster and workload management across fleets of clusters is complex and challenging. The community has been working on this. Large teams at enormous enterprises have invested heavily in bespoke systems for this. And these efforts have enjoyed limited success thus far.

Smaller clusters offer the following benefits:

Smaller blast radius
: Cluster failures will impact fewer workloads.

Tenancy flexibility
: Kubernetes provides all the mechanisms needed to build a multitenant platform. However, in some cases you will spend far less engineering effort by provisioning a new cluster for a particular tenant. For example, if one tenant needs access to a cluster-wide resource like Custom Resource Definitions, and another tenant needs stringent guarantees of isolation for security and/or compliance, it may be justified to dedicate clusters to such teams, especially if their workloads demand significant compute resources.

Less tuning for scale
: As clusters scale into several hundred workers, we often encounter issues of scale that need to be solved for. These issues vary case to case, but bottlenecks in your control plane can occur. Engineering effort will need to be expended in

troubleshooting and tuning clusters. Smaller clusters considerably reduce this expenditure.

Upgrade options
Using smaller clusters lends itself more readily to replacing clusters in order to upgrade them. Cluster replacements certainly come with their own challenges, and these are covered later in this chapter in "Upgrades" on page 52, but this replacement strategy is an attractive upgrade option in many cases, and operating smaller clusters can make it even more attractive.

Node pool alternative
If you have workloads with specialized concerns such as GPUs or memory optimized nodes, and your systems readily accommodate lots of smaller clusters, it will be trivial to run dedicated clusters to accommodate these kinds of specialized concerns. This alleviates the complexity of managing multiple node pools as discussed earlier in this chapter.

Compute Infrastructure

To state the obvious, a Kubernetes cluster needs machines. Managing pools of these machines is the core purpose, after all. An early consideration is what types of machines you should choose. How many cores? How much memory? How much onboard storage? What grade of network interface? Do you need any specialized devices such as GPUs? These are all concerns that are driven by the demands of the software you plan to run. Are the workloads compute intensive? Or are they memory hungry? Are you running machine learning or AI workloads that necessitate GPUs? If your use case is very typical in that your workloads fit general-purpose machines' compute-to-memory ratio well, and if your workloads don't vary greatly in their resource consumption profile, this will be a relatively simple exercise. However, if you have less typical and more diverse software to run, this will be a little more involved. Let's consider the different types of machines to consider for your cluster:

etcd machines (optional)
This is an optional machine type that is only applicable if you run a dedicated etcd clusters for your Kubernetes clusters. We covered the trade-offs with this option in an earlier section. These machines should prioritize disk read/write performance, so never use old spinning disk hard drives. Also consider dedicating a storage disk to etcd, even if running etcd on dedicated machines, so that etcd suffers no contention with the OS or other programs for use of the disk. Also consider network performance, including proximity on the network, to reduce network latency between machines in a given etcd cluster.

Control plane nodes (required)
These machines will be dedicated to running control plane components for the cluster. They should be general-purpose machines that are sized and numbered according to the anticipated size of the cluster as well as failure tolerance requirements. In a larger cluster, the API server will have more clients and manage more traffic. This can be accommodated with more compute resources per machine, or more machines. However, components like the scheduler and controller-manager have only one active leader at any given time. Increasing capacity for these cannot be achieved with more replicas the way it can with the API server. Scaling vertically with more compute resources per machine must be used if these components become starved for resources. Additionally, if you are colocating etcd on these control plane machines, the same considerations for etcd machines noted above also apply.

Worker Nodes (required)
These are general-purpose machines that host non–control plane workloads.

Memory optimized Nodes (optional)
If you have workloads that have a memory profile that doesn't make them a good fit for general-purpose worker nodes, you should consider a node pool that is memory optimized. For example, if you are using AWS general-purpose M5 instance types for worker nodes that have a CPU:memory ratio of 1CPU:4GiB, but you have a workload that consumes resources at a ratio of 1CPU:8GiB, these workloads will leave unused CPU when resources are requested (reserved) in your cluster at this ratio. This inefficiency can be overcome by using memory-optimized nodes such as the R5 instance types on AWS, which have a ratio of 1CPU:8GiB.

Compute optimized Nodes (optional)
Alternatively, if you have workloads that fit the profile of a compute-optimized node such as the C5 instance type in AWS with 1CPU:2GiB, you should consider adding a node pool with these machine types for improved efficiency.

Specialized hardware Nodes (optional)
A common hardware ask is GPUs. If you have workloads (e.g., machine learning) requiring specialized hardware, adding a node pool in your cluster and then targeting those nodes for the appropriate workloads will work well.

Networking Infrastructure

Networking is easy to brush off as an implementation detail, but it can have important impacts on your deployment models. First, let's examine the elements that you will need to consider and design for.

Routability

You almost certainly do not want your cluster nodes exposed to the public internet. The convenience of being able to connect to those nodes from anywhere almost never justifies the threat exposure. You will need to solve for gaining access to those nodes should you need to connect to them, but a bastion host or jump box that is well secured and that will allow SSH access, and in turn allow you to connect to cluster nodes, is a low barrier to hop.

However, there are more nuanced questions to answer, such as network access on your private network. There will be services on your network that will need connectivity to and from your cluster. For example, it is common to need connectivity with storage arrays, internal container registries, CI/CD systems, internal DNS, private NTP servers, etc. Your cluster will also usually need access to public resources such as public container registries, even if via an outbound proxy.

If outbound public internet access is out of the question, those resources such as open source container images and system packages will need to be made available in some other way. Lean toward simpler systems that are consistent and effective. Lean away from, if possible, mindless mandates and human approval for infrastructure needed for cluster deployments.

Redundancy

Use availability zones (AZs) to help maintain uptime where possible. For clarity, an availability zone is a datacenter that has a distinct power source and backup as well as a distinct connection to the public internet. Two subnets in a datacenter with a shared power supply do not constitute two availability zones. However, two distinct datacenters that are in relatively close proximity to one another and have a low-latency, high-bandwidth network connection between them do constitute a pair of availability zones. Two AZs is good. Three is better. More than that depends of the level of catastrophe you need to prepare for. Datacenters have been known to go down. For multiple datacenters in a region to suffer simultaneous outages is possible, but rare and would often indicate a kind of disaster that will require you to consider how critical your workloads are. Are you running workloads necessary to national defense, or an online store? Also consider where you need redundancy. Are you building redundancy for your workloads? Or the control plane of the cluster itself? In our experience it is acceptable to run etcd across AZs but, if doing so, revisit "Network considerations" on page 28. Keep in mind that distributing your control plane across AZs gives redundant *control* over the cluster. Unless your workloads depend on the cluster control plane (which should be avoided), your workload availability will not be affected by a control plane outage. What will be affected is your ability to make any changes to your running software. A control plane outage is not trivial. It is a high-priority emergency. But it is not the same as an outage for user-facing workloads.

Load balancing

You will need a load balancer for the Kubernetes API servers. Can you programmatically provision a load balancer in your environment? If so, you will be able to spin up and configure it as a part of the deployment of your cluster's control plane. Think through the access policies to your cluster's API and, subsequently, what firewalls your load balancer will sit behind. You almost certainly will not make this accessible from the public internet. Remote access to your cluster's control plane is far more commonly done so via a VPN that provides access to the local network that your cluster resides on. On the other hand, if you have workloads that are publicly exposed, you will need a separate and distinct load balancer that connects to your cluster's ingress. In most cases this load balancer will serve all incoming requests to the various workloads in your cluster. There is little value in deploying a load balancer and cluster ingress for each workload that is exposed to requests from outside the cluster. If running a dedicated etcd cluster, do not put a load balancer between the Kubernetes API and etcd. The etcd client that the API uses will handle the connections to etcd without the need for a load balancer.

Automation Strategies

In automating the infrastructure components for your Kubernetes clusters, you have some strategic decisions to make. We'll break this into two categories, the first being the tools that exist today that you can leverage. Then, we'll talk about how Kubernetes operators can be used in this regard. Recognizing that automation capabilities will look very different for bare metal installations, we will start from the assumption that you have an API with which to provision machines or include them in a pool for Kubernetes deployment. If that is not the case, you will need to fill in the gaps with manual operations up to the point where you do have programmatic access and control. Let's start with some of the tools you may have at your disposal.

Infra management tools

Tools such as Terraform and CloudFormation for AWS allow you to declare the desired state for your compute and networking infrastructure and then apply that state. They use data formats or configuration languages that allow you to define the outcome you require in text files and then tell a piece of software to satisfy the desired state declared in those text files.

They are advantageous in that they use tooling that engineers can readily adopt and get outcomes with. They are good at simplifying the process of relatively complex infrastructure provisioning. They excel when you have a prescribed set of infrastructure that needs to be stamped out repeatedly and there is not a lot of variance between instances of the infrastructure. It greatly lends itself to the principle of immutable infrastructure because the repeatability is reliable, and infrastructure *replacement* as opposed to *mutation* becomes quite manageable.

These tools begin to decline in value when the infrastructure requirements become significantly complex, dynamic, and dependent on variable conditions. For example, if you are designing Kubernetes deployment systems across multiple cloud providers, these tools will become cumbersome. Data formats like JSON and configuration languages are not good at expressing conditional statements and looping functions. This is where general-purpose programming languages shine.

In development stages, infra management tools are very commonly used successfully. They are indeed used to manage production environments in certain shops, too. But they become cumbersome to work with over time and often take on a kind of technical debt that is almost impossible to pay down. For these reasons, strongly consider using or developing Kubernetes operators for this purpose.

Kubernetes operators

If infra management tools impose limitations that warrant writing software using general-purpose programming languages, what form should that software take? You could write a web application to manage your Kubernetes infrastructure. Or a command-line tool. If considering custom software development for this purpose, strongly consider Kubernetes operators.

In the context of Kubernetes, operators use custom resources and custom-built Kubernetes controllers to manage systems. Controllers use a method of managing state that is powerful and reliable. When you create an instance of a Kubernetes resource, the controller responsible for that resource kind is notified by the API server via its watch mechanism and then uses the declared desired state in the resource spec as instructions to fulfill the desired state. So extending the Kubernetes API with new resource kinds that represent infrastructure concerns, and developing Kubernetes operators to manage the state of these infrastructure resources, is very powerful. The topic of Kubernetes operators is covered in more depth in Chapter 11.

This is exactly what the Cluster API project is. It is a collection of Kubernetes operators that can be used to manage the infrastructure for Kubernetes clusters. And you can certainly leverage that open source project for your purposes. In fact, we would recommend you examine this project to see if it may fit your needs before starting a similar project from scratch. And if it doesn't fulfill your requirements, could your team get involved in contributing to that project to help develop the features and/or supported infrastructure providers that you require?

Kubernetes offers excellent options for automating the management of containerized software deployments. Similarly, it offers considerable benefits for cluster infrastructure automation strategies through the use of Kubernetes operators. Strongly consider using and, where possible, contributing to projects such as Cluster API. If you have custom requirements and prefer to use infrastructure management tools, you can certainly be successful with this option. However, your solutions will have less

flexibility and more workarounds due to the limitations of using configuration languages and formats rather than full-featured programming languages.

Machine Installations

When the machines for your cluster are spun up, they will need an operating system, certain packages installed, and configurations written. You will also need some utility or program to determine environmental and other variable values, apply them, and coordinate the process of starting the Kubernetes containerized components.

There are two broad strategies commonly used here:

- Configuration management tools
- Machine images

Configuration Management

Configuration management tools such as Ansible, Chef, Puppet, and Salt gained popularity in a world where software was installed on virtual machines and run directly on the host. These tools are quite magnificent for automating the configuration of multitudes of remote machines. They follow varying models but, in general, administrators can declaratively prescribe how a machine must look and apply that prescription in an automated fashion.

These config management tools are excellent in that they allow you to reliably establish machine consistency. Each machine can get an effectively identical set of software and configurations installed. And it is normally done with declarative recipes or playbooks that are checked into version control. These all make them a positive solution.

Where they fall short in a Kubernetes world is the speed and reliability with which you can bring cluster nodes online. If the process you use to join a new worker node to a cluster includes a config management tool performing installations of packages that pull assets over network connections, you are adding significant time to the join process for that cluster node. Furthermore, errors occur during configuration and installation. Everything from temporarily unavailable package repositories to missing or incorrect variables can cause a config management process to fail. This interrupts the cluster node join altogether. And if you're relying on that node to join an autoscaled cluster that is resource constrained, you may well invoke or prolong an availability problem.

Machine Images

Using machine images is a superior alternative. If you use machine images with all required packages installed, the software is ready to run as soon as the machine boots

up. There is no package install that depends on the network and an available package repo. Machine images improve the reliability of the node joining the cluster and considerably shorten the lead time for the node to be ready to accept traffic.

The added beauty of this method is that you can often use the config management tools you are familiar with to build the machine images. For example, using Packer from HashiCorp you can employ Ansible to build an Amazon Machine Image and have that prebuilt image ready to apply to your instances whenever they are needed. An error running an Ansible playbook to build a machine image is not a big deal. In contrast, having a playbook error occur that interrupts a worker node joining a cluster could induce a significant production incident.

You can—and should—still keep the assets used for builds in version control, and all aspects of the installations can remain declarative and clear to anyone that inspects the repository. Anytime upgrades or security patches need to occur, the assets can be updated, committed and, ideally, run automatically according to a pipeline once merged.

Some decisions involve difficult trade-offs. Some are dead obvious. This is one of those. Use prebuilt machine images.

What to Install

So what do you need to install on the machine?

To start with the most obvious, you need an operating system. A Linux distribution that Kubernetes is commonly used and tested with is the safe bet. RHEL/CentOS or Ubuntu are easy choices. If you have enterprise support for, or if you are passionate about, another distro and you're willing to invest a little extra time in testing and development, that is fine, too. Extra points if you opt for a distribution designed for containers such as Flatcar Container Linux.

To continue in order of obviousness, you will need a container runtime such as docker or containerd. When running containers, one must have a container runtime.

Next is the kubelet. This is the interface between Kubernetes and the containers it orchestrates. This is the component that is installed on the machine that coordinates the containers. Kubernetes is a containerized world. Modern conventions follow that Kubernetes itself runs in containers. With that said, the kubelet is one of the components that runs as a regular binary or process on the host. There have been attempts to run the kubelet as a container, but that just complicates things. Don't do that. Install the kubelet on the host and run the rest of Kubernetes in containers. The mental model is clear and the practicalities hold true.

So far we have a Linux OS, a container runtime to run containers, and an interface between Kubernetes and the container runtime. Now we need something that can

bootstrap the Kubernetes control plane. The kubelet can get containers running, but without a control plane it doesn't yet know what Kubernetes Pods to spin up. This is where kubeadm and static Pods come in.

Kubeadm is far from the only tool that can perform this bootstrapping. But it has gained wide adoption in the community and is used successfully in many enterprise production systems. It is a command-line program that will, in part, stamp out the static Pod manifests needed to get Kubernetes up and running. The kubelet can be configured to watch a directory on the host and run Pods for any Pod manifest it finds there. Kubeadm will configure the kubelet appropriately and deposit the manifests as needed. This will bootstrap the core, essential Kubernetes control plane components, notably etcd, kube-apiserver, kube-scheduler, and kube-controller-manager.

Thereafter, the kubelet will get all further instructions to create Pods from manifests submitted to the Kubernetes API. Additionally, kubeadm will generate bootstrap tokens you can use to securely join other nodes to your shiny new cluster.

Lastly, you will need some kind of *bootstrap utility*. The Cluster API project uses Kubernetes custom resources and controllers for this. But a command-line program installed on the host also works well. The primary function of this utility is to call kubeadm and manage runtime configurations. When the machine boots, arguments provided to the utility allow it to configure the bootstrapping of Kubernetes. For example, in AWS you can leverage user data to run your bootstrap utility and pass arguments to it that will inform which flags should be added to the kubeadm command or what to include in a kubeadm config file. Minimally, it will include a runtime config that tells the bootstrap utility whether to create a new cluster with `kubeadm init` or join the machine to an existing cluster with `kubeadm join`. It should also include a secure location to store the bootstrap token if initializing, or to retrieve the bootstrap token if joining. These tokens ensure only approved machines are attached to your cluster, so treat them with care. To gain a clear idea of what runtime configs you will need to provide to your bootstrap utility, run through a manual install of Kubernetes using kubeadm, which is well documented in the official docs. As you run through those steps it will become apparent what will be needed to meet your requirements in your environment. Figure 2-5 illustrates the steps involved in bringing up a new machine to create the first control plane node in a new Kubernetes cluster.

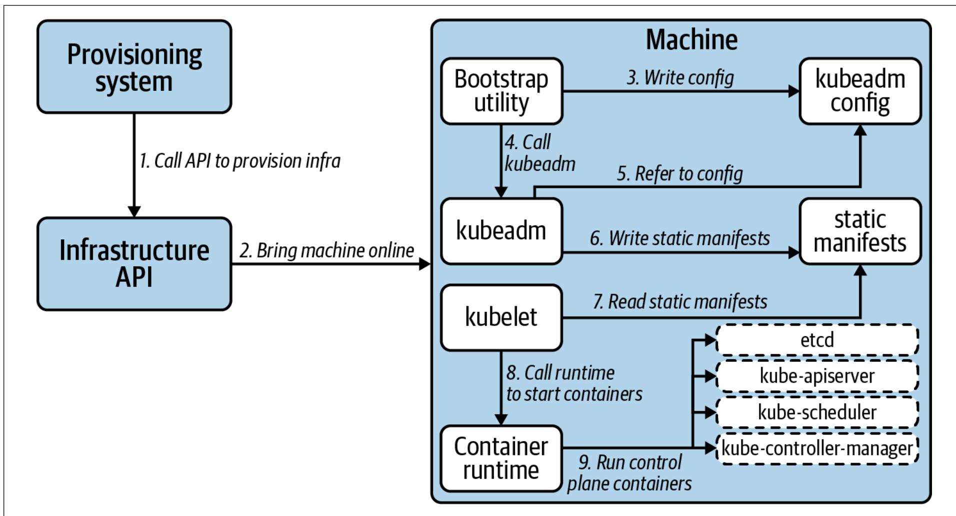


Figure 2-5. Bootstrapping a machine to initialize Kubernetes.

Now that we've covered what to install on the machines that are used as part of a Kubernetes cluster, let's move on to the software that runs in containers to form the control plane for Kubernetes.

Containerized Components

The static manifests used to spin up a cluster should include those essential components of the control plane: etcd, kube-apiserver, kube-scheduler, and kube-controller-manager. You can provide additional custom Pod manifests as needed, but strictly limit them to Pods that absolutely need to run before the Kubernetes API is available or registered into a federated system. If a workload can be installed by way of the API server later on, do so. Any Pods created with static manifests can be managed only by editing those static manifests on the machine's disk, which is much less accessible and prone to automation.

If using kubeadm, which is strongly recommended, the static manifests for your control plane, including etcd, will be created when a control plane node is initialized with `kubeadm init`. Any flag specifications you need for these components can be passed to kubeadm using the kubeadm config file. The bootstrap utility that we discussed in the previous section that calls kubeadm can write a templated kubeadm config file, for example.

Avoid customizing the static Pod manifests directly with your bootstrap utility. If really necessary, you can perform separate static manifest creation and cluster initialization steps with kubeadm that will give you the opportunity to inject customization if needed, but only do so if it's important and cannot be achieved via the kubeadm

config. A simpler, less complicated bootstrapping of the Kubernetes control plane will be more robust, faster, and will be far less likely to break with Kubernetes version upgrades.

Kubeadm will also generate self-signed TLS assets that are needed to securely connect components of your control plane. Again, avoid tinkering with this. If you have security requirements that demand using your corporate CA as a source of trust, then you can do so. If this is a requirement, it's important to be able to automate the acquisition of the intermediate authority. And keep in mind that if your cluster bootstrapping systems are secure, the trust of the self-signed CA used by the control plane will be secure and will be valid only for the control plane of a single cluster.

Now that we've covered the nuts and bolts of installing Kubernetes, let's dive into the immediate concerns that come up once you have a running cluster. We'll begin with approaches for getting the essential add-ons installed onto Kubernetes. These add-ons constitute the components you need to have in addition to Kubernetes to deliver a production-ready application platform. Then we'll get into the concerns and strategies for carrying out upgrades to your platform.

Add-ons

Cluster add-ons broadly cover those additions of platform services layered onto a Kubernetes cluster. We will not cover *what* to install as a cluster add-on in this section. That is essentially the topic of the rest of the chapters in this book. Rather, this is a look at *how* to go about installing the components that will turn your raw Kubernetes cluster into a production-ready, developer-friendly platform.

The add-ons that you add to a cluster should be considered as a part of the deployment model. Add-on installation will usually constitute the final phase of a cluster deployment. These add-ons should be managed and versioned in combination with the Kubernetes cluster itself. It is useful to consider Kubernetes and the add-ons that comprise the platform as a package that is tested and released together since there will inevitably be version and configuration dependencies between certain platform components.

Kubeadm installs "required" add-ons that are necessary to pass the Kubernetes project's conformance tests, including cluster DNS and kube-proxy, which implements Kubernetes Service resources. However, there are many more, similarly critical components that will need to be applied after kubeadm has finished its work. The most glaring example is a container network interface plug-in. Your cluster will not be good for much without a Pod network. Suffice to say you will end up with a significant list of components that need to be added to your cluster, usually in the form of DaemonSets, Deployments, or StatefulSets that will add functionality to the platform you're building on Kubernetes.

Earlier, in "Architecture and Topology" on page 28, we discussed cluster federation and the registration of new clusters into that system. That is usually a precursor to add-on installation because the systems and definitions for installation often live in a management cluster.

Whatever the architecture used, once registration is achieved, the installation of cluster add-ons can be triggered. This installation process will be a series of calls to the cluster's API server to create the Kubernetes resources needed for each component. Those calls can come from a system outside the cluster or inside.

One approach to installing add-ons is to use a continuous delivery pipeline using existing tools such as Jenkins. The "continuous" part is irrelevant in this context since the trigger is not a software update but rather a new target for installation. The "continuous" part of CI and CD usually refers to automated rollouts of software once new changes have been merged into a branch of version-controlled source code. Triggering installations of cluster add-on software into a newly deployed cluster is an entirely different mechanism but is useful in that the pipeline generally contains the capabilities needed for the installations. All that is needed to implement is the call to run a pipeline in response to the creation of a new cluster along with any variables to perform proper installation.

Another approach that is more native to Kubernetes is to use a Kubernetes operator for the task. This more advanced approach involves extending the Kubernetes API with one or more custom resources that allow you to define the add-on components for the cluster and their versions. It also involves writing the controller logic that can execute the proper installation of the add-on components given the defined spec in the custom resource.

This approach is useful in that it provides a central, clear source of truth for what the add-ons are for a cluster. But more importantly, it offers the opportunity to programmatically manage the ongoing life cycle of these add-ons. The drawback is the complexity of developing and maintaining more complex software. If you take on this complexity, it should be because you will implement those day-2 upgrades and ongoing management that will greatly reduce future human toil. If you stop at day-1 installation and do not develop the logic and functionality to manage upgrades, you will be taking on a significant software engineering cost for little ongoing benefit. Kubernetes operators offer the most value in ongoing operational management with their watch functionality of the custom resources that represent desired state.

To be clear, the add-on operator concept isn't necessarily entirely independent from external systems such as a CI/CD. In reality, they are far more likely to be used in conjunction. For example, you may use a CD pipeline to install the operator and add-on custom resources and then let the operator take over. Also, the operator will likely need to fetch manifests for installation, perhaps from a code repository that contains templated Kubernetes manifests for the add-ons.

Using an operator in this manner reduces external dependencies, which drives improved reliability. However, external dependencies cannot be eliminated altogether. Using an operator to solve add-ons should be undertaken only when you have engineers that know the Kubernetes operator pattern well and have experience leveraging it. Otherwise, stick with tools and systems that your team knows well while you advance the knowledge and experience of your team in this domain.

That brings us to the conclusion of the "day 1" concerns: the systems to install a Kubernetes cluster and its add-ons. Now we will turn to the "day 2" concern of upgrades.

Upgrades

Cluster life cycle management is closely related to cluster deployment. A cluster deployment system doesn't necessarily need to account for future upgrades; however, there are enough overlapping concerns to make it advisable. At the very least, your upgrade strategy needs to be solved before going to production. Being able to deploy the platform without the ability to upgrade and maintain it is hazardous at best. When you see production workloads running on versions of Kubernetes that are way behind the latest release, you are looking at the outcome of developing a cluster deployment system that has been deployed to production before upgrade capabilities were added to the system. When you first go to production with revenue-producing workloads running, considerable engineering budget will be spent attending to features you find missing or to sharp edges you find your team cutting themselves on. As time goes by, those features will be added and the sharp edges removed, but the point is they will naturally take priority while the upgrade strategy sits in the backlog getting stale. Budget early for those day-2 concerns. Your future self will thank you.

In addressing this subject of upgrades we will first look at versioning your platform to help ensure dependencies are well understood for the platform itself and for the workloads that will use it. We will also address how to approach planning for rollbacks in the event something goes wrong and the testing to verify that everything has gone according to plan. Finally, we will compare and contrast specific strategies for upgrading Kubernetes.

Platform Versioning

First of all, version your platform and document the versions of all software used in that platform. That includes the operating system version of the machines and all packages installed on them, such as the container runtime. It obviously includes the version of Kubernetes in use. And it should also include the version of each add-on that is added to make up your application platform. It is somewhat common for teams to adopt the Kubernetes version for their platform so that everyone knows that version 1.18 of the application platform uses Kubernetes version 1.18 without any

mental overhead or lookup. This is of trivial importance compared to the fact of just doing the versioning and documenting it. Use whatever system your team prefers. But have the system, document the system, and use it religiously. My only objection to pinning your platform version to any component of that system is that it may occasionally induce confusion. For example, if you need to update your container runtime's version due to a security vulnerability, you should reflect that in the version of your platform. If using semantic versioning conventions, that would probably look like a change to the bugfix version number. That may be confused with a version change in Kubernetes itself, i.e., v1.18.5 → 1.18.6. Consider giving your platform its own independent version numbers, especially if using semantic versioning that follows the major/minor/bugfix convention. It's almost universal that software has its own independent version with dependencies on other software and their versions. If your platform follows those same conventions, the meaning will be immediately clear to all engineers.

Plan to Fail

Start from the premise that something will go wrong during the upgrade process. Imagine yourself in the situation of having to recover from a catastrophic failure, and use that fear and anguish as motivation to prepare thoroughly for that outcome. Build automation to take and restore backups for your Kubernetes resources—both with direct etcd snapshots as well as Velero backups taken through the API. Do the same for the persistent data used by your applications. And address disaster recovery for your critical applications and their dependencies directly. For complex, stateful, distributed applications it will likely not be enough to merely restore the application's state and Kubernetes resources without regard to order and dependencies. Brainstorm all the possible failure modes, and develop automated recovery systems to remedy and then test them. For the most critical workloads and their dependencies, consider having standby clusters ready to fail over to—and then automate and test those fail-overs where possible.

Consider your rollback paths carefully. If an upgrade induces errors or outages that you cannot immediately diagnose, having rollback options is good insurance. Complex distributed systems can take time to troubleshoot, and that time can be extended by the stress and distraction of production outages. Predetermined playbooks and automation to fall back on are more important than ever when dealing with complex Kubernetes-based platforms. But be practical and realistic. In the real world, rollbacks are not always a good option. For example, if you're far enough along in an upgrade process, rolling all earlier changes back may be a terrible idea. Think that through ahead of time, know where your points of no return are, and strategize before you execute those operations live.

Integration Testing

Having a well-documented versioning system that includes all component versions is one thing, but how you manage these versions is another. In systems as complex as Kubernetes-based platforms, it is a considerable challenge to ensure everything integrates and works together as expected every time. Not only is compatibility between all components of the platform critical, but compatibility between the workloads that run on the platform and the platform itself must also be tested and confirmed. Lean toward platform agnosticism for your applications to reduce possible problems with platform compatibility, but there are many instances when application workloads yield tremendous value when leveraging platform features.

Unit testing for all platform components is important, along with all other sound software engineering practices. But integration testing is equally vital, even if considerably more challenging. An excellent tool to aid in this effort is the Sonobuoy conformance test utility. It is most commonly used to run the upstream Kubernetes end-to-end tests to ensure you have a properly running cluster; i.e., all the cluster's components are working together as expected. Often teams will run a Sonobuoy scan after a new cluster is provisioned to automate what would normally be a manual process of examining control plane Pods and deploying test workloads to ensure the cluster is properly operational. However, I suggest taking this a couple of steps further. Develop your own plug-ins that test the specific functionality and features of your platform. Test the operations that are critical to your organization's requirements. And run these scans routinely. Use a Kubernetes CronJob to run at least a subset of plug-ins, if not the full suite of tests. This is not exactly available out of the box today but can be achieved with a little engineering and is well worth the effort: expose the scan results as metrics that can be displayed in dashboards and alerted upon. These conformance scans can essentially test that the various parts of a distributed system are working together to produce the functionality and features you expect to be there and constitute a very capable automated integration testing approach.

Again, integration testing must be extended to the applications that run on the platform. Different integration testing strategies will be employed by different app dev teams, and this may be largely out of the platform team's hands, but advocate strongly for it. Run the integrations tests on a cluster that closely resembles the production environment, but more on that shortly. This will be more critical for workloads that leverage platform features. Kubernetes operators are a compelling example of this. These extend the Kubernetes API and are naturally deeply integrated with the platform. And if you're using an operator to deploy and manage the life cycle for any of your organization's software systems, it is imperative that you perform integration tests across versions of your platform, especially when Kubernetes version upgrades are involved.

Strategies

We're going to look at three strategies for upgrading your Kubernetes-based platforms:

- Cluster replacement
- Node replacement
- In-place upgrades

We're going to address them in order of highest cost with lowest risk to lowest cost with highest risk. As with most things, there is a trade-off that eliminates the opportunity for a one-size-fits-all, universally ideal solution. The costs and benefits need to be considered to find the right solution for your requirements, budget, and risk tolerance. Furthermore, within each strategy, there are degrees of automation and testing that, again, will depend on factors such as engineering budget, risk tolerance, and upgrade frequency.

Keep in mind, these strategies are not mutually exclusive. You can use combinations. For example, you could perform in-place upgrades for a dedicated etcd cluster and then use node replacements for the rest of the Kubernetes cluster. You *can* also use different strategies in different tiers where the risk tolerances are different. However, it is advisable to use the same strategy everywhere so that the methods you employ in production have first been thoroughly tested in development and staging.

Whichever strategy you employ, a few principles remain constant: test thoroughly and automate as much as is practical. If you build automation to perform actions and test that automation thoroughly in testing, development, and staging clusters, your production upgrades will be far less likely to produce issues for end users and far less likely to invoke stress in your platform operations team.

Cluster replacement

Cluster replacement is the highest cost, lowest risk solution. It is low risk in that it follows immutable infrastructure principles applied to the entire cluster. An upgrade is performed by deploying an entirely new cluster alongside the old. Workloads are migrated from the old cluster to the new. The new, upgraded cluster is scaled out as needed as workloads are migrated on. The old cluster's worker nodes are scaled in as workloads are moved off. But throughout the upgrade process there is an addition of an entirely distinct new cluster and the costs associated with it. The scaling out of the new and scaling in of the old mitigates this cost, which is to say that if you are upgrading a 300-node production cluster, you do not need to provision a new cluster with 300 nodes at the outset. You would provision a cluster with, say, 20 nodes. And when the first few workloads have been migrated, you can scale in the old cluster that has reduced usage and scale out the new to accommodate other incoming workloads.

The use of cluster autoscaling and cluster overprovisioning can make this quite seamless, but upgrades alone are unlikely to be a sound justification for using those technologies. There are two common challenges when tackling a cluster replacement.

The first is managing ingress traffic. As workloads are migrated from one cluster to the next, traffic will need to be rerouted to the new, upgraded cluster. This implies that DNS for the publicly exposed workloads does not resolve to the cluster ingress, but rather to a global service load balancer (GSLB) or reverse proxy that, in turn, routes traffic to the cluster ingress. This gives you a point from which to manage traffic routing into multiple clusters.

The other is persistent storage availability. If using a storage service or appliance, the same storage needs to be accessible from both clusters. If using a managed service such as a database service from a public cloud provider, you must ensure the same service is available from both clusters. In a private datacenter this could be a networking and firewalling question. In the public cloud it will be a question of networking and availability zones; for example, AWS EBS volumes are available from specific availability zones. And managed services in AWS often have specific Virtual Private Clouds (VPCs) associated. You may consider using a single VPC for multiple clusters for this reason. Oftentimes Kubernetes installers assume a VPC per cluster, but this isn't always the best model.

Next, you will concern yourself with workload migrations. Primarily, we're talking about the Kubernetes resources themselves—the Deployments, Services, ConfigMaps, etc. You can do this workload migration in one of two ways:

1. Redeploy from a declared source of truth
2. Copy the existing resources over from the old cluster

The first option would likely involve pointing your deployment pipeline at the new cluster and having it redeploy the same resource to the new cluster. This assumes the source of truth for your resource definitions that you have in version control is reliable and that no in-place changes have taken place. In reality, this is quite uncommon. Usually, humans, controllers, and other systems have made in-place changes and adjustments. If this is the case, you will need go with option 2 and make a copy of the existing resources and deploy them to the new cluster. This is where a tool like Velero can be extremely valuable. Velero is more commonly touted as a backup tool, but its value as a migration tool is as high or possibly even higher. Velero can take a snapshot of all resources in your cluster, or a subset. So if you migrate workloads one Namespace at a time, you can take snapshots of each Namespace at the time of migration and restore those snapshots into the new cluster in a highly reliable manner. It takes these snapshots not directly from the etcd data store, but rather through the Kubernetes API, so as long as you can provide access to Velero to the API server for both clusters, this method can be very useful. Figure 2-6 illustrates this approach.

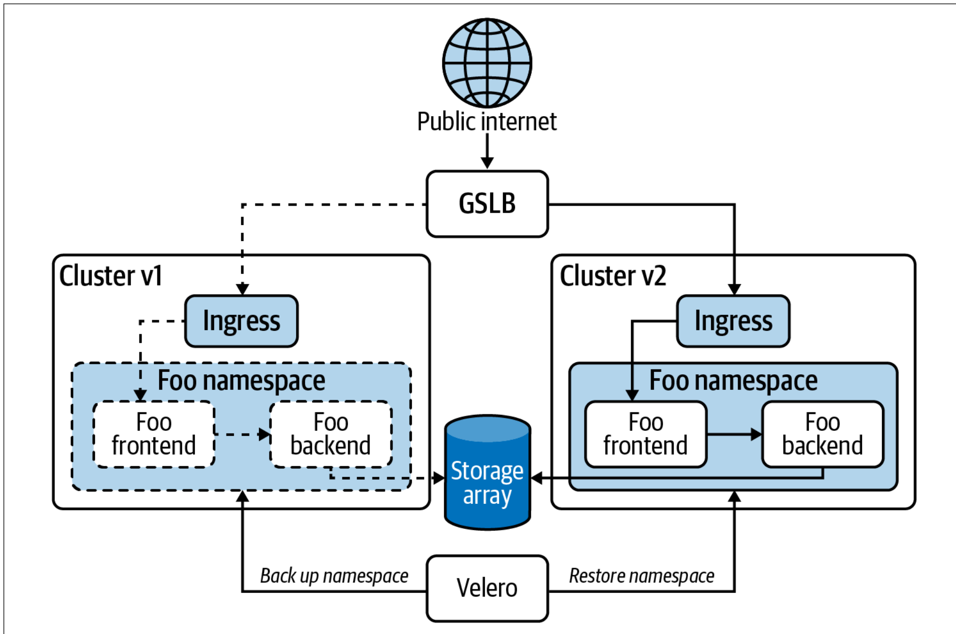


Figure 2-6. Migrating workloads between clusters with a backup and restore using Velero.

Node replacement

The node replacement option represents a middle ground for cost and risk. It is a common approach and is supported by Cluster API. It is a palatable option if you're managing larger clusters and compatibility concerns are well understood. Those compatibility concerns represent one of the biggest risks for this method because you are upgrading the control plane in-place as far as your cluster services and workloads are concerned. If you upgrade Kubernetes in-place and an API version that one of your workloads is using is no longer present, your workload could suffer an outage. There are several ways to mitigate this:

- Read the Kubernetes release notes. Before rolling out a new version of your platform that includes a Kubernetes version upgrade, read the CHANGELOG thoroughly. Any API deprecations or removals are well documented there, so you will have plenty of advance notice.
- Test thoroughly before production. Run new versions of your platform extensively in development and staging clusters before rolling out to production. Get the latest version of Kubernetes running in dev shortly after it is released and you will be able to thoroughly test and still have recent releases of Kubernetes running in production.

- Avoid tight coupling with the API. This doesn't apply to platform services that run in your cluster. Those, by their nature, need to integrate closely with Kubernetes. But keep your end user, production workloads as platform-agnostic as possible. Don't have the Kubernetes API as a dependency. For example, your application should know nothing of Kubernetes Secrets. It should simply consume an environment variable or read a file that is exposed to it. That way, as long as the manifests used to deploy your app are upgraded, the application workload itself will continue to run happily, regardless of API changes. If you find that you want to leverage Kubernetes features in your workloads, consider using a Kubernetes operator. An operator outage should not affect the availability of your application. An operator outage will be an urgent problem to fix, but it will not be one your customers or end users should see, which is a world of difference.

The node replacement option can be very beneficial when you build machine images ahead of time that are well tested and verified. Then you can bring up new machines and readily join them to the cluster. The process will be rapid because all updated software, including operating system and packages, are already installed and the processes to deploy those new machines can use much the same process as original deployment.

When replacing nodes for your cluster, start with the control plane. If you're running a dedicated etcd cluster, start there. The persistent data for your cluster is critical and must be treated carefully. If you encounter a problem upgrading your first etcd node, if you are properly prepared, it will be relatively trivial to abort the upgrade. If you upgrade all your worker nodes and the Kubernetes control plane, then find yourself with issues upgrading etcd, you are in a situation where rolling back the entire upgrade is not practical—you need to remedy the live problem as a priority. You have lost the opportunity to abort the entire process, regroup, retest, and resume later. You need to solve that problem or at the very least diligently ensure that you can leave the existing versions as-is safely for a time.

For a dedicated etcd cluster, consider replacing nodes subtractively; i.e., remove a node and then add in the upgraded replacement, as opposed to first adding a node to the cluster and then removing the old. This method gives you the opportunity to leave the member list for each etcd node unchanged. Adding a fourth member to a three-node etcd cluster, for example, will require an update to all etcd nodes' member list, which will require a restart. It will be far less disruptive to drop a member and replace it with a new one that has the same IP address as the old, if possible. The etcd documentation on upgrades is excellent and may lead you to consider doing in-place upgrades for etcd. This will necessitate in-place upgrades to OS and packages on the machine as applicable, but this will often be quite palatable and perfectly safe.

For the control plane nodes, they can be replaced additively. Using `kubeadm join` with the `--control-plane` flag on new machines that have the upgraded Kubernetes binaries—kubeadm, kubectl, kubelet—installed. As each of the control plane nodes comes online and is confirmed operational, one old-versioned node can be drained and then deleted. If you are running etcd colocated on the control plane nodes, include etcd checks when confirming operationality and etcdctl to manage the members of the cluster as needed.

Then you can proceed to replace the worker nodes. These can be done additively or subtractively—one at a time or several at a time. A primary concern here is cluster utilization. If your cluster is highly utilized, you will want to add new worker nodes before draining and removing existing nodes to ensure you have sufficient compute resources for the displaced Pods. Again, a good pattern is to use machine images that have all the updated software installed that are brought online and use `kubeadm join` to be added to the cluster. And, again, this could be implemented using many of the same mechanisms as used in cluster deployment. Figure 2-7 illustrates this operation of replacing control plane nodes one at a time and worker nodes in batches.

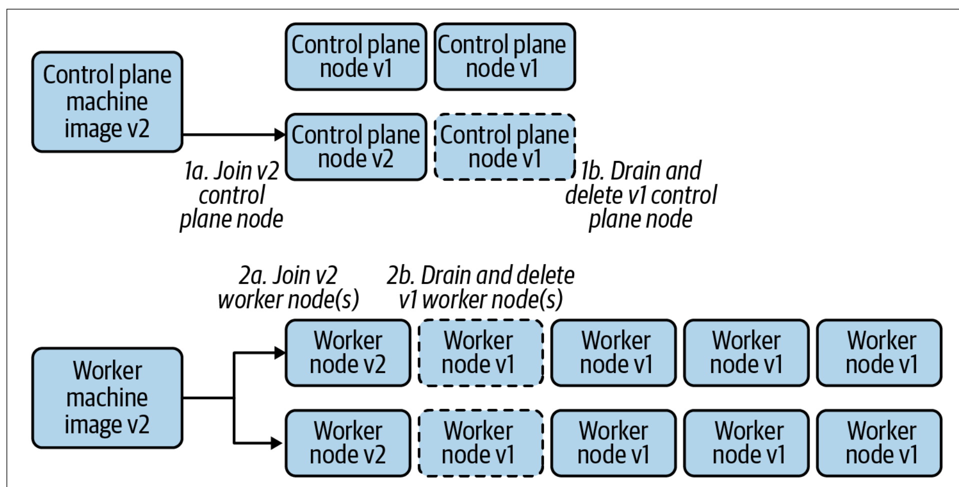


Figure 2-7. Performing upgrades by replacing nodes in a cluster.

In-place upgrades

In-place upgrades are appropriate in resource-constrained environments where replacing nodes is not practical. The rollback path is more difficult and, hence, the risk is higher. But this can and should be mitigated with comprehensive testing. Keep in mind as well that Kubernetes in production configurations is a highly available system. If in-place upgrades are done one node at a time, the risk is reduced. So, if using a config management tool such as Ansible to execute the steps of this upgrade operation, resist the temptation to hit all nodes at once in production.

For etcd nodes, following the documentation for that project, you will simply take each node offline, one at a time, performing the upgrade for OS, etcd, and other packages, and then bringing it back online. If running etcd in a container, consider pre-pulling the image in question prior to bringing the member offline to minimize downtime.

For the Kubernetes control plane and worker nodes, if kubeadm was used for initializing the cluster, that tool should also be used for upgrades. The upstream docs have detailed instructions on how to perform this process for each minor version upgrade from 1.13 forward. At the risk of sounding like a broken record, as always, plan for failure, automate as much as possible, and test extensively.

That brings us to end of upgrade options. Now, let's circle back around to the beginning of the story—what mechanisms you use to trigger these cluster provisioning and upgrade options. We're tackling this topic last because it requires the context of everything we've covered so far in this chapter.

Triggering Mechanisms

Now that we've looked at all the concerns to solve for in your Kubernetes deployment model, it's useful to consider the triggering mechanisms that fire off the automation for installation and management, whatever form that takes. Whether using a Kubernetes managed service, a prebuilt installer, or your own custom automation built from the ground up, how you fire off cluster builds, cluster scaling, and cluster upgrades is important.

Kubernetes installers generally have a CLI tool that can be used to initiate the installation process. However, using that tool in isolation leaves you without a single source of truth or cluster inventory record. Managing your cluster inventory is difficult when you don't have a list of that inventory.

A GitOps approach has become popular in recent years. In this case the source of truth is a code repository that contains the configurations for the clusters under management. When configurations for a new cluster are committed, automation is triggered to provision a new cluster. When existing configurations are updated, automation is triggered to update the cluster, perhaps to scale the number of worker nodes or perform an upgrade of Kubernetes and the cluster add-ons.

Another approach that is more Kubernetes-native is to represent clusters and their dependencies in Kubernetes custom resources and then use Kubernetes operators to respond to the declared state in those custom resources by provisioning clusters. This is the approach taken by projects like Cluster API. The sources of truth in this case are the Kubernetes resources stored in etcd in the management cluster. However, multiple management clusters for different regions or tiers are commonly employed. Here, the GitOps approach can be used in conjunction whereby the cluster resource

manifests are stored in source control and the pipeline submits the manifests to the appropriate management cluster. In this way, you get the best of both the GitOps and Kubernetes-native worlds.

Summary

When developing a deployment model for Kubernetes, consider carefully what managed services or existing Kubernetes installers (free and licensed) you may leverage. Keep automation as a guiding principle for all the systems you build. Wrap your wits around all the architecture and topology concerns, particularly if you have uncommon requirements that need to be met. Think through the infrastructure dependencies and how to integrate them into your deployment process. Consider carefully how to manage the machines that will comprise your clusters. Understand the containerized components that will form the control plane of your cluster. Develop consistent patterns for installing the cluster add-ons that will provide the essential features of your app platform. Version your platform and get your day-2 management and upgrade paths in place before you put production workloads on your clusters.

CHAPTER 3
Container Runtime

Kubernetes is a container orchestrator. Yet, Kubernetes itself does not know how to create, start, and stop containers. Instead, it delegates these operations to a pluggable component called the *container runtime*. The container runtime is a piece of software that creates and manages containers on a cluster node. In Linux, the container runtime uses a set of kernel primitives such as control groups (cgroups) and namespaces to spawn a process from a container image. In essence, Kubernetes, and more specifically, the kubelet, works together with the container runtime to run containers.

As we discussed in Chapter 1, organizations building platforms on top of Kubernetes are faced with multiple choices. Which container runtime to use is one such choice. Choice is great as it lets you customize the platform to your needs, enabling innovation and advanced use cases that might otherwise not be possible. However, given the fundamental nature of a container runtime, why does Kubernetes not provide an implementation? Why does it choose to provide a pluggable interface and offload the responsibility to another component?

To answer these questions, we will look back and briefly review the history of containers and how we got here. We will first discuss the advent of containers and how they changed the software development landscape. After all, Kubernetes would probably not exist without them. We will then discuss the Open Container Initiative (OCI), which arose from the need for standardization around container runtimes, images, and other tooling. We will review the OCI specifications and how they pertain to Kubernetes. After OCI, we will discuss the Kubernetes-specific Container Runtime Interface (CRI). The CRI is the bridge between the kubelet and the container runtime. It specifies the interface that the container runtime must implement to be compatible with Kubernetes. Finally, we will discuss how to choose a runtime for your platform and review the available options in the Kubernetes ecosystem.

The Advent of Containers

Control groups (cgroups) and namespaces are the primary ingredients necessary to implement containers. Cgroups impose limits on the amount of resources a process can use (e.g., CPU, memory, etc.), while namespaces control what a process can see (e.g., mounts, processes, network interfaces, etc.). Both these primitives have been in the Linux kernel since 2008. Even earlier in the case of namespaces. So why did containers, as we know them today, become popular years later?

To answer this question, we first need to consider the environment surrounding the software and IT industry at the time. An initial factor to think about is the complexity of applications. Application developers built applications using service-oriented architectures and even started to embrace microservices. These architectures brought various benefits to organizations, such as maintainability, scalability, and productivity. However, they also resulted in an explosion in the number of components that made up an application. Meaningful applications could easily involve a dozen services, potentially written in multiple languages. As you can imagine, developing and shipping these applications was (and continues to be) complex. Another factor to remember is that software quickly became a business differentiator. The faster you could ship new features, the more competitive your offerings. Having the ability to deploy software in a reliable manner was key to a business. Finally, the emergence of the public cloud as a hosting environment is another important factor. Developers and operations teams had to ensure that applications behaved the same across all environments, from a developer's laptop to a production server running in someone else's datacenter.

Keeping these challenges in mind, we can see how the environment was ripe for innovation. Enter Docker. Docker made containers accessible to the masses. They built an abstraction that enabled developers to build and run containers with an easy-to-use CLI. Instead of developers having to know the low-level kernel constructs needed to leverage container technology, all they had to do was type `docker run` in their terminal.

While not the answer to all our problems, containers improved many stages of the software development life cycle. First, containers and container images allowed developers to codify the application's environment. Developers no longer had to wrestle with missing or mismatched application dependencies. Second, containers impacted testing by providing reproducible environments for testing applications. Lastly, containers made it easier to deploy software to production. As long as there was a Docker Engine in the production environment, the application could be deployed with minimum friction. Overall, containers helped organizations to ship software from zero to production in a more repeatable and efficient manner.

The advent of containers also gave birth to an abundant ecosystem full of different tools, container runtimes, container image registries, and more. This ecosystem was well received but introduced a new challenge: How do we make sure that all these container solutions are compatible with each other? After all, the encapsulation and portability guarantees are one of the main benefits of containers. To solve this challenge and to improve the adoption of containers, the industry came together and collaborated on an open source specification under the umbrella of the Linux Foundation: the Open Container Initiative.

The Open Container Initiative

As containers continued to gain popularity across the industry, it became clear that standards and specifications were required to ensure the success of the container movement. The Open Container Initiative (OCI) is an open source project established in 2015 to collaborate on specifications around containers. Notable founders of this initiative included Docker, which donated runc to the OCI, and CoreOS, which pushed the needle on container runtimes with rkt.

The OCI includes three specifications: the OCI runtime specification, the OCI image specification, and the OCI distribution specification. These specs enable development and innovation around containers and container platforms such as Kubernetes. Furthermore, the OCI aims to allow end users to use containers in a portable and interoperable manner, enabling them to move between products and solutions more easily when necessary.

In the following sections, we will explore the runtime and image specifications. We will not dig into the distribution specification, as it is primarily concerned with container image registries.

OCI Runtime Specification

The OCI runtime specification determines how to instantiate and run containers in an OCI-compatible fashion. First, the specification describes the schema of a container's configuration. The schema includes information such as the container's root filesystem, the command to run, the environment variables, the user and group to use, resource limits, and more. The following snippet is a trimmed example of a container configuration file obtained from the OCI runtime specification:

```
{
    "ociVersion": "1.0.1",
    "process": {
        "terminal": true,
        "user": {
            "uid": 1,
            "gid": 1,
            "additionalGids": [
```

```
                    5,
                    6
                ]
            },
            "args": [
                "sh"
            ],
            "env": [
                "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "TERM=xterm"
            ],
            "cwd": "/",
            ...
        },
        ...
        "mounts": [
            {
                "destination": "/proc",
                "type": "proc",
                "source": "proc"
            },
        ...
        },
        ...
    }
```

The runtime specification also determines the operations that a container runtime must support. These operations include create, start, kill, delete, and state (which provides information about the container's state). In addition to the operations, the runtime spec describes the life cycle of a container and how it progresses through different stages. The life cycle stages are (1) `creating`, which is active when the container runtime is creating the container; (2) `created`, which is when the runtime has completed the `create` operation; (3) `running`, which is when the container process has started and is running; and (4) `stopped`, which is when the container process has finished.

The OCI project also houses *runc*, a low-level container runtime that implements the OCI runtime specification. Other higher-level container runtimes such as Docker, containerd, and CRI-O use runc to spawn containers according to the OCI spec, as shown in Figure 3-1. Leveraging runc enables container runtimes to focus on higher-level features such as pulling images, configuring networking, handling storage, and so on while conforming to the OCI runtime spec.

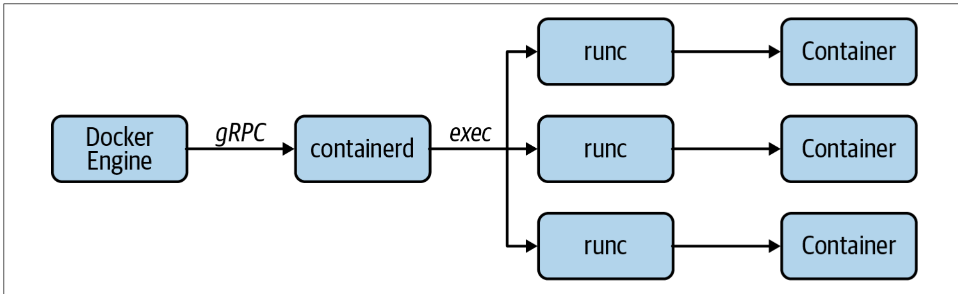


Figure 3-1. Docker Engine, containerd, and other runtimes use runc to spawn containers according to the OCI spec.

OCI Image Specification

The OCI image specification focuses on the container image. The specification defines a manifest, an optional image index, a set of filesystem layers, and a configuration. The image manifest describes the image. It includes a pointer to the image's configuration, a list of image layers, and an optional map of annotations. The following is an example manifest obtained from the OCI image specification:

```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "size": 7023,
    "digest": "sha256:b5b2b2c507a0944348e0303114d8d93aaaa081732b86451d9bce1f4..."
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 32654,
      "digest": "sha256:9834876dcfb05cb167a5c24953eba58c4ac89b1adf57f28f2f9d0..."
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 16724,
      "digest": "sha256:3c3a4604a545cdc127456d94e421cd355bca5b528f4a9c1905b15..."
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "size": 73109,
      "digest": "sha256:ec4b8955958665577945c89419d1af06b5f7636b4ac3da7f12184..."
    }
  ],
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

The *image index* is a top-level manifest that enables the creation of multiplatform container images. The image index contains pointers to each of the platform-specific manifests. The following is an example index obtained from the specification. Notice how the index points to two different manifests, one for `ppc64le/linux` and another for `amd64/linux`:

```
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7143,
      "digest": "sha256:e692418e4cbaf90ca69d05a66403747baa33ee08806650b51fab...",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "size": 7682,
      "digest": "sha256:5b0bcabd1ed22e9fb1310cf6c2dec7cdef19f0ad69efa1f392e9...",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ],
  "annotations": {
    "com.example.key1": "value1",
    "com.example.key2": "value2"
  }
}
```

Each OCI image manifest references a container image configuration. The configuration includes the image's entry point, command, working directory, environment variables, and more. The container runtime uses this configuration when instantiating a container from the image. The following snippet shows the configuration of a container image, with some fields removed for brevity:

```
{
  "architecture": "amd64",
  "config": {
    ...
    "ExposedPorts": {
      "53/tcp": {},
      "53/udp": {}
    },
    "Tty": false,
    "OpenStdin": false,
    "StdinOnce": false,
```

```
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": null,
    "Image": "sha256:7ccecf40b555e5ef2d8d3514257b69c2f4018c767e7a20dbaf4733...",
    "Volumes": null,
    "WorkingDir": "",
    "Entrypoint": [
      "/coredns"
    ],
    "OnBuild": null,
    "Labels": null
  },
  "created": "2020-01-28T19:16:47.907002703Z",
  ...
```

The OCI image spec also describes how to create and manage container image layers. Layers are essentially TAR archives that include files and directories. The specification defines different media types for layers, including uncompressed layers, gzipped layers, and nondistributable layers. Each layer is uniquely identified by a digest, usually a SHA256 sum of the contents of the layer. As we discussed before, the container image manifest references one or more layers. The references use the SHA256 digest to point to a specific layer. The final container image filesystem is the result of applying each of the layers, as listed in the manifest.

The OCI image specification is crucial because it ensures that container images are portable across different tools and container-based platforms. The spec enables the development of different image build tools, such as kaniko and Buildah for userspace container builds, Jib for Java-based containers, and Cloud Native Buildpacks for streamlined and automated builds. (We will explore some of these tools in Chapter 15). Overall, this specification ensures that Kubernetes can run container images regardless of the tooling used to build them.

The Container Runtime Interface

As we've discussed in prior chapters, Kubernetes offers many extension points that allow you to build a bespoke application platform. One of the most critical extension points is the Container Runtime Interface (CRI). The CRI was introduced in Kubernetes v1.5 as an effort to enable the growing ecosystem of container runtimes, which included rkt by CoreOS and hypervisor-based runtimes such as Intel's Clear Containers, which later became Kata Containers.

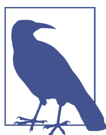
Prior to CRI, adding support for a new container runtime required a new release of Kubernetes and intimate knowledge of the Kubernetes code base. Once the CRI was established, container runtime developers could simply adhere to the interface to ensure compatibility of the runtime with Kubernetes.

Overall, the goal of the CRI was to abstract the implementation details of the container runtime away from Kubernetes, more specifically the kubelet. This is a classic example of the dependency inversion principle. The kubelet evolved from having container runtime–specific code and if-statements scattered throughout to a leaner implementation that relied on the interface. Thus, the CRI reduced the complexity of the kubelet implementation while also making it more extensible and testable. These are all important qualities of well-designed software.

The CRI is implemented using gRPC and Protocol Buffers. The interface defines two services: the *RuntimeService* and the *ImageService*. The kubelet leverages these services to interact with the container runtime. The RuntimeService is responsible for all the Pod-related operations, including creating Pods, starting and stopping containers, deleting Pods, and so on. The ImageService is concerned with container image operations, including listing, pulling, and removing container images from the node.

While we could detail the APIs of both the RuntimeService and ImageService in this chapter, it is more useful to understand the flow of perhaps the most important operation in Kubernetes: starting a Pod on a node. Thus, let's explore the interaction between the kubelet and the container runtime through CRI in the following section.

Starting a Pod



The following descriptions are based on Kubernetes v1.18.2 and containerd v1.3.4. These components use the v1alpha2 version of the CRI.

Once the Pod is scheduled onto a node, the kubelet works together with the container runtime to start the Pod. As mentioned, the kubelet interacts with the container runtime through the CRI. In this case, we will explore the interaction between the kubelet and the containerd CRI plug-in.

The containerd CRI plug-in starts a gRPC server that listens on a Unix socket. By default, this socket is located at */run/containerd/containerd.sock*. The kubelet is configured to interact with containerd through this socket with the `container-runtime` and `container-runtime-endpoint` command-line flags:

```
/usr/bin/kubelet
  --container-runtime=remote
  --container-runtime-endpoint=/run/containerd/containerd.sock
  ... other flags here ...
```

To start the Pod, the kubelet first creates a Pod sandbox using the `RunPodSandbox` method of the RuntimeService. Because a Pod is composed of one or more containers, the sandbox must be created first to establish the Linux network namespace

(among other things) for all containers to share. When calling this method, the kubelet sends metadata and configuration to containerd, including the Pod's name, unique ID, Kubernetes Namespace, DNS configuration, and more. Once the container runtime creates the sandbox, the runtime responds with a Pod sandbox ID that the kubelet uses to create containers in the sandbox.

Once the sandbox is available, the kubelet checks whether the container image is present on the node using the `ImageStatus` method of the ImageService. The `ImageStatus` method returns information about the image. When the image is not present, the method returns null and the kubelet proceeds to pull the image. The kubelet uses the `PullImage` method of the ImageService to pull the image when necessary. Once the runtime pulls the image, it responds with the image SHA256 digest, which the kubelet then uses to create the container.

After creating the sandbox and pulling the image, the kubelet creates the containers in the sandbox using the `CreateContainer` method of the RuntimeService. The kubelet provides the sandbox ID and the container configuration to the container runtime. The container configuration includes all the information you might expect, including the container image digest, command and arguments, environment variables, volume mounts, etc. During the creation process, the container runtime generates a container ID that it then passes back to the kubelet. This ID is the one you see in the Pod's status field under container statuses:

```
containerStatuses:
  - containerID: containerd://0018556b01e1662c5e7e2dcddb2bb09d0edff6cf6933...
    image: docker.io/library/nginx:latest
```

The kubelet then proceeds to start the container using the `StartContainer` method of the RuntimeService. When calling this method, it uses the container ID it received from the container runtime.

And that's it! In this section, we've learned how the kubelet interacts with the container runtime using the CRI. We specifically looked at the gRPC methods invoked when starting a Pod, which include those on the ImageService and the RuntimeService. Both of these CRI services provide additional methods that the kubelet uses to complete other tasks. Besides the Pod and container management (i.e., CRUD) methods, the CRI also defines methods to execute a command inside a container (`Exec` and `ExecSync`), attach to a container (`Attach`), forward a specific container port (`PortForward`), and others.

Choosing a Runtime

Given the availability of the CRI, platform teams get the flexibility of choice when it comes to container runtimes. The reality, however, is that over the last couple of years the container runtime has become an implementation detail. If you are using a Kubernetes distribution or leveraging a managed Kubernetes service, the container runtime will most likely be chosen for you. This is the case even for community projects such as Cluster API, which provide prebaked node images that include a container runtime.

With that said, if you do have the option to choose a runtime or have a use case for a specialized runtime (e.g., VM-based runtimes), you should be equipped with information to make that decision. In this section, we will discuss considerations you should make when choosing a container runtime.

The first question we like to ask when helping organizations in the field is which container runtime they have experience with. In most cases, organizations that have a long history with containers are using Docker and are familiar with Docker's toolchain and user experience. While Kubernetes supports Docker, we discourage its use as it has an extended set of capabilities that Kubernetes does not need, such as building images, creating container networks, and so on. In other words, the fully fledged Docker daemon is too heavy or bloated for the purposes of Kubernetes. The good news is that Docker uses containerd under the covers, one of the most prevalent container runtimes in the community. The downside is that platform operators have to learn the containerd CLI.

Another consideration to make is the availability of support. Depending on where you are getting Kubernetes from, you might get support for the container runtime. Kubernetes distributions such as VMware's Tanzu Kubernetes Grid, RedHat's OpenShift, and others usually ship a specific container runtime. You should stick to that choice unless you have an extremely compelling reason not to. In that case, ensure that you understand the support implications of using a different container runtime.

Closely related to support is conformance testing of the container runtime. The Kubernetes project, specifically the Node Special Interest Group (sig-node), defines a set of CRI validation tests and node conformance tests to ensure container runtimes are compatible and behave as expected. These tests are part of every Kubernetes release, and some runtimes might have more coverage than others. As you can imagine, the more test coverage the better, as any issues with the runtime are caught during the Kubernetes release process. The community makes all tests and results available through the Kubernetes Test Grid. When choosing a runtime, you should consider the container runtime's conformance tests and more broadly, the runtime's relationship with the overall Kubernetes project.

Lastly, you should determine if your workloads need stronger isolation guarantees than those provided by Linux containers. While less common, there are use cases that require VM-level isolation for workloads, such as executing untrusted code or running applications that require strong multitenancy guarantees. In these cases, you can leverage specialized runtimes such as Kata Containers.

Now that we have discussed the considerations you should make when choosing a runtime, let's review the most prevalent container runtimes: Docker, containerd, and CRI-O. We will also explore Kata Containers to understand how we can run Pods in VMs instead of Linux Containers. Finally, while not a container runtime or component that implements CRI, we will learn about Virtual Kubelet, as it provides another way to run workloads on Kubernetes.

Docker

Kubernetes supports the Docker Engine as a container runtime through a CRI shim called the *dockershim*. The shim is a component that's built into the kubelet. Essentially, it is a gRPC server that implements the CRI services we described earlier in this chapter. The shim is required because the Docker Engine does not implement the CRI. Instead of special-casing all the kubelet code paths to work with both the CRI and the Docker Engine, the dockershim serves as a facade that the kubelet can use to communicate with Docker via the CRI. The dockershim handles the translation between CRI calls to Docker Engine API calls. Figure 3-2 depicts how the kubelet interacts with Docker through the shim.

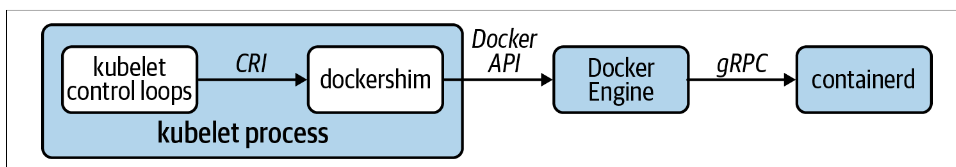


Figure 3-2. Interaction between the kubelet and Docker Engine via the dockershim.

As we mentioned earlier in the chapter, Docker leverages containerd under the hood. Thus, the incoming API calls from the kubelet are eventually relayed to containerd, which starts the containers. In the end, the spawned container ends up under containerd and not the Docker daemon:

```
systemd
  └─containerd
      └─containerd-shim -namespace moby -workdir ...
          └─nginx
              └─nginx
```

From a troubleshooting perspective, you can use the Docker CLI to list and inspect containers running on a given node. While Docker does not have the concept of Pods, the dockershim encodes the Kubernetes Namespace, Pod name, and Pod ID

into the name of containers. For example, the following listing shows the containers that belong to a Pod called `nginx` in the `default` namespace. The Pod infrastructure container (aka, pause container) is the one with the `k8s_POD_` prefix in the name:

```
$ docker ps --format='{{.ID}}\t{{.Names}}' | grep nginx_default
3c8c01f47424    k8s_nginx_nginx_default_6470b3d3-87a3-499c-8562-d59ba27bced5_3
c34ad8d80c4d    k8s_POD_nginx_default_6470b3d3-87a3-499c-8562-d59ba27bced5_3
```

You can also use the containerd CLI, `ctr`, to inspect containers, although the output is not as user friendly as the Docker CLI output. The Docker Engine uses a containerd namespace called `moby`:

```
$ ctr --namespace moby containers list
CONTAINER                    IMAGE    RUNTIME
07ba23a409f31bec7f163a...    -        io.containerd.runtime.v1.linux
0bfc5a735c213b9b296dad...    -        io.containerd.runtime.v1.linux
2d1c9cb39c674f75caf595...    -        io.containerd.runtime.v1.linux
...
```

Finally, you can use `crictl` if available on the node. The `crictl` utility is a command-line tool developed by the Kubernetes community. It is a CLI client for interacting with container runtimes over the CRI. Even though Docker does not implement the CRI, you can use `crictl` with the dockershim Unix socket:

```
$ crictl --runtime-endpoint unix:///var/run/dockershim.sock ps --name nginx
CONTAINER ID   IMAGE                 CREATED         STATE     NAME    POD ID
07ba23a409f31  nginx@sha256:b0a...   3 seconds ago   Running   nginx   ea179944...
```

containerd

containerd is perhaps the most common container runtime we encounter when building Kubernetes-based platforms in the field. At the time of writing, containerd is the default container runtime in Cluster API-based node images and is available across various managed Kubernetes offerings (e.g., AKS, EKS, and GKE).

The containerd container runtime implements the CRI through the containerd CRI plug-in. The CRI plug-in is a native containerd plug-in that is available since containerd v1.1 and is enabled by default. containerd exposes its gRPC APIs over a Unix socket at */run/containerd/containerd.sock*. The kubelet uses this socket to interact with containerd when it comes to running Pods, as depicted in Figure 3-3.

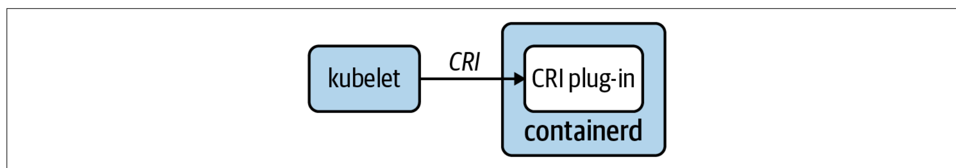


Figure 3-3. Interaction between the kubelet and containerd through the containerd CRI plug-in.

The process tree of spawned containers looks exactly the same as the process tree when using the Docker Engine. This is expected, as the Docker Engine uses containerd to manage containers:

```
systemd
  └─containerd
      └─containerd-shim -namespace k8s.io -workdir ...
          └─nginx
              └─nginx
```

To inspect containers on a node, you can use `ctr`, the containerd CLI. As opposed to Docker, the containers managed by Kubernetes are in a containerd namespace called `k8s.io` instead of `moby`:

```
$ ctr --namespace k8s.io containers ls | grep nginx
c85e47fa...    docker.io/library/nginx:latest    io.containerd.runtime.v1.linux
```

You can also use the `crictl` CLI to interact with containerd through the containerd unix socket:

```
$ crictl --runtime-endpoint unix:///run/containerd/containerd.sock ps
  --name nginx
CONTAINER ID    IMAGE          CREATED         STATE      NAME   POD ID
c85e47faf3616   4bb46517cac39  39 seconds ago  Running    nginx  73caea404b92a
```

CRI-O

CRI-O is a container runtime specifically designed for Kubernetes. As you can probably tell from the name, it is an implementation of the CRI. Thus, in contrast to Docker and containerd, it does not cater to uses outside of Kubernetes. At the time of writing, one of the primary consumers of the CRI-O container runtime is the RedHat OpenShift platform.

Similar to containerd, CRI-O exposes the CRI over a Unix socket. The kubelet uses the socket, typically located at */var/run/crio/crio.sock*, to interact with CRI-O. Figure 3-4 depicts the kubelet interacting directly with CRI-O through the CRI.

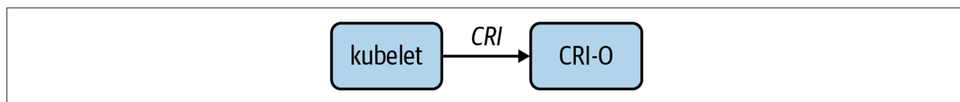


Figure 3-4. Interaction between the kubelet and CRI-O using the CRI APIs.

When spawning containers, CRI-O instantiates a process called *conmon*. Conmon is a container monitor. It is the parent of the container process and handles multiple concerns, such as exposing a way to attach to the container, storing the container's STDOUT and STDERR streams to logfiles, and handling container termination:

```
systemd
  └conmon -s -c ed779... -n k8s_nginx_nginx_default_e9115... -u8cdf0c...
      └nginx
          └nginx
```

Because CRI-O was designed as a low-level component for Kubernetes, the CRI-O project does not provide a CLI. With that said, you can use `crictl` with CRI-O as with any other container runtime that implements the CRI:

```
$ crictl --runtime-endpoint unix:///var/run/crio/crio.sock ps --name nginx
CONTAINER    IMAGE                  CREATED          STATE      NAME     POD ID
8cdf0c...    nginx@sha256:179...    2 minutes ago    Running    nginx    eabf15237...
```

Kata Containers

Kata Containers is an open source, specialized runtime that uses lightweight VMs instead of containers to run workloads. The project has a rich history, resulting from the merge of two prior VM-based runtimes: Clear Containers from Intel and RunV from Hyper.sh.

Due to the use of VMs, Kata provides stronger isolation guarantees than Linux containers. If you have security requirements that prevent workloads from sharing a Linux kernel or resource guarantee requirements that cannot be met by cgroup isolation, Kata Containers can be a good fit. For example, a common use case for Kata containers is to run multitenant Kubernetes clusters that run untrusted code. Cloud providers such as Baidu Cloud and Huawei Cloud use Kata Containers in their cloud infrastructure.

To use Kata Containers with Kubernetes, there is still a need for a pluggable container runtime to sit between the kubelet and the Kata runtime, as shown in Figure 3-5. The reason is that Kata Containers does not implement the CRI. Instead, it leverages existing container runtimes such as containerd to handle the interaction with Kubernetes. To integrate with containerd, the Kata Containers project implements the containerd runtime API, specifically the v2 containerd-shim API.

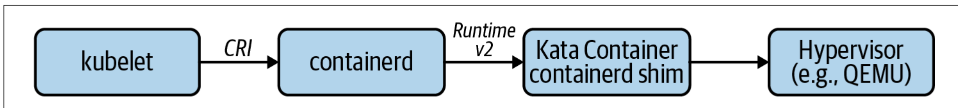


Figure 3-5. Interaction between the kubelet and Kata Containers through containerd.

Because containerd is required and available on the nodes, it is possible to run Linux container Pods and VM-based Pods on the same node. Kubernetes provides a mechanism to configure and run multiple container runtimes called Runtime Class. Using the RuntimeClass API, you can offer different runtimes in the same Kubernetes platform, enabling developers to use the runtime that better fits their needs. The following snippet is an example RuntimeClass for the Kata Containers runtime:

```yaml
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
    name: kata-containers
handler: kata
```

To run a Pod under the `kata-containers` runtime, developers must specify the runtime class name in their Pod's specification:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kata-example
spec:
  containers:
  - image: nginx
    name: nginx
  runtimeClassName: kata-containers
```

Kata Containers supports different hypervisors to run workloads, including QEMU, NEMU, and AWS Firecracker. When using QEMU, for example, we can see a QEMU process after launching a Pod that uses the `kata-containers` runtime class:

```
$ ps -ef | grep qemu
root     38290     1  0 16:02 ?        00:00:17
   /snap/kata-containers/690/usr/bin/qemu-system-x86_64
   -name sandbox-c136a9addde4f26457901ccef9de49f02556cc8c5135b091f6d36cfc97...
   -uuid aaae32b3-9916-4d13-b385-dd8390d0daf4
   -machine pc,accel=kvm,kernel_irqchip
   -cpu host
   -m 2048M,slots=10,maxmem=65005M
   ...
```

While Kata Containers provides interesting capabilities, we consider it a niche and have not seen it used in the field. With that said, if you need VM-level isolation guarantees in your Kubernetes cluster, Kata Containers is worth looking into.

Virtual Kubelet

Virtual Kubelet is an open source project that behaves like a kubelet but offers a pluggable API on the backend. While not a container runtime per se, its main purpose is to surface alternative runtimes to run Kubernetes Pods. Because of the Virtual Kubelet's extensible architecture, these alternative runtimes can essentially be any systems that can run an application, such as serverless frameworks, edge frameworks, etc. For example, as shown in Figure 3-6, the Virtual Kubelet can launch Pods on a cloud service such as Azure Container Instances or AWS Fargate.

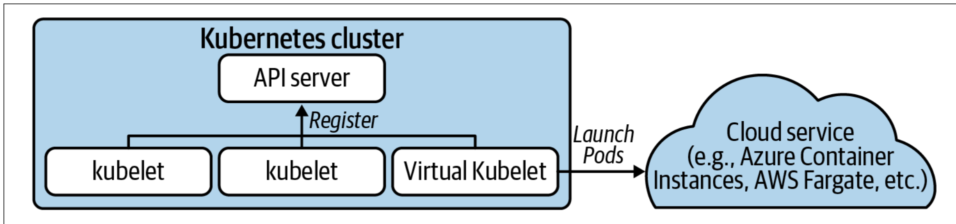


Figure 3-6. Virtual Kubelet running Pods on a cloud service, such as Azure Container Instances, AWS Fargate, etc.

The Virtual Kubelet community offers a variety of providers that you can leverage if they fit your need, including AWS Fargate, Azure Container Instances, HashiCorp Nomad, and others. If you have a more specific use case, you can implement your own provider as well. Implementing a provider involves writing a Go program using the Virtual Kubelet libraries to handle the integration with Kubernetes, including node registration, running Pods, and exporting APIs expected by Kubernetes.

Even though Virtual Kubelet enables interesting scenarios, we have yet to run into a use case that needed it in the field. With that said, it is useful to know that it exists, and you should keep it in your Kubernetes toolbox.

Summary

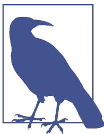
The container runtime is a foundational component of a Kubernetes-based platform. After all, it is impossible to run containerized workloads without a container runtime. As we learned in this chapter, Kubernetes uses the Container Runtime Interface (CRI) to interact with the container runtime. One of the main benefits of the CRI is its pluggable nature, which allows you to use the container runtime that best fits your needs. To give you an idea of the different container runtimes available in the ecosystem, we discussed those that we typically see in the field, such as Docker, containerd, etc. Learning about the different options and further exploring their capabilities should help you select the container runtime that satisfies the requirements of your application platform.

CHAPTER 4

Container Storage

While Kubernetes cut its teeth in the world of stateless workloads, running stateful services has become increasingly common. Even complex stateful workloads such as databases and message queues are finding their way to Kubernetes clusters. To support these workloads, Kubernetes needs to provide storage capabilities beyond ephemeral options. Namely, systems that can provide increased resilience and availability in the face of various events such as an application crashing or a workload being rescheduled to a different host.

In this chapter we are going to explore how our platform can offer storage services to applications. We'll start by covering key concerns of application persistence and storage system expectations before moving on to address the storage primitives available in Kubernetes. As we get into more advanced storage needs, we will look to the Container Storage Interface (CSI), which enables our integration with various storage providers. Lastly, we'll explore using a CSI plug-in to provide self-service storage to our applications.



Storage is a vast subject in itself. Our intentions are to give you just enough detail to make informed decisions about the storage you may offer to workloads. If storage is not your background, we highly recommend going over these concepts with your infrastructure/storage team. Kubernetes does not negate the need for storage expertise in your organization!

Storage Considerations

Before getting into Kubernetes storage patterns and options, we should take a step back and analyze some key considerations around potential storage needs. At an infrastructure and application level, it is important to think through the following requirements.

- Access modes
- Volume expansion
- Dynamic provisioning
- Backup and recovery
- Block, file, and object storage
- Ephemeral data
- Choosing a provider

Access Modes

There are three access modes that can be supported for applications:

ReadWriteOnce (RWO)
  A single Pod can read and write to the volume.

ReadOnlyMany (ROX)
  Multiple Pods can read the volume.

ReadWriteMany (RWX)
  Multiple Pods can read and write to the volume.

For cloud native applications, RWO is by far the most common pattern. When leveraging common providers such Amazon Elastic Block Storage (EBS) or Azure Disk Storage, you are limited to RWO because the disk may only be attached to one node. While this limitation may seem problematic, most cloud native applications work best with this kind of storage, where the volume is exclusively theirs and offers high-performance read/write.

Many times, we find legacy applications that have a requirement for RWX. Often, they are built to assume access to a network file system (NFS). When services need to share state, there are often more elegant solutions than sharing data over NFS; for example, the use of message queues or databases. Additionally, should an application wish to share data, it's typically best to expose this over an API, rather than grant access to its file system. This makes many use cases for RWX, at times, questionable. Unless NFS is the correct design choice, platform teams may be confronted with the tough choice of whether to offer RWX-compatible storage or request their developers

re-architect applications. Should the call be made that supporting ROX or RWX is required, there are several providers that can be integrated with, such as Amazon Elastic File System (EFS) and Azure File Share.

Volume Expansion

Over time, an application may begin to fill up its volume. This can pose a challenge since replacing the volume with a larger one would require migration of data. One solution to this is supporting volume expansion. From the perspective of a container orchestrator such as Kubernetes, this involves a few steps:

1. Request additional storage from the orchestrator (e.g., via a PersistentVolume-Claim).
2. Expand the size of the volume via the storage provider.
3. Expand the filesystem to make use of the larger volume.

Once complete, the Pod will have access to the additional space. This feature is contingent on our choice of storage backend and whether the integration in Kubernetes can facilitate the preceding steps. We will explore an example of volume expansion later in this chapter.

Volume Provisioning

There are two provisioning models available to you: dynamic and static provisioning. Static provisioning assumes volumes are created on nodes for Kubernetes to consume. Dynamic provisioning is when a driver runs within the cluster and can satisfy storage requests of workloads by talking to a storage provider. Out of these two models, dynamic provisioning, when possible, is preferred. Often, the choice between the two is a matter of whether your underlying storage system has a compatible driver for Kubernetes. We'll dive into these drivers later in the chapter.

Backup and Recovery

Backup is one of the most complex aspects of storage, especially when automated restores are a requirement. In general terms, a backup is a copy of data that is stored for use in case of data loss. Typically, we balance backup strategies with the availability guarantees of our storage systems. For example, while backups are always important, they are less critical when our storage system has a replication guarantee where loss of hardware will *not* result in loss of data. Another consideration is that applications may require different procedures to facilitate backup and restores. The idea that we can take a backup of an entire cluster and restore it at any time is typically a naive outlook, or at minimum, one that requires mountains of engineering effort to achieve.

Deciding who should be responsible for backup and recovery of applications can be one of the most challenging debates within an organization. Arguably, offering restore features as a platform service can be a "nice to have." However, it can tear at the seams when we get into application-specific complexity—for example, when an app cannot restart and needs actions to take place that are known only to developers.

One of the most popular backup solutions for both Kubernetes state and application state is Project Velero. Velero can back up Kubernetes objects should you have a desire to migrate or restore them across clusters. Additionally, Velero supports the scheduling of volume snapshots. As we dive deeper into volume snapshotting in this chapter, we'll learn that the ability to schedule and manage snapshots is *not* taken care of for us. More so, we are often given the snapshotting primitives but need to define an orchestration flow around them. Lastly, Velero supports backup and restore hooks. These enable us to run commands in the container before performing backup or recovery. For example, some applications may require stopping traffic or triggering a flush before a backup should be taken. This is made possible using hooks in Velero.

Block Devices and File and Object Storage

The storage types our applications expect are key to selecting the appropriate underlying storage and Kubernetes integration. The most common storage type used by applications is file storage. File storage is a block device with a filesystem on top. This enables applications to write to files in the way we are familiar with on any operating system.

Underlying a filesystem is a block device. Rather than establishing a filesystem on top, we can offer the device such that applications may communicate directly with raw block. Filesystems inherently add overhead to writing data. In modern software development, it's pretty rare to be concerned about filesystem overhead. However, if your use case warrants direct interaction with raw block devices, this is something certain storage systems can support.

The final storage type is object storage. Object storage deviates from files in the sense that there is not the conventional hierarchy. Object storage enables developers to take unstructured data, give it a unique identifier, add some metadata around it, and store it. Cloud-provider object stores such as Amazon S3 have become popular locations for organizations to host images, binaries, and more. This popularity has been accelerated by its fully featured web API and access control. Object stores are *most commonly* interacted with from the application itself, where the application uses a library to authenticate and interact with the provider. Since there is less standardization around interfaces for interaction with object stores, it is less common to see them integrated as platform services that applications can interact with transparently.

Ephemeral Data

While storage may imply a level of persistence that is beyond the life cycle of a Pod, there are valid use cases for supporting ephemeral data usage. By default, containers that write to their own filesystem will utilize ephemeral storage. If the container were to restart, this storage would be lost. The emptyDir volume type is available for ephemeral storage that is resilient to restarts. Not only is this resilient to container restarts, but it can be used to share files between containers in the same Pod.

The biggest risk with ephemeral data is ensuring your Pods don't consume too much of the host's storage capacity. While numbers like 4Gi per Pod might not seem like much, consider a node can run hundreds, in some cases thousands, of Pods. Kubernetes supports the ability to limit the cumulative amount of ephemeral storage available to Pods in a Namespace. Configuration of these concerns are covered in Chapter 12.

Choosing a Storage Provider

There is no shortage of storage providers available to you. Options span from storage solutions you might manage yourself such as Ceph to fully managed systems like Google Persistent Disk or Amazon Elastic Block Store. The variance in options is far beyond the scope of this book. However, we do recommend understanding the capabilities of storage systems along with which of those capabilities are easily integrated with Kubernetes. This will surface perspective on how well one solution may satisfy your application requirements relative to another. Additionally, in the case you may be managing your own storage system, consider using something you have operational experience with when possible. Introducing Kubernetes alongside a new storage system adds a lot of new operational complexity to your organization.

Kubernetes Storage Primitives

Out of the box, Kubernetes provides multiple primitives to support workload storage. These primitives provide the building blocks we will utilize to offer sophisticated storage solutions. In this section, we are going to cover PersistentVolumes, PersistentVolumeClaims, and StorageClasses using an example of allocating fast pre-provisioned storage to containers.

Persistent Volumes and Claims

Volumes and claims live at the foundation of storage in Kubernetes. These are exposed using the PersistentVolume and PersistentVolumeClaim APIs. The PersistentVolume resource represents a storage volume known to Kubernetes. Let's assume an administrator has prepared a node to offer 30Gi of fast, on-host, storage. Let's also assume the administrator has provisioned this storage at */mnt/fast-disk/pod-0*. To

represent this volume in Kubernetes, the administrator can then create a PersistentVolume object:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0
spec:
  capacity:
    storage: 30Gi ❶
  volumeMode: Filesystem ❷
  accessModes:
  - ReadWriteOnce ❸
  storageClassName: local-storage ❹
  local:
    path: /mnt/fast-disk/pod-0
  nodeAffinity: ❺
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - test-w
```

❶ The amount of storage available in this volume. Used to determine whether a claim can bind to this volume.

❷ Specifies whether the volume is a block device or filesystem.

❸ Specifies the access mode of the volume. Includes `ReadWriteOnce`, `ReadMany`, and `ReadWriteMany`.

❹ Associates this volume with a storage class. Used to pair an eventual claim to this volume.

❺ Identifies which node this volume should be associated with.

As you can see, the PersistentVolume contains details around the implementation of the volume. To provide one more layer of abstraction, a PersistentVolumeClaim is introduced, which binds to an appropriate volume based on its request. Most commonly, this will be defined by the application team, added to their Namespace, and referenced from their Pod:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0
spec:
  storageClassName: local-storage ❶
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 30Gi ❷
---
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: fast-disk
      persistentVolumeClaim:
        claimName: pvc0 ❸
  containers:
    - name: ml-processer
      image: ml-processer-image
      volumeMounts:
        - mountPath: "/var/lib/db"
          name: fast-disk
```

❶ Checks for a volume that is of the class `local-storage` with the access mode `ReadWriteOnce`.

❷ Binds to a volume with >= `30Gi` of storage.

❸ Declares this Pod a consumer of the PersistentVolumeClaim.

Based on the PersistentVolume's `nodeAffinity` settings, the Pod will be automatically scheduled on the host where this volume is available. There is no additional affinity configuration required from the developer.

This process has demonstrated a very manual flow for how administrators could make this storage available to developers. We refer to this as static provisioning. With proper automation this could be a viable way to expose fast disk on hosts to Pods. For example, the Local Persistence Volume Static Provisioner can be deployed to the

cluster to detect preallocated storage and expose it, automatically, as PersistentVolumes. It also provides some life cycle management capabilities such as deleting data upon destruction of the PersistentVolumeClaim.

There are multiple ways to achieve local storage that can lead you into a bad practice. For example, it can seem compelling to allow developers to use hostPath rather than needing to preprovision a local storage. `hostPath` enables you to specify a path on the host to bind to rather than having to use a PersistentVolume and PersistentVolumeClaim. This can be a huge security risk as it enables developers to bind to directories on the host, which can have a negative impact on the host and other Pods. If you desire to provide developers ephemeral storage that can withstand a Pod restart but not the Pod being deleted or moved to a different node, you can use EmptyDir. This will allocate storage in the filesystem managed by Kube and be transparent to the Pod.

Storage Classes

In many environments, expecting nodes to be prepared ahead of time with disks and volumes is unrealistic. These cases often warrant dynamic provisioning, where volumes can be made available based on the needs of our claims. To facilitate this model, we can make classes of storage available to our developers. These are defined using the StorageClass API. Assuming your cluster runs in AWS and you want to offer EBS volumes to Pods dynamically, the following StorageClass can be added:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-standard ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: true ❷
provisioner: kubernetes.io/aws-ebs ❸
parameters: ❹
  type: io2
  iopsPerGB: "17"
  fsType: ext4
```

❶ The name of the StorageClass that can be referenced from claims.

❷ Sets this StorageClass as the default. If a claim does not specify a class, this will be used.

❸ Uses the `aws-ebs` provisioner to create the volumes based on claims.

❹ Provider-specific configuration for how to provision volumes.

You can offer a variety of storage options to developers by making multiple StorageClasses available. This includes supporting more than one provider in a single cluster—for example, running Ceph alongside VMware vSAN. Alternatively, you may offer different tiers of storage via the same provider. An example would be offering cheaper storage alongside more expensive options. Unfortunately, Kubernetes lacks granular controls to limit what classes developers can request. Control can be implemented as validating admission control, which is covered in Chapter 8.

Kubernetes offers a wide variety of providers including AWS EBS, Glusterfs, GCE PD, Ceph RBD, and many more. Historically, these providers were implemented in-tree. This means storage providers needed to implement their logic in the core Kubernetes project. This code would then get shipped in the relevant Kubernetes control plane components.

There were several downsides to this model. For one, the storage provider could not be managed out of band. All changes to the provider needed to be tied to a Kubernetes release. Also, every Kubernetes deployment shipped with unnecessary code. For example, clusters running AWS still had the provider code for interacting with GCE PDs. It quickly became apparent there was high value in externalizing these provider integrations and deprecating the in-tree functionality. FlexVolume drivers were an out-of-tree implementation specification that initially aimed to solve this problem. However, FlexVolumes have been put into maintenance mode in favor of our next topic, the Container Storage Interface (CSI).

The Container Storage Interface (CSI)

The Container Storage Interface is the answer to how we provide block and file storage to our workloads. The implementations of CSI are referred to as drivers, which have the operational knowledge for talking to storage providers. These providers span from cloud systems such as Google Persistent Disks to storage systems (such as Ceph) deployed and managed by you. The drivers are implemented by storage providers in projects that live out-of-tree. They can be entirely managed out of band from the cluster they are deployed within.

At a high level, CSI implementations feature a controller plug-in and a node plug-in. CSI driver developers have a lot of flexibility in how they implement these components. Typically, implementations bundle the controller and node plug-ins in the same binary and enable either mode via an environment variable such as `X_CSI_MODE`. The only expectations are that the driver registers with the kubelet and the endpoints in the CSI specification are implemented.

The controller service is responsible for managing the creation and deletion of volumes in the storage provider. This functionality extends into (optional) features such as taking volume snapshots and expanding volumes. The node service is responsible

for preparing volumes to be consumed by Pods on the node. Often this means setting up the mounts and reporting information about volumes on the node. Both the node and controller service also implement identity services that report plug-in info, capabilities, and whether the plug-in is healthy. With this in mind, Figure 4-1 represents a cluster architecture with these components deployed.

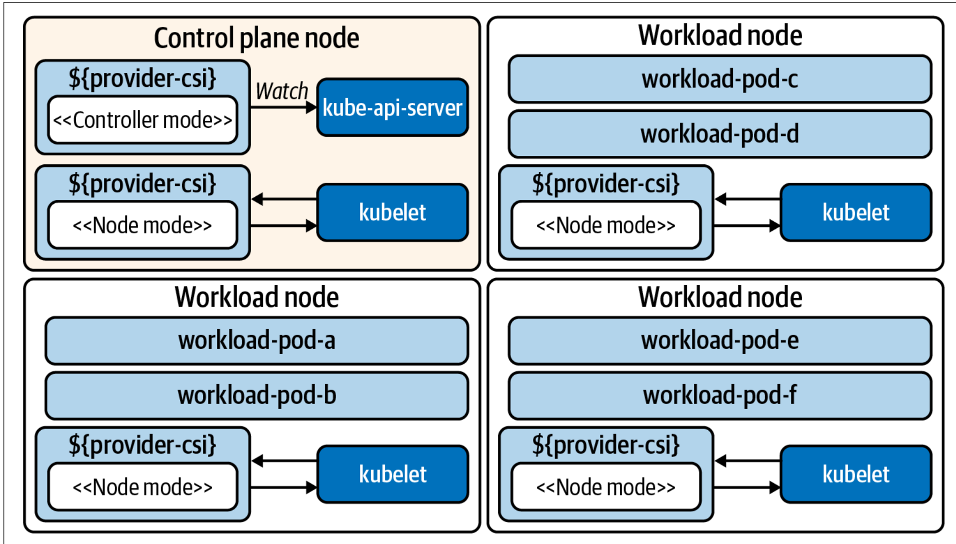


Figure 4-1. Cluster running a CSI plug-in. The driver runs in a node and controller mode. The controller is typically run as a Deployment. The node service is deployed as a DaemonSet, which places a Pod on each host.

Let's take a deeper look at these two components, the controller and the node.

CSI Controller

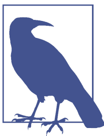
The CSI Controller service provides APIs for managing volumes in a persistent storage system. The Kubernetes control plane *does not* interact with the CSI Controller service directly. Instead, controllers maintained by the Kubernetes storage community react to Kubernetes events and translate them into CSI instructions, such as CreateVolumeRequest when a new PersistentVolumeClaim is created. Because the CSI Controller service exposes its APIs over UNIX sockets, the controllers are usually deployed as sidecars alongside the CSI Controller service. There are multiple external controllers, each with different behavior:

external-provisioner
: When PersistentVolumeClaims are created, this requests a volume be created from the CSI driver. Once the volume is created in the storage provider, this provisioner creates a PersistentVolume object in Kubernetes.

external-attacher
: Watches the VolumeAttachment objects, which declare that a volume should be attached or detached from a node. Sends the attach or detach request to the CSI driver.

external-resizer
: Detects storage-size changes in PersistentVolumeClaims. Sends requests for expansion to the CSI driver.

external-snapshotter
: When VolumeSnapshotContent objects are created, snapshot requests are sent to the driver.



When implementing CSI plug-ins, developers are not required to use the aforementioned controllers. However, their use is encouraged to prevent duplication of logic in every CSI plug-in.

CSI Node

The Node plug-in typically runs the same driver code as the controller plug-in. However, running in the "node mode" means it is focused on tasks such as mounting attached volumes, establishing their filesystem, and mounting volumes to Pods. Requests for these behaviors is done via the kubelet. Along with the driver, the following sidecars are often included in the Pod:

node-driver-registrar
: Sends a registration request to the kubelet to make it aware of the CSI driver.

liveness-probe
: Reports the health of the CSI driver.

Implementing Storage as a Service

We have now covered key considerations for application storage, storage primitives available in Kubernetes, and driver integration using the CSI. Now it's time to bring these ideas together and look at an implementation that offers developers storage as a service. We want to provide a declarative way to request storage and make it available to workloads. We also prefer to do this dynamically, not requiring an administrator to preprovision and attach volumes. Rather, we'd like to achieve this on demand based on the needs of workloads.

In order to get started with this implementation, we'll use Amazon Web Services (AWS). This example integrates with AWS's elastic block storage system. If your

choice in provider differs, the majority of this content will still be relevant! We are simply using this provider as a concrete example of how all the pieces fit together.

Next we are going to dive into installation of the integration/driver, exposing storage options to developers, consuming the storage with workloads, resizing volumes, and taking volume snapshots.

Installation

Installation is a fairly straightforward process consisting of two key steps:

1. Configure access to the provider.
2. Deploy the driver components to the cluster.

The provider, in this case AWS, will require the driver to identify itself, ensuring it has appropriate access. In this case, we have three options available to us. One is to update the instance profile of the Kubernetes nodes. This will prevent us from worrying about credentials at the Kubernetes level but will provide universal privileges to workloads that can reach the AWS API. The second and likely most secure option is to introduce an identity service that can provide IAM permissions to specific workloads. A project that is an example of this is kiam. This approach is covered in Chapter 10. Lastly, you can add credentials in a secret that gets mounted into the CSI driver. In this model, the secret would look as follows:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: aws-secret
  namespace: kube-system
stringData:
  key_id: "AKIAWJQHICPELCJVKYNU"
  access_key: "jqWi1ut4KyrAHADIOrhH2Pd/vXpgqA9OZ3bCZ"
```

This account will have access to manipulating an underlying storage system. Access to this secret should be carefully managed. See Chapter 7 for more information.

With this configuration in place, the CSI components may be installed. First, the controller is installed as a Deployment. When running multiple replicas, it will use leader-election to determine which instance should be active. Then, the node plug-in is installed, which comes in the form of a DaemonSet running a Pod on every node. Once initialized, the instances of the node plug-in will register with their kubelets. The kubelet will then report the CSI-enabled node by creating a CSINode object for every Kubernetes node. The output of a three-node cluster is as follows:

```
$ kubectl get csinode

NAME                                        DRIVERS   AGE
ip-10-0-0-205.us-west-2.compute.internal   1         97m
ip-10-0-0-224.us-west-2.compute.internal   1         79m
ip-10-0-0-236.us-west-2.compute.internal   1         98m
```

As we can see, there are three nodes listed with one driver registered on each node. Examining the YAML of one CSINode exposes the following:

```
apiVersion: storage.k8s.io/v1
kind: CSINode
metadata:
  name: ip-10-0-0-205.us-west-2.compute.internal
spec:
  drivers:
    - allocatable:
        count: 25 ❶
      name: ebs.csi.aws.com
      nodeID: i-0284ac0df4da1d584
      topologyKeys:
        - topology.ebs.csi.aws.com/zone ❷
```

❶ The maximum number of volumes allowed on this node.

❷ When a node is picked for a workload, this value will be passed in the CreateVolumeRequest so that the driver knows *where* to create the volume. This is important for storage systems where nodes in the cluster won't have access to the same storage. For example, in AWS, when a Pod is scheduled in an availability zone, the Volume must be created in the same zone.

Additionally, the driver is officially registered with the cluster. The details can be found in the CSIDriver object:

```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: aws-ebs-csi-driver ❶
  labels:
    app.kubernetes.io/name: aws-ebs-csi-driver
spec:
  attachRequired: true ❷
  podInfoOnMount: false ❸
  volumeLifecycleModes:
  - Persistent ❹
```

❶ The name of the provider representing this driver. This name will be bound to class(es) of storage we offer to platform users.

❷ Specifies that an attach operation must be completed before volumes are mounted.

❸ Does not need to pass Pod metadata in as context when setting up a mount.

❹ The default model for provisioning persistent volumes. Inline support can be enabled by setting this option to `Ephemeral`. In the ephemeral mode, the storage is expected to last only as long as the Pod.

The settings and objects we have explored so far are artifacts of our bootstrapping process. The CSIDriver object makes for easier discovery of driver details and was included in the driver's deployment bundle. The CSINode objects are managed by the kubelet. A generic registrar sidecar is included in the node plug-in Pod and gets details from the CSI driver and registers the driver with the kubelet. The kubelet then reports up the quantity of CSI drivers available on each host. Figure 4-2 demonstrates this bootstrapping process.

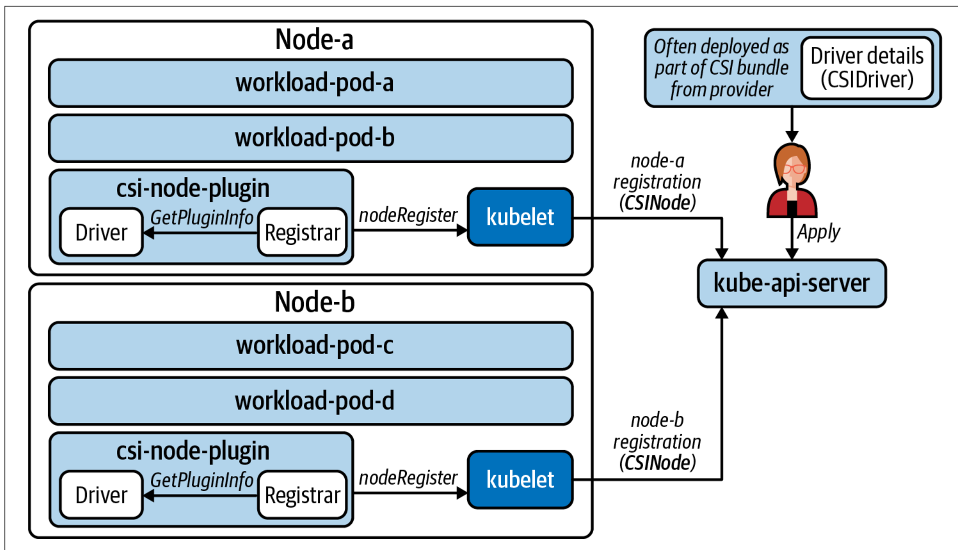


Figure 4-2. CSIDriver object is deployed and part of the bundle while the node plug-in registers with the kubelet. This in turn creates/manages the CSINode objects.

Exposing Storage Options

In order to provide storage options to developers, we need to create StorageClasses. For this scenario we'll assume there are two types of storage we'd like to expose.The first option is to expose cheap disk that can be used for workload persistence needs. Many times, applications don't need an SSD as they are just persisting some files that do not require quick read/write. As such, the cheap disk (HDD) will be the default

option. Then we'd like to offer faster SSD with a custom IOPS per gigabyte configured. Table 4-1 shows our offerings; prices reflect AWS costs at the time of this writing.

Table 4-1. Storage offerings

Offering name	Storage type	Max throughput per volume	AWS cost
default-block	HDD (optimized)	40–90 MB/s	$0.045 per GB per month
performance-block	SSD (io1)	~1000 MB/s	$0.125 per GB per month + $0.065 per provisioned IOPS per month

In order to create these offerings, we'll create a storage class for each. Inside each storage class is a `parameters` field. This is where we can configure settings that satisfy the features in Table 4-1.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: default-block ❶
  annotations:
    storageclass.kubernetes.io/is-default-class: "true" ❷
provisioner: ebs.csi.aws.com ❸
allowVolumeExpansion: true ❹
volumeBindingMode: WaitForFirstConsumer ❺
parameters:
  type: st1 ❻
---
kind: StorageClass ❼
apiVersion: storage.k8s.io/v1
metadata:
  name: performance-block
provisioner: ebs.csi.aws.com
parameters:
  type: io1
  iopsPerGB: "20"
```

❶ This is the name of the storage offering we are providing to platform users. It will be referenced from PeristentVolumeClaims.

❷ This sets the offering as the default. If a PersistentVolumeClaim is created *without* a StorageClass specified, `default-block` will be used.

❸ Mapping to which CSI driver should be executed.

❹ Allow expansion of the volume size via changes to a PersistentVolumeClaim.

❺ Do not provision the volume until a Pod consumes the PersistentVolumeClaim. This will ensure the volume is created in the appropriate availability zone of the

scheduled Pod. It also prevents orphaned PVCs from creating volumes in AWS, which you will be billed for.

❻ Specifies what type of storage the driver should acquire to satisfy claims.

❼ Second class, tuned to high-performance SSD.

Consuming Storage

With the preceding pieces in place, we are now ready for users to consume these different classes of storage. We will start by looking at the developer experience of requesting storage. Then we'll walk through the internals of how it is satisfied. To start off, let's see what a developer gets when listing available StorageClasses:

```
$ kubectl get storageclasses.storage.k8s.io

NAME                     PROVISIONER       RECLAIMPOLICY   VOLUMEBINDINGMODE
default-block (default)  ebs.csi.aws.com   Delete          Immediate
performance-block        ebs.csi.aws.com   Delete          WaitForFirstConsumer

ALLOWVOLUMEEXPANSION
true
true
```

By enabling developers to create PVCs, we will be allowing them to reference *any* StorageClass. If this is problematic, you may wish to consider implementing Validating Admission control to assess whether requests are appropriate. This topic is covered in Chapter 8.

Let's assume the developer wants to make a cheaper HDD and more performant SSD available for an application. In this case, two PersistentVolumeClaims are created. We'll refer to these as `pvc0` and `pvc1`, respectively:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0 ❶
spec:
  resources:
    requests:
      storage: 11Gi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
spec:
  resources:
```

```
      requests:
        storage: 14Gi
    storageClassName: performance-block ②
```

❶ This will use the default storage class (`default-block`) and assume other defaults such as RWO and filesystem storage type.

❷ Ensure `performance-block` is requested to the driver rather than `default-block`.

Based on the StorageClass settings, these two will exhibit different provisioning behaviors. The performant storage (from `pvc1`) is created as an unattached volume in AWS. This volume can be attached quickly and is ready to use. The default storage (from `pv0`) will sit in a `Pending` state where the cluster waits until a Pod consumes the PVC to provision storage in AWS. While this will require more work to provision when a Pod finally consumes the claim, you will not be billed for the unused storage! The relationship between the claim in Kubernetes and volume in AWS can be seen in Figure 4-3.

Create Volume   Actions

Filter by tags and attributes or search by keyword

CSIVolumeName	Volume ID	Size	Volume Type
pvc-123d0302-d2fc-46f0-b00d-48716358b567	vol-0bbe36e93a2422c77	4 GiB	gp3
	vol-07d0256950216ca6b	80 GiB	gp2

```
$ k get pv,pvc
NAME                                                        CAPACITY   ACCESS MODES
persistentvolume/pvc-123d0302-d2fc-46f0-b00d-48716358b567   4Gi        RWO

NAME                                 STATUS   VOLUME
persistentvolumeclaim/ebs-claim      Bound    pvc-123d0302-d2fc-46f0-b00d-48716358b567
```

Figure 4-3. `pv1` is provisioned as a volume in AWS, and the CSIVolumeName is propagated for ease of correlation. `pv0` will not have a respective volume created until a Pod references it.

Now let's assume the developer creates two Pods. One Pod references `pv0` while the other references `pv1`. Once each Pod is scheduled on a Node, the volume will be attached to that node for consumption. For `pv0`, before this can occur the volume will also be created in AWS. With the Pods scheduled and volumes attached, a filesystem is established and the storage is mounted into the container. Because these are persistent volumes, we have now introduced a model where even if the Pod is rescheduled to another node, the volume can come with it. The end-to-end flow for how we've satisfied the self-service storage request is shown in Figure 4-4.

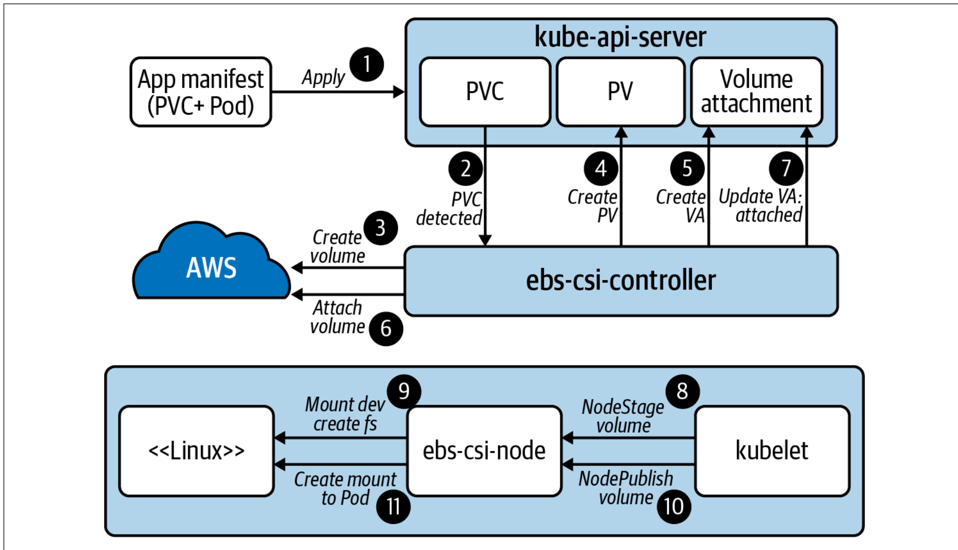
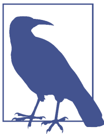


Figure 4-4. End-to-end flow of the driver and Kubernetes working together to satisfy the storage request.



Events are particularly helpful in debugging storage interaction with CSI. Because provisioning, attaching, and mounting are all happening in order to satisfy a PVC, you should view events on these objects as different components report what they have done. `kubectl describe -n $NAMESPACE pvc $PVC_NAME` is an easy way to view these events.

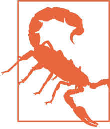
Resizing

Resizing is a supported feature in the `aws-ebs-csi-driver`. In most CSI implementations, the `external-resizer` controller is used to detect changes in PersistentVolumeClaim objects. When a size change is detected, it is forwarded to the driver, which will expand the volume. In this case, the driver running in the controller plug-in will facilitate expansion with the AWS EBS API.

Once the volume is expanded in EBS, the new space is *not* immediately usable to the container. This is because the filesystem still occupies only the original space. In order for the filesystem to expand, we'll need to wait for the node plug-in's driver instance to expand the filesystem. This can all be done *without* terminating the Pod. The filesystem expansion can be seen in the following logs from the node plug-in's CSI driver:

```
mount_linux.go: Attempting to determine if disk "/dev/nvme1n1" is formatted
using blkid with args: ([-p -s TYPE -s PTTYPE -o export /dev/nvme1n1])
```

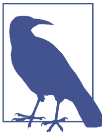
```
mount_linux.go: Output: "DEVNAME=/dev/nvme1n1\nTYPE=ext4\n", err: <nil>

resizefs_linux.go: ResizeFS.Resize - Expanding mounted volume /dev/nvme1n1

resizefs_linux.go: Device /dev/nvme1n1 resized successfully
```



Kubernetes does not support downsizing a PVC's size field. Unless the CSI-driver provides a workaround for this, you may not be able to downsize without re-creating a volume. Keep this in mind when growing volumes.

Snapshots

To facilitate periodic backups of volume data used by containers, snapshot functionality is available. The functionality is often broken into two controllers, which are responsible for two different CRDs. The CRDs include VolumeSnapshot and VolumeContentSnapshot. At a high-level, the VolumeSnapshot is responsible for the life cycle of volumes. Based on these objects, VolumeContentSnapshots are managed by the external-snapshotter controller. This controller is typically run as a sidecar in the CSI's controller plug-in and forwards requests to the driver.



At the time of this writing, these objects are implemented as CRDs and not core Kubernetes API objects. This requires the CSI driver or Kubernetes distribution to deploy the CRD definitions ahead of time.

Similar to offering storage via StorageClasses, snapshotting is offered by introducing a Snapshot class. The following YAML represents this class:

```
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: default-snapshots
driver: ebs.csi.aws.com ❶
deletionPolicy: Delete ❷
```

❶ Which driver to delegate snapshot request to.

❷ Whether the VolumeSnapshotContent should be deleted when the VolumeSnapshot is deleted. In effect, the actual volume could be deleted (depending on support from the provider).

In the Namespace of the application and PersistentVolumeClaim, a VolumeSnapshot may be created. An example is as follows:

```yaml
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshot
metadata:
  name: snap1
spec:
  volumeSnapshotClassName: default-snapshots ❶
  source:
    persistentVolumeClaimName: pvc0 ❷
```

❶ Specifies the class, which informs the driver to use.

❷ Specifies the volume claim, which informs the volume to snapshot.

The existence of this object will inform the need to create a VolumeSnapshotContent object. This object has a scope of cluster-wide. The detection of a VolumeSnapshot-Content object will cause a request to create a snapshot and the driver will satisfy this by communicating with AWS EBS. Once satisfied, the VolumeSnapshot will report ReadyToUse. Figure 4-5 demonstrates the relationship between the various objects.

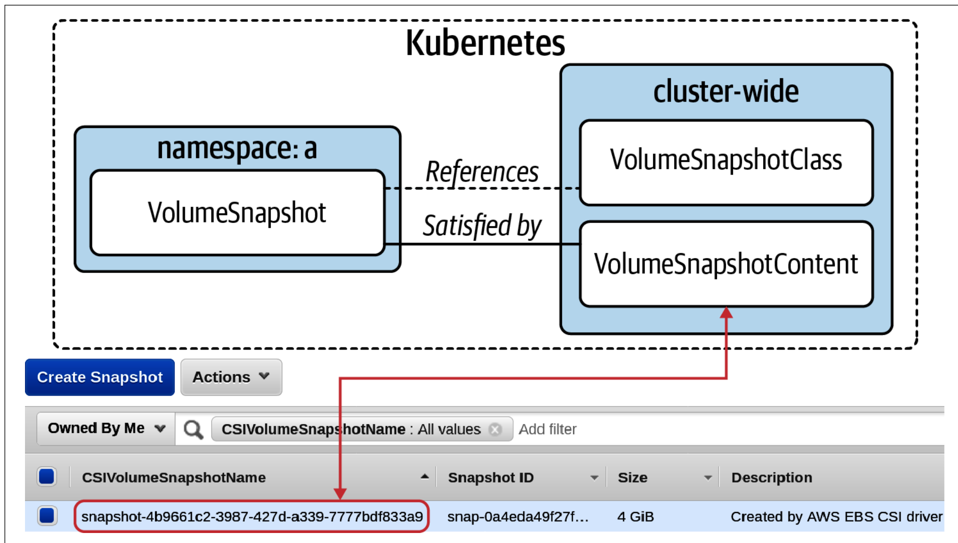


Figure 4-5. The various objects and their relations that make up the snapshot flow.

With a snapshot in place, we can explore a scenario of data loss. Whether the original volume was accidentally deleted, had a failure, or was removed due to an accidental deletion of a PersistentVolumeClaim, we can reestablish the data. To do this, a new PersistentVolumeClaim is created with the `spec.dataSource` specified. `dataSource` supports referencing a VolumeSnapshot that can populate data into the new claim. The following manifest recovers from the previously created snapshot:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-reclaim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: default-block
  resources:
    requests:
      storage: 600Gi
  dataSource:
    name: snap1 ❶
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
```

❶ The VolumeSnapshot instance that references the EBS snapshot to replenish the new PVC.

Once the Pod is re-created to reference this new claim, the last snapshotted state will return to the container! Now we have access to all the primitives for creating a robust backup and recovery solution. Solutions could range from scheduling snapshots via a CronJob, writing a custom controller, or using tools such as Velero to back up Kubernetes objects along with data volumes on a schedule.

Summary

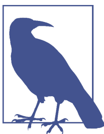
In this chapter, we've explored a variety of container storage topics. First, we want to have a deep understanding of application requirements to best inform our technical decision. Then we want to ensure that our underlying storage provider can satisfy these needs and that we have the operational expertise (when required) to operate them. Lastly, we should establish an integration between the orchestrator and the storage system, ensuring developers can get the storage they need without being proficient in an underlying storage system.

CHAPTER 5
Pod Networking

Since the early days of networking, we have concerned ourselves with how to facilitate host-to-host communication. These concerns include uniquely addressing hosts, routing of packets across networks, and propagation of known routes. For more than a decade, software-defined networks (SDNs) have seen rapid growth by solving these concerns in our increasingly dynamic environments. Whether it is in your datacenter with VMware NSX or in the cloud with Amazon VPCs, you are likely a consumer of an SDN.

In Kubernetes, these principles and desires hold. Although our unit moves from hosts to Pods, we need to ensure we have addressability and routability of our workloads. Additionally, given Pods are running as software on our hosts, we will most commonly establish networks that are entirely software-defined.

This chapter will explore the concept of Pod networks. We will start off by addressing some key networking concepts that must be understood and considered before implementing Pod networks. Then we will cover the Container Networking Interface (CNI), which enables your choice of network implementation based on your networking requirements. Lastly, we will examine common plug-ins, such as Calico and Cilium, in the ecosystem to make the trade-offs more concrete. In the end, you'll be more equipped to make decisions around the right networking solution and configuration for your application platform.



Networking is a vast subject in itself. Our intentions are to give you just enough to make informed decisions on your Pod network. If your background is not networking, we highly recommend you go over these concepts with your networking team. Kubernetes does not negate the need to have networking expertise in your organization!

Networking Considerations

Before diving into implementation details around Pod networks, we should start with a few key areas of consideration. These areas include:

- IP Address Management (IPAM)
- Routing protocols
- Encapsulation and tunneling
- Workload routability
- IPv4 and IPv6
- Encrypted workload traffic
- Network policy

With an understanding of these areas, you can begin to make determinations around the correct networking solution for your platform.

IP Address Management

In order to communicate to and from Pods, we must ensure they are uniquely addressable. In Kubernetes, each Pod receives an IP. These IPs may be internal to the cluster or externally routable. Each Pod having its own address simplifies the networking model, considering we do not have to be concerned with colliding ports on shared IPs. However, this IP-per-Pod model does come with its own challenges.

Pods are best thought of as ephemeral. Specifically, they are prone to being restarted or rescheduled based on the needs of the cluster or system failure. This requires IP allocation to execute quickly and the management of the cluster's IP pool to be efficient. This management is often referred to as IPAM and is not unique to Kubernetes. As we dive deeper into container networking approaches, we will explore a variety of ways IPAM is implemented.

This ephemeral expectation of a workload's IP causes issues in some legacy workloads, for example, workloads that pin themselves to a specific IP and expect it to remain static. Depending on your implementation of container networking (covered later in this chapter), you may be able to explicitly reserve IPs for specific workloads. However, we recommend against this model unless necessary. There are many capable service discovery or DNS mechanisms that workloads can take advantage of to properly remedy this issue. Review Chapter 6 for examples.

IPAM is implemented based on your choice of CNI plug-in. There are a few commonalities in these plug-ins that pertain to Pod IPAM. First, when clusters are created, a Pod network's Classless Inter-Domain Routing (CIDR) can be specified. How it is set varies based on how you bootstrap Kubernetes. In the case of `kubeadm`, a flag can be passed as follows:

```
kubeadm init --pod-network-cidr 10.30.0.0/16
```

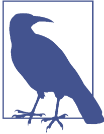
In effect, this sets the `--cluster-cidr` flag on the kube-controller-manager. Kubernetes will then allocate a chunk of this cluster-cidr to every node. By default, each node is allocated `/24`. However, this can be controlled by setting the `--node-cidr-mask-size-ipv4` and/or `--node-cidr-mask-size-ipv6` flags on the kube-controller-manager. A Node object featuring this allocation is as follows:

```yaml
apiVersion: v1
kind: Node
metadata:
  labels:
    kubernetes.io/arch: amd64
    kubernetes.io/hostname: test
    kubernetes.io/os: linux
    manager: kubeadm
  name: master-0
spec:
  podCIDR: 10.30.0.0/24 ❶
  podCIDRs:
  - 10.30.0.0/24 ❷
```

❶ This field exists for compatibility. `podCIDRs` was later introduced as an array to support dual stack (IPv4 and IPv6 CIDRs) on a single node.

❷ The IP range assigned to this node is 10.30.0.0 - 10.30.0.255. This is 254 addresses for Pods, out of the 65,534 available in the 10.30.0.0/16 cluster CIDR.

Whether these values are used in IPAM is up to the CNI plug-in. For example, Calico detects and respects this setting, while Cilium offers an option to either manage IP pools independent of Kubernetes (default) or respect these allocations. In most CNI implementations, it is important that your CIDR choice *does not* overlap with the cluster's host/node network. However, assuming your Pod network will remain internal to the cluster, the CIDR chosen can overlap with network space outside the cluster. Figure 5-1 demonstrates the relationship of these various IP spaces and examples of allocations.



How large you should set your cluster's Pod CIDR is often a product of your networking model. In most deployments, a Pod network is entirely internal to the cluster. As such, the Pod CIDR can be very large to accommodate for future scale. When the Pod CIDR is routable to the larger network, thus consuming address space, you may have to do more careful consideration. Multiplying the number of Pods per node by your eventual node count can give you a rough estimate. The number of Pods per node is configurable on the kubelet, but by default is 110.

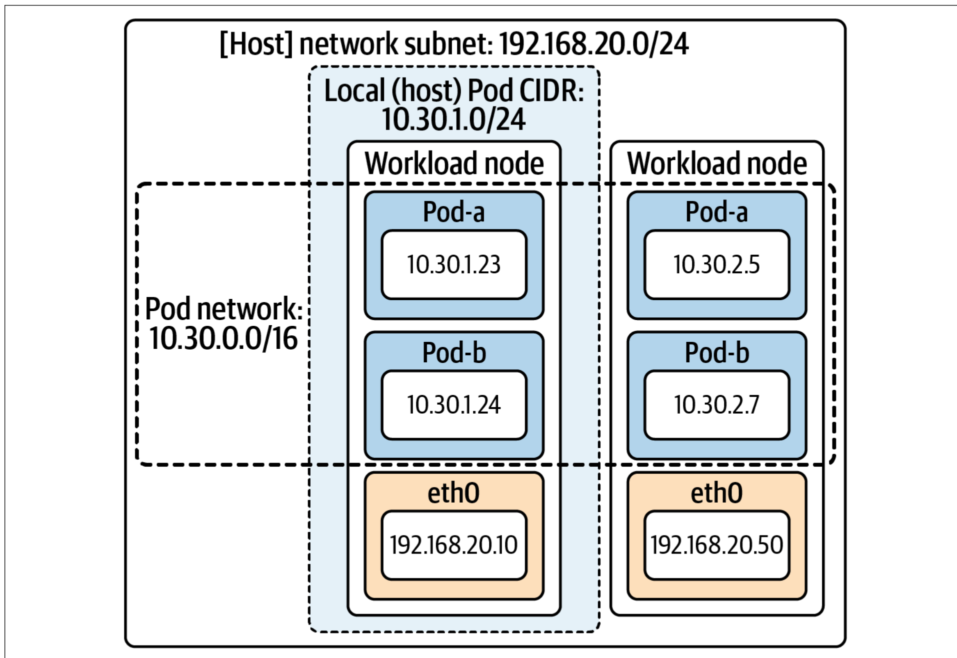


Figure 5-1. The IP spaces and IP allocations of the host network, Pod network, and each [host] local CIDR.

Routing Protocols

Once Pods are addressed, we need to ensure that routes to and from them are understood. This is where routing protocols come in to play. Routing protocols can be thought of as different ways to propagate routes to and from places. Introducing a routing protocol often enables dynamic routing, relative to configuring static routes. In Kubernetes, understanding a multitude of routes becomes important when not leveraging encapsulation (covered in the next section), since the network will often be unaware of how to route workload IPs.

Border Gateway Protocol (BGP) is one of the most commonly used protocols to distribute workload routes. It is used in projects such as Calico and Kube-Router. Not only does BGP enable communication of workload routes in the cluster but its internal routers can also be peered with external routers. Doing so can make external network fabrics aware of how to route to Pod IPs. In implementations such as Calico, a BGP daemon is run as part of the Calico Pod. This Pod runs on every host. As routes to workloads become known, the Calico Pod modifies the kernel routing table to include routes to each potential workload. This provides native routing via the workload IP, which can work especially well when running in the same L2 segment. Figure 5-2 demonstrates this behavior.

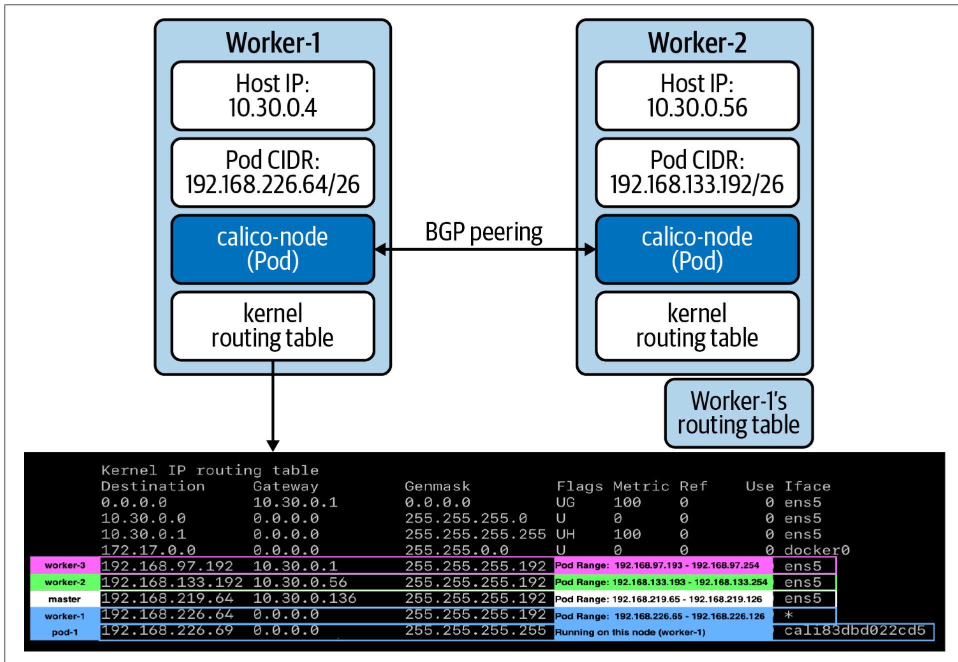


Figure 5-2. The `calico-pod` sharing routes via its BGP peer. The kernel routing table is then programmed accordingly.

Making Pod IPs routable to larger networks may seem appealing at first glance but should be carefully considered. See "Workload Routability" on page 108 for more details.

In many environments, native routing to workload IPs is not possible. Additionally, routing protocols such as BGP may not be able to integrate with an underlying network; such is the case running in a cloud-provider's network. For example, let's consider a CNI deployment where we wish to support native routing and share routes via BGP. In an AWS environment, this can fail for two reasons:

Source/Destination checks are enabled
: This ensures that packets hitting the host have the destination (and source IP) of the target host. If it does not match, the packet is dropped. This setting can be disabled.

Packet needs to traverse subnets
: If the packet needs to leave the subnet, the destination IP is evaluated by the underlying AWS routers. When the Pod IP is present, it will not be able to route.

In these scenarios, we look to tunneling protocols.

Encapsulation and Tunneling

Tunneling protocols give you the ability to run your Pod network in a way that is mostly unknown to the underlying network. This is achieved using encapsulation. As the name implies, encapsulation involves putting a packet (the inner packet) inside another packet (the outer packet). The inner packet's src IP and dst IP fields will reference the workload (Pod) IPs, whereas the outer packet's src IP and dst IP fields will reference the host/node IPs. When the packet leaves a node, it will appear to the underlying network as any other packet since the workload-specific data is in the payload. There are a variety of tunneling protocols such as VXLAN, Geneve, and GRE. In Kubernetes, VXLAN has become one of the most commonly used methods by networking plug-ins. Figure 5-3 demonstrates an encapsulated packet crossing the wire via VXLAN.

As you can see, VXLAN puts an entire Ethernet frame inside a UDP packet. This essentially gives you a fully virtualized layer-2 network, often referred to as an overlay network. The network beneath the overlay, referred to as the underlay network, does not concern itself with the overlay. This is one of the primary benefits to tunneling protocols.

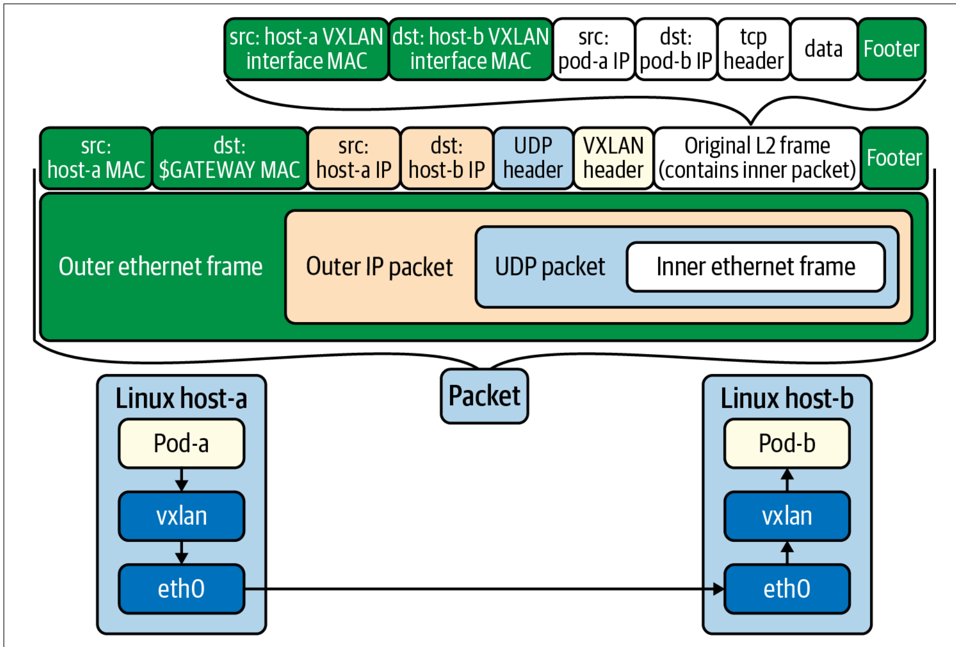


Figure 5-3. VXLAN encapsulation used to move an inner packet, for workloads, across hosts. The network cares only about the outer packet, so it needs to have zero awareness of workload IPs and their routes.

Often, you choose whether to use a tunneling protocol based on the requirements/capabilities of your environment. Encapsulation has the benefit of working in many scenarios since the overlay is abstracted from the underlay network. However, this approach comes with a few key downsides:

Traffic can be harder to understand and troubleshoot
: Packets within packets can create extra complexity when troubleshooting network issues.

Encapsulation/decapsulation will incur processing cost
: When a packet goes to leave a host it must be encapsulated, and when it enters a host it must be decapsulated. While likely small, this will add overhead relative to native routing.

Packets will be larger
: Due to the embedding of packets, they will be larger when transmitted over the wire. This may require adjustments to the maximum transmission unit (MTU) to ensure they fit on the network.

Workload Routability

In most clusters, Pod networks are internal to the cluster. This means Pods can directly communicate with each other, but external clients cannot reach Pod IPs directly. Considering Pod IPs are ephemeral, communicating directly with a Pod's IP is often bad practice. Relying on service discovery or load balancing mechanics that abstract the underlying IP is preferable. A huge benefit to the internal Pod network is that it *does not* occupy precious address space within your organization. Many organizations manage address space to ensure addresses stay unique within the company. Thus, you would certainly get a dirty look when you ask for a `/16` space (65,536 IPs) for each Kubernetes cluster you bootstrap!

When Pods are not directly routable, we have several patterns to facilitate external traffic to Pod IPs. Commonly we will expose an Ingress controller on the host network of a subset of dedicated nodes. Then, once the packet enters the Ingress controller proxy, it can route directly to Pod IPs since it takes part in the Pod network. Some cloud providers even include (external) load balancer integration that wires this all together automatically. We explore a variety of these ingress models, and their trade-offs, in Chapter 6.

At times, requirements necessitate that Pods are routable to the larger network. There are two primary means to accomplish this. The first is to use a networking plug-in that integrates with the underlying network directly. For example, AWS's VPC CNI attaches multiple secondary IPs to each node and allocates them to Pods. This makes each Pod routable just as an EC2 host would be. The primary downside to this model is it will consume IPs in your subnet/VPC. The second option is to propagate routes to Pods via a routing protocol such as BGP, as described in "Routing Protocols" on page 104. Some plug-ins using BGP will even enable you to make a subset of your Pod network routable, rather than having to expose the entire IP space.

Avoid making your Pod network externally routable unless absolutely necessary. We often see legacy applications driving the desire for routable Pods. For example, consider a TCP-based workload where a client must be pinned to the same backend. Typically, we recommend updating the application(s) to fit within the container networking paradigm using service discovery and possibly re-architecting the backend to not require client-server affinity (when possible). While exposing the Pod networking may seem like a simple solution, doing so comes at the cost of eating up IP space and potentially making IPAM and route propagation configurations more complex.

IPv4 and IPv6

The overwhelming majority of clusters today run IPv4 exclusively. However, we are seeing the desire to run IPv6-networked clusters in certain clients such as telcos where addressability of many workloads is critical. Kubernetes does support IPv6 via dual-stack as of `1.16`. At the time of this writing, dual-stack is an alpha feature. Dual-stack enables you to configure IPv4 and IPv6 address spaces in your clusters.

If your use case requires IPv6, it can easily be enabled but requires a few components to line up:

- While still in alpha, a feature-gate must be enabled on the kube-apiserver and kubelet.
- The kube-apiserver, kube-controller-manager, and kube-proxy all require an additional configuration to specify the IPv4 and IPv6 space.
- You must use a CNI plug-in that supports IPv6, such as Calico or Cilium.

With the preceding in place, you will see two CIDR allocations on each Node object:

```
spec:
  podCIDR: 10.30.0.0/24
  podCIDRs:
  - 10.30.0.0/24
  - 2002:1:1::/96
```

The CNI plug-in's IPAM is responsible for determining whether an IPv4, IPv6, or both is assigned to each Pod.

Encrypted Workload Traffic

Pod-to-Pod traffic is rarely (if ever) encrypted by default. This means that packets sent over the wire without encryption, such as TLS, can be sniffed as plain text. Many network plug-ins support encrypting traffic over the wire. For example, Antrea supports encryption with IPsec when using a GRE tunnel. Calico is able to encrypt traffic by tapping into a node's WireGuard installation.

Enabling encryption may seem like a no-brainer. However, there are trade-offs to be considered. We recommend talking with your networking team to understand how host-to-host traffic is handled today. Is data encrypted when it goes between hosts in your datacenter? Additionally, what other encryption mechanisms may be at play? For example, does every service talk over TLS? Do you plan to leverage a service mesh where workload proxies leverage mTLS? If so, is encrypting at the service proxy and CNI layer required? While encryption will increase the depth of defense, it will also add complexity to network management and troubleshooting. Most importantly, needing to encrypt and decrypt packets will impact performance, thus lowering your potential throughput.

Network Policy

Once the Pod network is wired up, a logical next step is to consider how to set up network policy. Network policy is similar to firewall rules or security groups, where we can define what ingress and egress traffic is allowed. Kubernetes offers a NetworkPolicy API, as part of the core networking APIs. Any cluster can have policies added to it. However, it is incumbent on the CNI provider to *implement* the policy. This means that a cluster running a CNI provider that does not support NetworkPolicy, such as flannel, will accept NetworkPolicy objects but not act on them. Today, most CNIs have some level of support for NetworkPolicy. Those that do not can often be used alongside plug-ins such as Calico, where the plug-in runs in a mode where it provides only policy enforcement.

NetworkPolicy being available inside of Kubernetes adds yet another layer where firewall-style rules can be managed. For example, many networks provide subnet or host-level rules available via a distributed firewall or security group mechanism. While good, often these existing solutions do not have visibility into the Pod network. This prevents the level of granularity that may be desired in setting up rules for Pod-based workload communication. Another compelling aspect of Kubernetes NetworkPolicy is that, like most objects we deal with in Kubernetes, it is defined declaratively and, we think, far easier to manage relative to most firewall management solutions! For these reasons, we generally recommend considering implementing network policy at the Kubernetes level rather than trying to make existing firewall solutions fit this new paradigm. This does not mean you should throw out your existing host-to-host firewall solution(s). More so, let Kubernetes handle the intra-workload policy.

Should you choose to utilize NetworkPolicy, it is important to note these policies are *Namespace-scoped*. By default, when NetworkPolicy objects are not present, Kubernetes allows all communication to and from workloads. When setting a policy, you can select what workloads the policy applies to. When present, the default behavior inverts and any egress and ingress traffic not allowed by the policy will be blocked. This means that the Kubernetes NetworkPolicy API specifies only what traffic is allowed. Additionally, the policy in a Namespace is additive. Consider the following NetworkPolicy object that configures ingress and egress rules:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: team-netpol
  namespace: org-1
spec:
  podSelector: {} ❶
  policyTypes:
  - Ingress
  - Egress
  ingress: ❷
```

```
  - from:
    - ipBlock:
        cidr: 10.40.0.0/24
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to: ❸
    ports:
    - protocol: UDP
      port: 53
  - to: ❹
    - namespaceSelector:
        matchLabels:
          name: org-2
    - podSelector:
        matchLabels:
          app: team-b
    ports:
    - protocol: TCP
      port: 80
```

❶ The empty `podSelector` implies this policy applies to all Pods in this Namespace. Alternatively, you can match against a label.

❷ This ingress rule allows traffic from sources with an IP in the range of 10.40.0.0/24, when the protocol is TCP and the port is 80.

❸ This egress rule allows DNS traffic from workloads.

❹ This egress rule limits sending traffic to packets destined for workloads in the `org-2` Namespace with the label `team-b`. Additionally, the protocol must be TCP and the destination port is 80.

Over time, we have seen the NetworkPolicy API be limiting to certain use cases. Some common desires include:

- Complex condition evaluation
- Resolution of IPs based on DNS records
- L7 rules (host, path, etc.)
- Cluster-wide policy, enabling global rules to be put in place, rather than having to replicate them in every Namespace.

To satisfy these desires, some CNI plug-ins offer their own, more capable, policy APIs. The primary trade-off to using provider-specific APIs is that your rules are no longer portable across plug-ins. We will explore examples of these when we cover Calico and Cilium later in the chapter.

Summary: Networking Considerations

In the previous sections we have covered key networking considerations that will enable you to make an informed decision about your Pod networking strategy. Before diving into CNI and plug-ins, let's recap some of the key areas of consideration:

- How large should your Pod CIDR be per cluster?
- What networking constraints does your underlay network put on your future Pod network?
- If using a Kubernetes managed service or vendor offering, what networking plug-in(s) are supported?
- Are routing protocols such as BGP supported in your infrastructure?
- Could unencapsulated (native) packets be routed through the network?
- Is using a tunnel protocol (encapsulation) desirable or required?
- Do you need to support (externally) routable Pods?
- Is running IPv6 a requirement for your workloads?
- On what level(s) will you expect to enforce network policy or firewall rules?
- Does your Pod network need to encrypt traffic on the wire?

With the answers to these questions fleshed out, you are in a good place to start learning about what enables you to plug in the correct technology to solve these issues, the Container Networking Interface (CNI).

The Container Networking Interface (CNI)

All the considerations discussed thus far make it clear that different use cases warrant different container networking solutions. In the early days of Kubernetes, most clusters were running a networking plug-in called flannel. Over time, solutions such as Calico and others gained popularity. These new plug-ins brought different approaches to creating and running networks. This drove the creation of a standard for how systems such as Kubernetes could request networking resources for its workloads. This standard is known as the Container Networking Interface (CNI). Today, all networking options compatible with Kubernetes conform to this interface. Similar to the Container Storage Interface (CSI) and Container Runtime Interface (CRI), this gives us flexibility in the networking stack of our application platform.

The CNI specification defines a few key operations:

`ADD`
: Adds a container to the network and responds with the associated interface(s), IP(s), and more.

`DELETE`
: Removes a container from the network and releases all associated resources.

`CHECK`
: Verifies a container's network is set up properly and responds with an error if there are issues.

`VERSION`
: Returns the CNI version(s) supported by the plug-in.

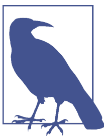
This functionality is implemented in a binary that is installed on the host. The kubelet will communicate with the appropriate CNI binary based on the configuration it expects on the host. An example of this configuration file is as follows:

```
{
  "cniVersion": "0.4.0", ❶
  "name": "dbnet", ❷
  "type": "bridge",
  "bridge": "cni0",
  "args": {
    "labels" : {
        "appVersion" : "1.0"
    }
  },
  "ipam": { ❸
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  }
}
```

❶ The CNI (specification) version this plug-in expects to talk over.

❷ The CNI driver (binary) to send networking setup requests to.

❸ The IPAM driver to use, specified when the CNI plug-in does not handle IPAM.



Multiple CNI configurations may exist in the CNI *conf* directory. They are evaluated lexicographically, and the first configuration will be used.

Along with the CNI configuration and CNI binary, most plug-ins run a Pod on each host that handles concerns beyond interface attachment and IPAM. This includes responsibilities such as route propagation and network policy programming.

CNI Installation

CNI drivers must be installed on every node taking part in the Pod network. Additionally, the CNI configuration must be established. The installation is typically handled when you deploy a CNI plug-in. For example, when deploying Cilium, a DaemonSet is created, which puts a `cilium` Pod on every node. This Pod features a PostStart command that runs the baked-in script *install-cni.sh*. This script will start by installing two drivers. First it will install the loopback driver to support the `lo` interface. Then it will install the `cilium` driver. The script executes conceptually as follows (the example has been greatly simplified for brevity):

```
# Install CNI drivers to host

# Install the CNI loopback driver; allow failure
cp /cni/loopback /opt/cin/bin/ || true

# install the cilium driver
cp /opt/cni/bin/cilium-cni /opt/cni/bin/
```

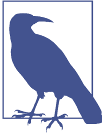
After installation, the kubelet still needs to know which driver to use. It will look within */etc/cni/net.d/* (configurable via flag) to find a CNI configuration. The same *install-cni.sh* script adds this as follows:

```
cat > /etc/cni/net.d/05-cilium.conf <<EOF
{
  "cniVersion": "0.3.1",
  "name": "cilium",
  "type": "cilium-cni",
  "enable-debug": ${ENABLE_DEBUG}
}
EOF
```

To demonstrate this order of operations, let's take a look a newly bootstrapped, single-node cluster. This cluster was bootstrapped using `kubeadm`. Examining all Pods reveals that the `core-dns` Pods are not running:

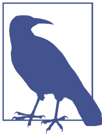
```
NAMESPACE     NAME                           READY   STATUS    RESTARTS   AGE
kube-system   coredns-f9fd979d6-26lfr        0/1     Pending   0          3m14s
kube-system   coredns-f9fd979d6-zqzft        0/1     Pending   0          3m14s
kube-system   etcd-test                      1/1     Running   0          3m26s
kube-system   kube-apiserver-test            1/1     Running   0          3m26s
kube-system   kube-controller-manager-test   1/1     Running   0          3m26s
kube-system   kube-proxy-xhh2p               1/1     Running   0          3m14s
kube-system   kube-scheduler-test            1/1     Running   0          3m26s
```

After examining the kubelet logs on the host scheduled to run `core-dns`, it becomes clear that the lack of CNI configuration is causing the container runtime to not start the Pod:



This case of DNS not starting is one of the most common indicators of CNI issues after cluster bootstrapping. Another symptom is nodes reporting `NotReady` status.

```
# journalctl -f -u kubelet

-- Logs begin at Sun 2020-09-27 15:40:13 UTC. --
Sep 27 17:11:18 test kubelet[2972]: E0927 17:11:18.817089 2972 kubelet.go:2103]
Container runtime network not ready: NetworkReady=false
reason:NetworkPluginNotReady message:docker: network plugin is not ready: cni
config uninitialized
Sep 27 17:11:19 test kubelet[2972]: W0927 17:11:19.198643 2972 cni.go:239]
Unable to update cni config: no networks found in /etc/cni/net.d
```



The reason Pods such as `kube-apiserver` and `kube-controller-manager` started successfully is due to their use of the host network. Since they leverage the host network and do not rely on the Pod network, they are not susceptible to the same behavior seen by `core-dns`.

Cilium can be deployed to the cluster by simply applying a YAML file from the Cilium documentation. In doing so, the aforementioned `cilium` Pod is deployed on every node, and the *cni-install.sh* script is run. Examining the CNI bin and configuration directories, we can see the installed components:

```
# ls /opt/cni/bin/ | grep -i cilium
cilium-cni

# ls /etc/cni/net.d/ | grep -i cilium
05-cilium.conf
```

With this in place, the kubelet and container runtime are functioning as expected. Most importantly, the `core-dns` Pod is up and running! Figure 5-4 demonstrates the relationship we've covered thus far in this section.

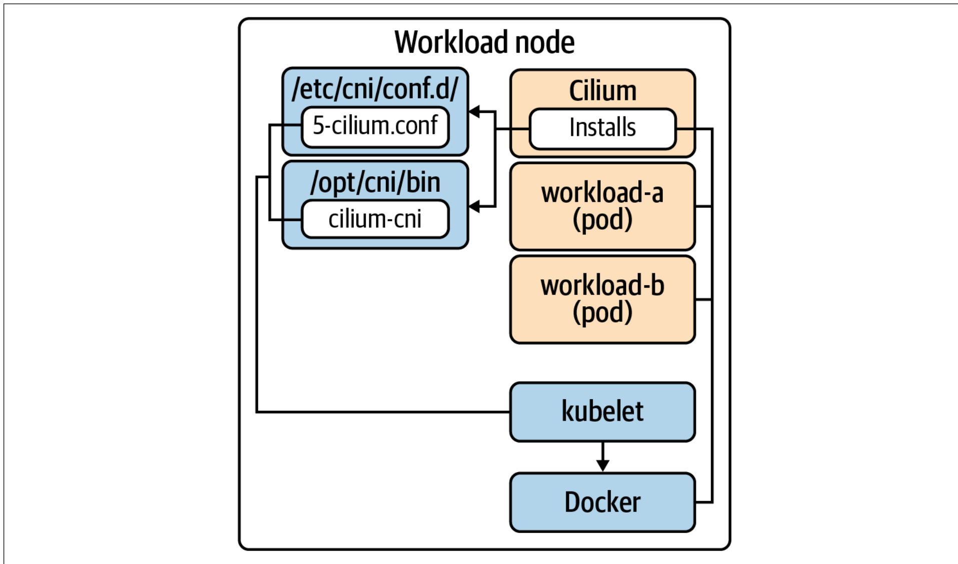
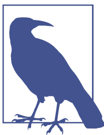


Figure 5-4. Docker is used to run containers. The kubelet interacts with the CNI to attach network interfaces and configure the Pod's network.

While this example explored installation via Cilium, most plug-ins follow a similar deployment model. The key justification for plug-in choice is based on the discussion in "Networking Considerations" on page 102. With this in mind, we'll transition to exploring some CNI plug-ins to better understand different approaches.

CNI Plug-ins

Now we are going to explore a few implementations of CNI. CNI has one of the largest array of options relative to other interfaces such as CRI. As such, we won't be exhaustive in the plug-ins we cover and encourage you to explore more than what we will. We chose the following plug-ins as a factor of being the most common we see at clients and unique enough to demonstrate the variety of approaches.



A Pod network is foundational to any Kubernetes cluster. As such, your CNI plug-in will be in the critical path. As time goes on, you may wish to change your CNI plug-in. If this occurs, we recommend rebuilding clusters as opposed to doing in-place migrations. In this approach, you spin up a new cluster featuring the new CNI. Then, depending on your architecture and operational model, migrate workloads to the new cluster. It is possible to do an in-place CNI migration, but it takes on nontrivial risk and should be carefully weighed against our recommendation.

Calico

Calico is a well-established CNI plug-in in the cloud native ecosystem. Project Calico is the open source project that supports this CNI plug-in, and Tigera is the commercial company offering enterprise features and support. Calico makes heavy use of BGP to propagate workload routes between nodes and to offer integration with larger datacenter fabrics. Along with installing a CNI binary, Calico runs a `calico-node` agent on each host. This agent features a BIRD daemon for facilitating BGP peering between nodes and a Felix agent, which takes the known routes and programs them into the kernel route tables. This relationship is demonstrated in Figure 5-5.

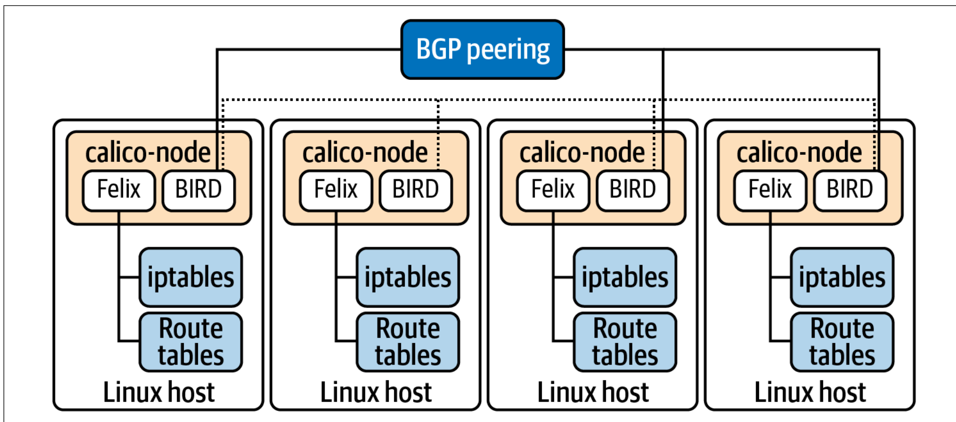


Figure 5-5. Calico component relationship showing the BGP peering to communicate routes and the programming of iptables and kernel routing tables accordingly.

For IPAM, Calico initially respects the `cluster-cidr` setting described in "IP Address Management" on page 102. However, its capabilities are far greater than relying on a CIDR allocation per node. Calico creates CRDs called IPPools. This provides a lot of flexibility in IPAM, specifically enabling features such as:

- Configuring block size per node
- Specifying what node(s) an IPPool applies to
- Allocating IPPools to Namespaces, rather than nodes
- Configuring routing behavior

Paired with the ability to have multiple pools per cluster, you have a lot of flexibility in IPAM and network architecture. By default, clusters run a single IPPool, as shown here:

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
  name: default-ipv4-ippool
spec:
  cidr: 10.30.0.0/16 ❶
  blockSize: 29 ❷
  ipipMode: Always ❸
  natOutgoing: true
```

❶ The cluster's Pod network CIDR.

❷ The size of each node-level CIDR allocation.

❸ The encapsulation mode.

Calico offers a variety of ways to route packets inside of the cluster. This includes:

Native
: No encapsulation of packets.

IP-in-IP
: Simple encapsulation. IP packet is placed in the payload of another.

VXLAN
: Advanced encapsulation. An entire L2 frame is encapsulated within a UDP packet. Establishes a virtual L2 overlay.

Your choice is often a function of what your network can support. As described in "Routing Protocols" on page 104, native routing will likely provide the best performance, smallest packet size, and simplest troubleshooting experience. However, in many environments, especially those involving multiple subnets, this mode is not possible. The encapsulation approaches work in most environments, especially VXLAN. Additionally, the VXLAN mode does not require usage of BGP, which can be a solution to environments where BGP peering is blocked. One unique feature of Calico's encapsulation approach is that it can be enabled exclusively for traffic that crosses a subnet boundary. This enables near native performance when routing within the subnet while not breaking routing outside the subnet. This can be enabled by setting the IPPool's `ipipMode` to `CrossSubnet`. Figure 5-6 demonstrates this behavior.

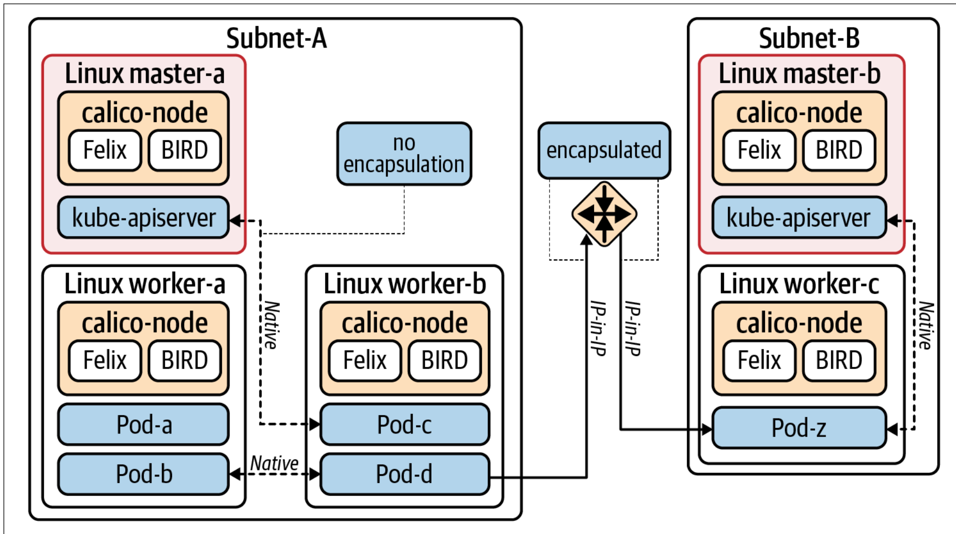


Figure 5-6. Traffic behavior when CrossSubnet IP-in-IP mode is enabled.

For deployments of Calico that keep BGP enabled, by default, no additional work is needed thanks to the built-in BGP daemon in the `calico-node` Pod. In more complex architectures, organizations use this BGP functionality as a way to introduce route reflectors, sometimes required at large scales when the (default) full-mesh approach becomes limited. Along with route reflectors, peering can be configured to talk to network routers, which in turn can make the overall network aware of routes to Pod IPs. This is all configured using Calico's BGPPeer CRD, seen here:

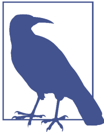
```
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: external-router
spec:
  peerIP: 192.23.11.100 ❶
  asNumber: 64567 ❷
  nodeSelector: routing-option == 'external' ❸
```

❶ The IP of the device to (bgp) peer with.

❷ The autonomous system ID of the cluster.

❸ Which cluster nodes should peer with this device. This field is optional. When omitted, the BGPPeer configuration is considered *global*. Not peering global is advisable only when a certain set of nodes should offer a unique routing capability, such as offering routable IPs.

In terms of network policy, Calico fully implements the Kubernetes NetworkPolicy API. Calico offers two additional CRDs for increased functionality. These include (projectcalico.org/v3).NetworkPolicy and GlobalNetworkPolicy. These Calico-specific APIs look similar to Kubernetes NetworkPolicy but feature more capable rules and richer expressions for evaluation. Also, policy ordering and application layer policy (requires integration with Istio) are supported. GlobalNetworkPolicy is particularly useful because it applies policy at a cluster-wide level. This makes it easier to achieve models such as micro-segmentation, where all traffic is denied by default and egress/ingress is opened up based on the workload needs. You can apply a GlobalNetworkPolicy that denies all traffic except for critical services such as DNS. Then, at a Namespace level, you can open up access to ingress and egress accordingly. Without GlobalNetworkPolicy, we'd need to add and manage deny-all rules in *every* Namespace.



Historically, Calico has made use of iptables to implement packet routing decisions. For Services, Calico relies on the programming done by kube-proxy to resolve an endpoint for a Service. For network policy, Calico programs iptables to determine whether a packet should be allowed to enter or leave the host. At the time of this writing, Calico has introduced an eBPF dataplane option. We expect, over time, more functionality used by Calico to be moved into this model.

Cilium

Cilium is a newer CNI plug-in relative to Calico. It's the first CNI plug-in to utilize the extended Berkeley Packet Filter (eBPF). This means that rather than processing packets in userspace it is able to do so without leaving kernel space. Paired with eXpress Data Path (XDP), hooks may be established in the NIC driver to make routing decisions. This enables routing decisions to occur immediately when the packet is received.

As a technology, eBPF has demonstrated performance and scale at organizations such as Facebook and Netflix. With the usage of eBPF, Cilium is able to claim increased features around scalability, observability, and security. This deep integration with BPF means that common CNI concerns such as NetworkPolicy enforcement are no longer handled via iptables in userspace. Instead, extensive use of eBPF maps enable decisions to occur quickly in a way that scales as more and more rules are added. Figure 5-7 shows a high-level overview of the stack with Cilium installed.

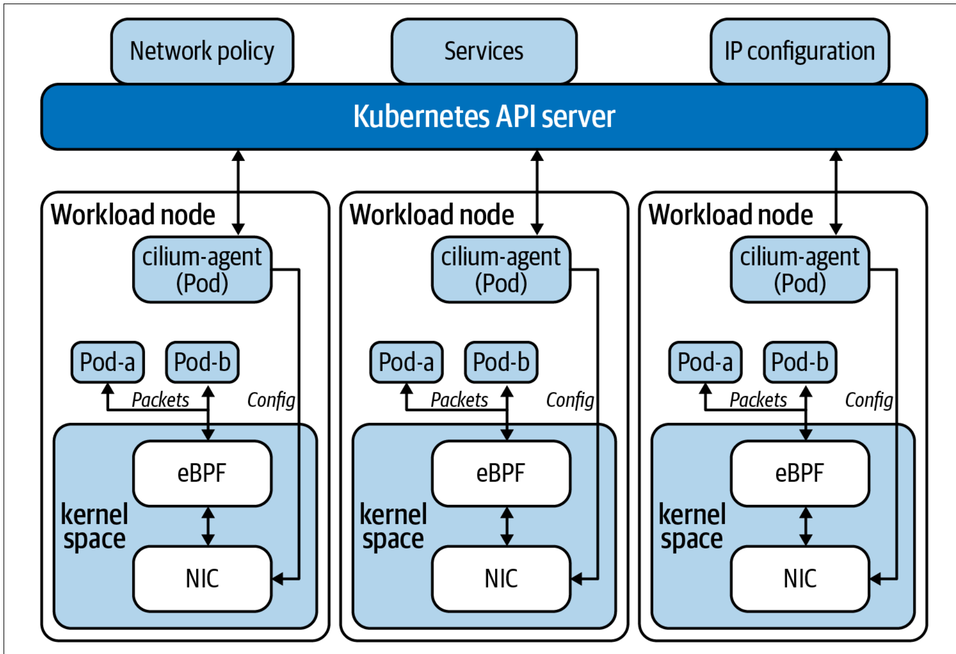


Figure 5-7. Cilium interacts with eBPF maps and programs at the kernel level.

For IPAM, Cilium follows the model of either delegating IPAM to a cloud provider integration or managing it itself. In the most common scenario of Cilium managing IPAM, it will allocate Pod CIDRs to each node. By default, Cilium will manage these CIDRs independent of Kubernetes Node allocations. The node-level addressing will be exposed in the `CiliumNode` CRD. This will provide greater flexibility in management of IPAM and is preferable. If you wish to stick to the default CIDR allocations done in Kubernetes based on its Pod CIDR, Cilium offer a `kubernetes` IPAM mode. This will rely on the Pod CIDR allocated to each node, which is exposed in the Node object. Following is an example of a `CiliumNode` object. You can expect one of these to exist for each node in the cluster:

```
apiVersion: cilium.io/v2
kind: CiliumNode
metadata:
  name: node-a
spec:
  addresses:
  - ip: 192.168.122.126 ❶
    type: InternalIP
  - ip: 10.0.0.245
    type: CiliumInternalIP
  health:
    ipv4: 10.0.0.78
```

```
ipam:
  podCIDRs:
  - 10.0.0.0/24 ❷
```

❶ IP address of this workload node.

❷ CIDR allocated to this node. The size of this allocation can be controlled in Cilium's config using `cluster-pool-ipv4-mask-size: "24"`.

Similar to Calico, Cilium offers encapsulated and native routing modes. The default mode is encapsulated. Cilium supports using tunneling protocols VXLAN or Geneve. This mode should work with most networks as long as host-to-host routability preexists. To run in native mode, Pod routes must be understood at some level. For example, Cilium supports using AWS's ENI for IPAM. In this model, the Pod IPs are known to the VPC and are inherently routable. To run a native-mode with Cilium-managed IPAM, assuming the cluster runs in the same L2 segment, `auto-direct-node-routes: true` can be added to Cilium's configuration. Cilium will then program the host's route tables accordingly. If you span L2 networks, you may need to introduce additional routing protocols such as BGP to distribute routes.

In terms of network policy, Cilium can enforce the Kubernetes NetworkPolicy API. As an alternative to this policy, Cilium offers its own CiliumNetworkPolicy and CiliumClusterwideNetworkPolicy. The key difference between these two is the scope of the policy. CiliumNetworkPolicy is Namespace scoped, while CiliumClusterwideNetworkPolicy is cluster-wide. Both of these have increased functionality beyond the capabilities of Kubernetes NetworkPolicy. Along with supporting label-based layer 3 policy, they support policy based on DNS resolution and application-level (layer 7) requests.

While most CNI plug-ins don't involve themselves with Services, Cilium offers a fully featured kube-proxy replacement. This functionality is built into the `cilium-agent` deployed to each node. To deploy in the mode, you'll want to ensure kube-proxy is absent from your cluster and that the `KubeProxyReplacement` setting is set to `strict` in Cilium. When using this mode, Cilium will configure routes for Services within eBPF maps, making resolution as fast as O(1). This is in contrast to kube-proxy, which implements Services in iptables chains and can cause issues at scale and/or when there is high churn of Services. Additionally, the CLI provided by Cilium offers a good experience when troubleshooting constructs such as Services or network policy. Rather than trying to interpret iptables chains, you can query the system as follows:

```
kubectl exec -it -n kube-system cilium-fmh8d -- cilium service list

ID   Frontend             Service Type   Backend
[...]
7    192.40.23.111:80     ClusterIP      1 => 10.30.0.28:80
                                         2 => 10.30.0.21:80
```

Cilium's use of eBPF programs and maps makes it an extremely compelling and interesting CNI option. By continuing to leverage eBPF programs, more functionality is being introduced that integrates with Cilium—for example, the ability to extract flow data, policy violations, and more. To extract and present this valuable data, hubble was introduced. It makes use of Cilium's eBPF programs to provide a UI and CLI for operators.

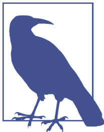
Lastly, we should mention that the eBPF functionality made available by Cilium can be run alongside many existing CNI providers. This is achieved by running Cilium in its CNI chaining mode. In this mode, an existing plug-in such as AWS's VPC CNI will handle routing and IPAM. Cilium's responsibility will exclusively be the functionality offered by its various eBPF programs including network observability, load balancing, and network policy enforcement. This approach can be preferable when you either cannot fully run Cilium in your environment or wish to test out its functionality alongside your current CNI choice.

AWS VPC CNI

AWS's VPC CNI demonstrates a very different approach to what we have covered thus far. Rather than running a Pod network independent of the node network, it fully integrates Pods into the same network. Since a second network is not being introduced, the concerns around distributing routes or tunneling protocols are no longer needed. When a Pod is provided an IP, it becomes part of the network in the same way an EC2 host would. It is subject to the same route tables as any other host in the subnet. Amazon refers to this as native VPC networking.

For IPAM, a daemon will attach a second elastic network interface (ENI) to the Kubernetes node. It will then maintain a pool of secondary IPs that will eventually get attached to Pods. The amount of IPs available to a node depends on the EC2 instance size. These IPs are typically "private" IPs from within the VPC. As mentioned earlier in this chapter, this will consume IP space from your VPC and make the IPAM system more complex than a completely independent Pod network. However, the routing of traffic and troubleshooting has been significantly simplified given we are not introducing a new network! Figure 5-8 demonstrates the IPAM setup with AWS VPC CNI.

Figure 5-8. The IPAM daemon is responsible for maintaining the ENI and pool of secondary IPs.



The use of ENIs will impact the number of Pods you can run per node. AWS maintains a list on its GitHub page that correlates instance type to max Pods.

Multus

So far, we have covered specific CNI plug-ins that attach an interface to a Pod, thus making it available on a network. But what if a Pod needs to be attached to more than one network? This is where the Multus CNI plug-in comes in. While not extremely common, there are use cases in the telecommunications industry that require their network function virtualizations (NFVs) to route traffic to a specific, dedicated, network.

Multus can be thought of as a CNI that enables using multiple other CNIs. In this model, Multus becomes the CNI plug-in interacted with by Kubernetes. Multus is configured with a default network that is commonly the network expected to facilitate Pod-to-Pod communication. This could even be one of the plug-ins we've talked about in this chapter! Then, Multus supports configuring secondary networks by specifying additional plug-ins that can be used to attach another interface to a Pod. Pods can then be annotated with something like `k8s.v1.cni.cncf.io/networks: sriov-conf` to attach an additional network. Figure 5-9 shows the result of this configuration.

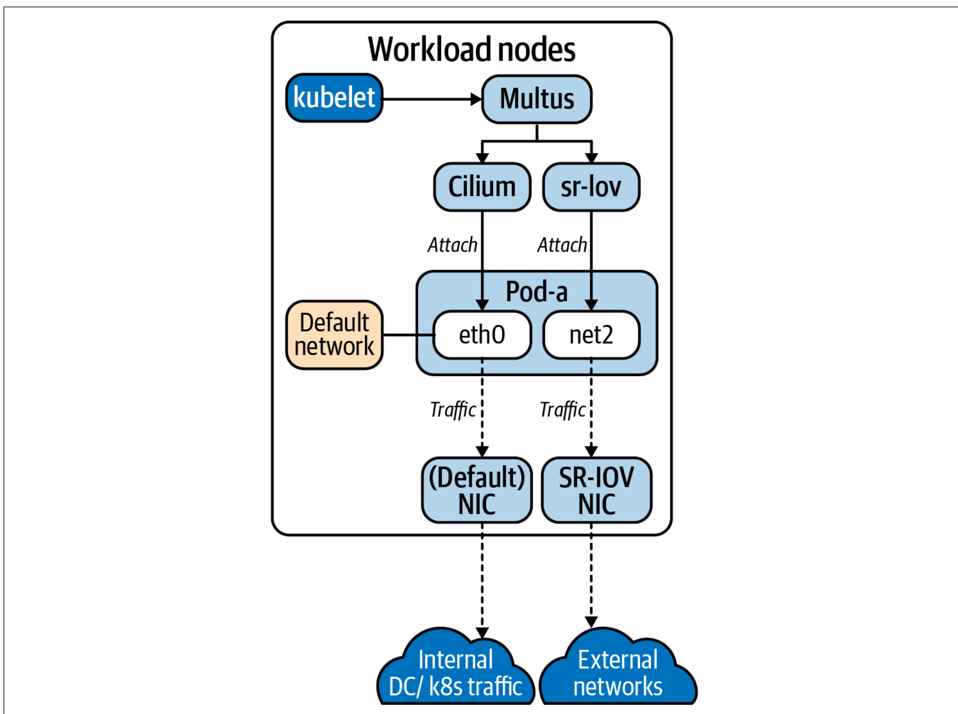


Figure 5-9. The traffic flow of a multinetwork Multus configuration.

Additional Plug-ins

The landscape of plug-ins is vast, and we've covered only a very small subset. However, the ones covered in this chapter do identify some of the key variances you'll find in plug-ins. The majority of alternatives take differing approaches to the engine used to facilitate the networking, yet many core principles stay the same. The following list identifies some additional plug-ins and gives a small glimpse into their networking approach:

Antrea
: Data plane is facilitated via Open vSwitch. Offers high-performance routing along with the ability to introspect flow data.

Weave
: Overlay network that provides many mechanisms to route traffic—for example, the fast datapath options using OVS modules to keep packet processing in the kernel.

flannel
: Simple layer-3 network for Pods and one of the early CNIs. It supports multiple backends yet is most commonly configured to use VXLAN.

Summary

The Kubernetes/container networking ecosystem is filled with options. This is good! As we've covered throughout this chapter, networking requirements can vary significantly from organization to organization. Choosing a CNI plug-in is likely to be one of the most foundational considerations for your eventual application platform. While exploring the many options may feel overwhelming, we highly recommend you work to better understand the networking requirements of your environment and applications. With a deep understanding of this, the right networking plug-in choice should fall into place!

CHAPTER 6
Service Routing

Service routing is a crucial capability of a Kubernetes-based platform. While the container networking layer takes care of the low-level primitives that connect Pods, developers need higher-level mechanisms to interconnect services (i.e., east-west service routing) and to expose applications to their clients (i.e., north-south service routing). Service routing encompasses three concerns that provide such mechanisms: Services, Ingress, and service mesh.

Services provide a way to treat a set of Pods as a single unit or network service. They provide load balancing and routing features that enable horizontal scaling of applications across the cluster. Furthermore, Services offer service discovery mechanisms that applications can use to discover and interact with their dependencies. Finally, Services also provide layer 3/4 mechanisms to expose workloads to network clients outside of the cluster.

Ingress handles north-south routing in the cluster. It serves as an entry point into workloads running in the cluster, mainly HTTP and HTTPS services. Ingress provides layer 7 load balancing capabilities that enable more granular traffic routing than Services. The load balancing of traffic is handled by an Ingress controller, which must be installed in the cluster. Ingress controllers leverage proxy technologies such as Envoy, NGINX, or HAProxy. The controller gets the Ingress configuration from the Kubernetes API and configures the proxy accordingly.

A service mesh is a service routing layer that provides advanced routing, security, and observability features. It is mainly concerned with east-west service routing, but some implementations can also handle north-south routing. Services in the mesh communicate with each other through proxies that augment the connection. The use of proxies makes meshes compelling, as they enhance workloads without changes to source code.

This chapter digs into these service routing capabilities, which are critical in production Kubernetes platforms. First, we will discuss Services, the different Service types, and how they are implemented. Next, we will explore Ingress, Ingress controllers, and the different considerations to take into account when running Ingress in production. Finally, we will cover service meshes, how they work on Kubernetes, and considerations to make when adopting a service mesh in a production platform.

Kubernetes Services

The Kubernetes Service is foundational to service routing. The Service is a network abstraction that provides basic load balancing across several Pods. In most cases, workloads running in the cluster use Services to communicate with each other. Using Services instead of Pod IPs is preferred because of the fungible nature of Pods.

In this section, we will review Kubernetes Services and the different Service types. We will also look at Endpoints, another Kubernetes resource that is intimately related to Services. We will then dive into the Service implementation details and discuss kube-proxy. Finally, we will discuss Service Discovery and considerations to make for the in-cluster DNS server.

The Service Abstraction

The Service is a core API resource in Kubernetes that load balances traffic across multiple Pods. The Service does load balancing at the L3/L4 layers in the OSI model. It takes a packet with a destination IP and port and forwards it to a backend Pod.

Load balancers typically have a frontend and a backend pool. Services do as well. The frontend of a Service is the ClusterIP. The ClusterIP is a virtual IP address (VIP) that is accessible from within the cluster. Workloads use this VIP to communicate with the Service. The backend pool is a collection of Pods that satisfy the Service's Pod selector. These Pods receive the traffic destined for the Cluster IP. Figure 6-1 depicts the frontend of a Service and its backend pool.

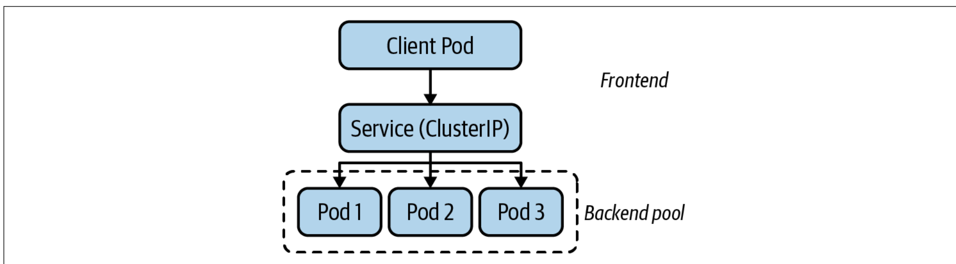


Figure 6-1. The Service has a frontend and a backend pool. The frontend is the ClusterIP, while the backend is a set of Pods.

Service IP Address Management

As we discussed in the previous chapter, you configure two ranges of IP addresses when deploying Kubernetes. On the one hand, the Pod IP range or CIDR block provides IP addresses to each Pod in the cluster. On the other hand, the Service CIDR block provides the IP addresses for Services in the cluster. This CIDR is the range that Kubernetes uses to assign ClusterIPs to Services.

The API server handles the IP Address Management (IPAM) for Kubernetes Services. When you create a Service, the API Server (with the help of etcd) allocates an IP address from the Service CIDR block and writes it to the Service's ClusterIP field.

When creating Services, you can also specify the ClusterIP in the Service specification. In this case, the API Server makes sure that the requested IP address is available and within the Services CIDR block. With that said, explicitly setting ClusterIPs is an antipattern.

The Service resource

The Service resource contains the configuration of a given Service, including the name, type, ports, etc. Example 6-1 is an example Service definition in its YAML representation named `nginx`.

Example 6-1. Service definition that exposes NGINX on a ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector: ❶
    app: nginx
  ports: ❷
    - protocol: TCP ❸
      port: 80 ❹
      targetPort: 8080 ❺
  clusterIP: 172.21.219.227 ❻
  type: ClusterIP
```

❶ The Pod selector. Kubernetes uses this selector to find the Pods that belong to this Service.

❷ Ports that are accessible through the Service.

❸ Kubernetes supports TCP, UDP, and SCTP protocols in Services.

❹ Port where the Service can be reached.

❺ Port where the backend Pod is listening, which can be different than the port exposed by the Service (the `port` field above).

❻ Cluster IP that Kubernetes allocated for this Service.

The Service's Pod selector determines the Pods that belong to the Service. The Pod selector is a collection of key/value pairs that Kubernetes evaluates against Pods in the same Namespace as the Service. If a Pod has the same key/value pairs in their labels, Kubernetes adds the Pod's IP address to the backend pool of the Service. The management of the backend pool is handled by the Endpoints controller through Endpoints resources. We will discuss Endpoints in more detail later in this chapter.

Service types

Up to this point, we have mainly talked about the ClusterIP Service, which is the default Service type. Kubernetes offers multiple Service types that offer additional features besides the Cluster IP. In this section, we will discuss each Service type and how they are useful.

ClusterIP. We have already discussed this Service type in the previous sections. To recap, the ClusterIP Service creates a virtual IP address (VIP) that is backed by one or more Pods. Usually, the VIP is available only to workloads running inside the cluster. Figure 6-2 shows a ClusterIP Service.

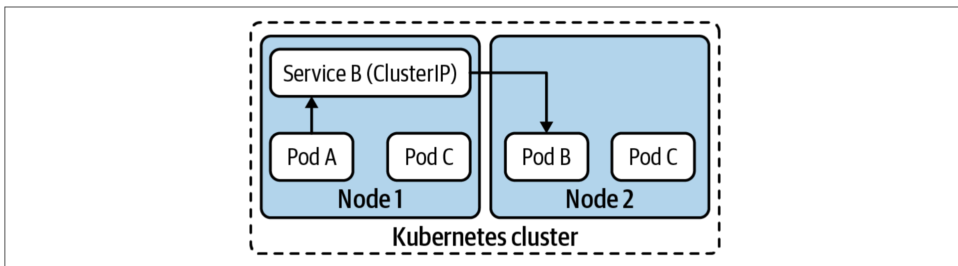


Figure 6-2. The ClusterIP Service is a VIP that is accessible to workloads running within the cluster.

NodePort. The NodePort Service is useful when you need to expose a Service to network clients outside of the cluster, such as existing applications running in VMs or users of a web application.

As the name suggests, the NodePort Service exposes the Service on a port across all cluster nodes. The port is assigned randomly from a configurable port range. Once assigned, all nodes in the cluster listen for connections on the given port. Figure 6-3 shows a NodePort Service.

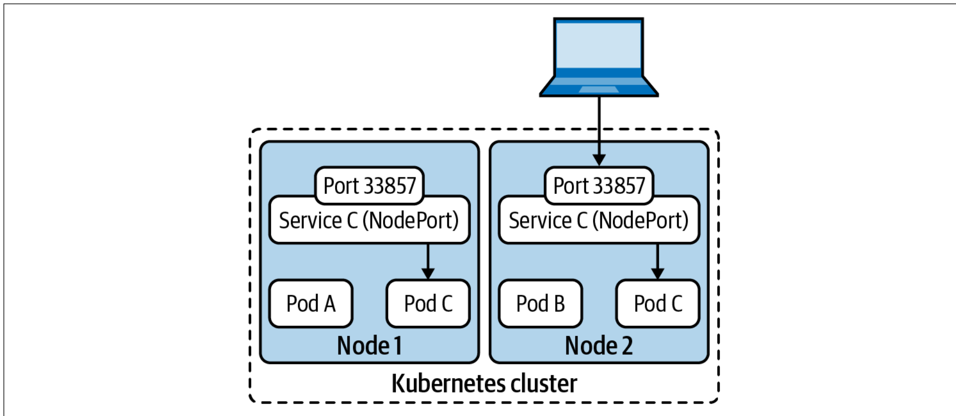


Figure 6-3. The NodePort Service opens a random port on all cluster nodes. Clients outside of the cluster can reach the Service through this port.

The primary challenge with NodePort Services is that clients need to know the Service's node port number and the IP address of at least one cluster node to reach the Service. This is problematic because nodes can fail or be removed from the cluster.

A common way to solve this challenge is to use an external load balancer in front of the NodePort Service. With this approach, clients don't need to know the IP addresses of cluster nodes or the Service's port number. Instead, the load balancer functions as the single entry point to the Service.

The downside to this solution is that you need to manage external load balancers and update their configuration constantly. Did a developer create a new NodePort Service? Create a new load balancer. Did you add a new node to the cluster? Add the new node to the backend pool of all load balancers.

In most cases, there are better alternatives to using a NodePort Service. The LoadBalancer Service, which we'll discuss next, is one of those options. Ingress controllers are another option, which we'll explore later in this chapter in "Ingress" on page 152.

LoadBalancer. The LoadBalancer Service builds upon the NodePort Service to address some of its downsides. At its core, the LoadBalancer Service is a NodePort Service under the hood. However, the LoadBalancer Service has additional functionality that is satisfied by a controller.

The controller, also known as a cloud provider integration, is responsible for automatically gluing the NodePort Service with an external load balancer. In other words, the controller takes care of creating, managing, and configuring external load balancers in response to the configuration of LoadBalancer Services in the cluster. The controller does this by interacting with APIs that provision or configure load balancers. This interaction is depicted in Figure 6-4.

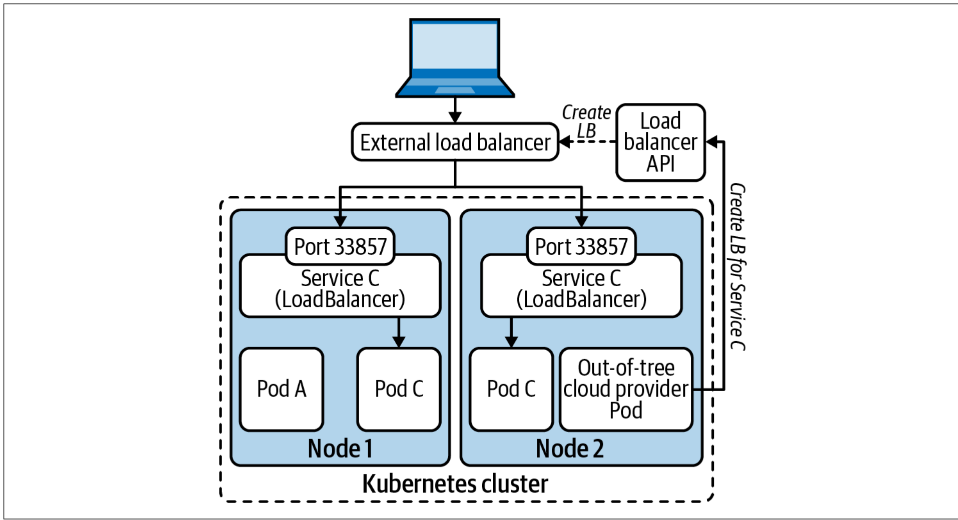


Figure 6-4. The LoadBalancer Service leverages a cloud provider integration to create an external load balancer, which forwards traffic to the node ports. At the node level, the Service is the same as a NodePort.

Kubernetes has built-in controllers for several cloud providers, including Amazon Web Services (AWS), Google Cloud, and Microsoft Azure. These integrated controllers are usually called in-tree cloud providers, as they are implemented inside the Kubernetes source code tree.

As the Kubernetes project evolved, out-of-tree cloud providers emerged as an alternative to in-tree providers. Out-of-tree providers enabled load balancer vendors to provide their implementations of the LoadBalancer Service control loop. At this time, Kubernetes supports both in-tree and out-of-tree providers. However, the community is quickly adopting out-of-tree providers, given that the in-tree providers are deprecated.

LoadBalancer Services Without a Cloud Provider

If you run Kubernetes without a cloud provider integration, you will notice that LoadBalancer Services remain in the "Pending" state. A great example of this problem is bare metal deployments. If you are running your platform on bare-metal, you might be able to leverage MetalLB to support LoadBalancer Services.

MetalLB is an open source project that provides support for LoadBalancer Services on bare metal. MetalLB runs in the cluster, and it can operate in one of two modes. In layer 2 mode, one of the cluster nodes becomes the leader and starts responding to ARP requests for the external IPs of LoadBalancer Services. Once traffic reaches the leader, kube-proxy handles the rest. If the leader fails, another node in the cluster

takes over and begins handling requests. A big downside of this mode is that it does not offer true load balancing capabilities, given that a single node is the one responding to the ARP requests.

The second mode of operation uses BGP to peer with your network routers. Through the peering relationship, MetalLB advertises the external IPs of LoadBalancer Services. Similar to the layer 2 mode, kube-proxy takes care of routing the traffic from one of the cluster nodes to the backend Pod. The BGP mode addresses the limitations of the layer 2 mode, given that traffic is load balanced across multiple nodes instead of a single, leader node.

If you need to support LoadBalancer Services, MetalLB might provide a viable path forward. In most cases, however, you can get away without supporting LoadBalancer Services. For example, if a large proportion of your applications are HTTP services, you can leverage an Ingress controller to load balance and bring traffic into these applications.

ExternalName. The ExternalName Service type does not perform any kind of load balancing or proxying. Instead, an ExternalName Service is primarily a service discovery construct implemented in the cluster's DNS. An ExternalName Service maps a cluster Service to a DNS name. Because there is no load balancing involved, Services of this type lack ClusterIPs.

ExternalName Services are useful in different ways. Piecemeal application migration efforts, for example, can benefit from ExternalName Services. If you migrate components of an application to Kubernetes while leaving some of its dependencies outside, you can use an ExternalName Service as a bridge while you complete the migration. Once you migrate the entire application, you can change the Service type to a ClusterIP without having to change the application deployment.

Even though useful in creative ways, the ExternalName Service is probably the least common Service type in use.

Headless Service. Like the ExternalName Service, the Headless Service type does not allocate a ClusterIP or provide any load balancing. The Headless Service merely functions as a way to register a Service and its Endpoints in the Kubernetes API and the cluster's DNS server.

Headless Services are useful when applications need to connect to specific replicas or Pods of a service. Such applications can use service discovery to find all the Pod IPs behind the Service and then establish connections to specific Pods.

Supported communication protocols

Kubernetes Services support a specific set of protocols: TCP, UDP, and SCTP. Each port listed in a Service resource specifies the port number and the protocol. Services can expose multiple ports with different protocols. For example, the following snippet shows the YAML definition of the kube-dns Service. Notice how the list of ports includes TCP port 53 and UDP port 53:

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: KubeDNS
  name: kube-dns
  namespace: kube-system
spec:
  clusterIP: 10.96.0.10
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  - name: dns-tcp
    port: 53
    protocol: TCP
    targetPort: 53
  - name: metrics
    port: 9153
    protocol: TCP
    targetPort: 9153
  selector:
    k8s-app: kube-dns
  type: ClusterIP
```

Protocols and Troubleshooting Services

While working with Kubernetes, you might have tried to use ping to troubleshoot Kubernetes Services. You probably found that any attempts to ping a Service resulted in 100% packet loss. The problem with using ping is that it uses ICMP datagrams, which are not supported in Kubernetes Services.

Instead of using ping, you must resort to alternative tools when it comes to troubleshooting Services. If you are looking to test connectivity, choose a tool that works with the Service's protocol. For example, if you need to troubleshoot a web server, you can use telnet to test whether you can establish a TCP connection to the server.

Another quick way to troubleshoot Services is to verify that the Pod selector is selecting at least one Pod by checking the corresponding Endpoints resource. Invalid selectors are a common issue with Services.

As we've discussed up to this point, Services load balance traffic across Pods. The Service API resource represents the frontend of the load balancer. The backend, or the collection of Pods that are behind the load balancer, are tracked by the Endpoints resource and controller, which we will discuss next.

Endpoints

The Endpoints resource is another API object that is involved in the implementation of Kubernetes Services. Every Service resource has a sibling Endpoints resource. If you recall the load balancer analogy, you can think of the Endpoints object as the pool of IP addresses that receive traffic. Figure 6-5 shows the relationship between a Service and an Endpoint.

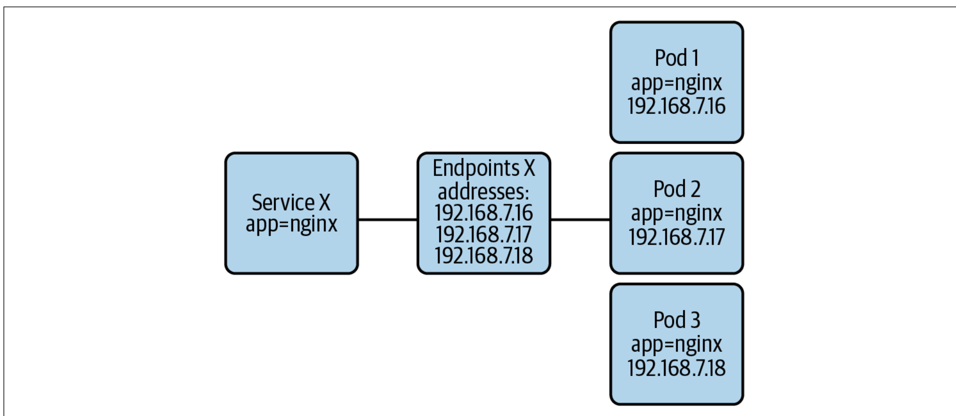


Figure 6-5. Relationship between the Service and the Endpoints resources.

The Endpoints resource

An example Endpoints resource for the `nginx` Service in Example 6-1 might look like this (some extraneous fields have been removed):

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    run: nginx
  name: nginx
  namespace: default
subsets:
- addresses:
```

```
- ip: 10.244.0.10
  nodeName: kube03
  targetRef:
    kind: Pod
    name: nginx-76df748b9-gblnn
    namespace: default
- ip: 10.244.0.9
  nodeName: kube04
  targetRef:
    kind: Pod
    name: nginx-76df748b9-gb7wl
    namespace: default
ports:
- port: 8080
  protocol: TCP
```

In this example, there are two Pods backing the `nginx` Service. Network traffic destined to the `nginx` ClusterIP is load balanced across these two Pods. Also notice how the port is 8080 and not 80. This port matches the `targetPort` field specified in the Service. It is the port that the backend Pods are listening on.

The Endpoints controller

An interesting thing about the Endpoints resource is that Kubernetes creates it automatically when you create a Service. This is somewhat different from other API resources that you usually interact with.

The Endpoints controller is responsible for creating and maintaining the Endpoints objects. Whenever you create a Service, the Endpoints controller creates the sibling Endpoints resource. More importantly, it also updates the list of IPs within the Endpoints object as necessary.

The controller uses the Service's Pod selector to find the Pods that belong to the Service. Once it has the set of Pods, the controller grabs the Pod IP addresses and updates the Endpoints resource accordingly.

Addresses in the Endpoints resource can be in one of two sets: (ready) addresses and notReadyAddresses. The Endpoints controller determines whether an address is ready by inspecting the corresponding Pod's Ready condition. The Pod's Ready condition, in turn, depends on multiple factors. One of them, for example, is whether the Pod was scheduled. If the Pod is pending (not scheduled), its Ready condition is false. Ultimately, a Pod is considered ready when it is running and passing its readiness probe.

Pod readiness and readiness probes

In the previous section, we discussed how the Endpoints controller determines whether a Pod IP address is ready to accept traffic. But how does Kubernetes tell whether a Pod is ready or not?

There are two complementary methods that Kubernetes uses to determine Pod readiness:

Platform information
: Kubernetes has information about the workloads under its management. For example, the system knows whether the Pod has been successfully scheduled to a node. It also knows whether the Pod's containers are up and running.

Readiness probes
: Developers can configure readiness probes on their workloads. When set, the kubelet probes the workload periodically to determine if it is ready to receive traffic. Probing Pods for readiness is more powerful than determining readiness based on platform information because the probe checks for application-specific concerns. For example, the probe can check whether the application's internal initialization process has completed.

Readiness probes are essential. Without them, the cluster would route traffic to workloads that might not be able to handle it, which would result in application errors and irritated end users. Ensure that you always define readiness probes in the applications you deploy to Kubernetes. In Chapter 14, we will further discuss readiness probes.

The EndpointSlices resource

The EndpointSlices resource is an optimization implemented in Kubernetes v1.16. It addresses scalability issues that can arise with the Endpoints resource in large cluster deployments. Let's review these issues and explore how EndpointSlices help.

To implement Services and make them routable, each node in the cluster watches the Endpoints API and subscribes for changes. Whenever an Endpoints resource is updated, it must be propagated to all nodes in the cluster to take effect. A scaling event is a good example. Whenever there is a change to the set of Pods in the Endpoints resource, the API server sends the entire updated object to all the cluster nodes.

This approach to handling the Endpoints API does not scale well with larger clusters for multiple reasons:

- Large clusters contain many nodes. The more nodes in the cluster, the more updates need to be sent when Endpoints objects change.
- The larger the cluster, the more Pods (and Services) you can host. As the number of Pods increases, the frequency of Endpoints resource updates also grows.

- The size of Endpoints resources increases as the number of Pods that belong to the Service grows. Larger Endpoints objects require more network and storage resources.

The EndpointSlices resource fixes these issues by splitting the set of endpoints across multiple resources. Instead of placing all the Pod IP addresses in a single Endpoints resource, Kubernetes splits the addresses across various EndpointSlice objects. By default, EndpointSlice objects are limited to 100 endpoints.

Let's explore a scenario to better understand the impact of EndpointSlices. Consider a Service with 10,000 endpoints, which would result in 100 EndpointSlice objects. If one of the endpoints is removed (due to a scale-in event, for example), the API server sends the affected EndpointSlice object to each node. Sending a single EndpointSlice with 100 endpoints is much more efficient than sending a single Endpoints resource with thousands of endpoints.

To summarize, the EndpointSlices resource improves the scalability of Kubernetes by splitting a large number of endpoints into a set of EndpointSlice objects. If you are running a platform that has Services with hundreds of endpoints, you might benefit from the EndpointSlice improvements. Depending on your Kubernetes version, the EndpointSlice functionality is opt-in. If you are running Kubernetes v1.18, you must set a feature flag in kube-proxy to enable the use of EndpointSlice resources. Starting with Kubernetes v1.19, the EndpointSlice functionality will be enabled by default.

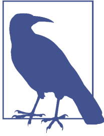
Service Implementation Details

Until now, we've talked about Services, Endpoints, and what they provide to workloads in a Kubernetes cluster. But how does Kubernetes implement Services? How does it all work?

In this section, we will discuss the different approaches available when it comes to realizing Services in Kubernetes. First, we will talk about the overall kube-proxy architecture. Next, we will review the different kube-proxy data plane modes. Finally, we will discuss alternatives to kube-proxy, such as CNI plug-ins that are capable of taking over kube-proxy's responsibilities.

Kube-proxy

Kube-proxy is an agent that runs on every cluster node. It is primarily responsible for making Services available to the Pods running on the local node. It achieves this by watching the API server for Services and Endpoints and programming the Linux networking stack (using iptables, for example) to handle packets accordingly.



Historically, kube-proxy acted as a network proxy between Pods running on the node and Services. This is where the kube-proxy name came from. As the Kubernetes project evolved, however, kube-proxy stopped being a proxy and became more of a node agent or localized control plane.

Kube-proxy supports three modes of operation: userspace, iptables, and IPVS. The userspace proxy mode is seldom used, since iptables and IPVS are better alternatives. Thus, we will only cover the iptables and IPVS modes in the following sections of this chapter.

Kube-proxy: iptables mode

The iptables mode is the default kube-proxy mode at the time of writing (Kubernetes v1.18). It is safe to say that the iptables mode is the most prevalent across cluster installations today.

In the iptables mode, kube-proxy leverages the network address translation (NAT) features of iptables.

ClusterIP Services. To realize ClusterIP Services, kube-proxy programs the Linux kernel's NAT table to perform Destination NAT (DNAT) on packets destined for Services. The DNAT rules replace the packet's destination IP address with the IP address of a Service endpoint (a Pod IP address). Once replaced, the network handles the packet as if it was originally sent to the Pod.

To load balance traffic across multiple Service endpoints, kube-proxy uses multiple iptables chains:

Services chain
: Top-level chain that contains a rule for each Service. Each rule checks whether the destination IP of the packet matches the ClusterIP of the Service. If it does, the packet is sent to the Service-specific chain.

Service-specific chain
: Each Service has its iptables chain. This chain contains a rule per Service endpoint. Each rule uses the `statistic` iptables extension to select a target endpoint randomly. Each endpoint has 1/n probability of being selected, where n is the number of endpoints. Once selected, the packet is sent to the Service endpoint chain.

Service endpoint chain
: Each Service endpoint has an iptables chain that performs DNAT on the packet.

The following listing of iptables rules shows an example of a ClusterIP Service. The Service is called nginx and has three endpoints (extraneous iptables rules have been removed for brevity):

```
$ iptables --list --table nat
Chain KUBE-SERVICES (2 references) ❶
target     prot opt source               destination
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.97.85.96
    /* default/nginx: cluster IP */ tcp dpt:80
KUBE-SVC-4N57TFCL4MD7ZTDA  tcp  --  anywhere             10.97.85.96
    /* default/nginx: cluster IP */ tcp dpt:80
KUBE-NODEPORTS  all  --  anywhere             anywhere
    /* kubernetes service nodeports; NOTE: this must be the last rule in
    this chain */ ADDRTYPE match dst-type LOCAL

Chain KUBE-SVC-4N57TFCL4MD7ZTDA (1 references) ❷
target     prot opt source               destination
KUBE-SEP-VUJFIIOGYVVPH7Q4  all  --  anywhere    anywhere    /* default/nginx: */
    statistic mode random probability 0.33333333349
KUBE-SEP-Y42457KCQHG7FFWI  all  --  anywhere    anywhere    /* default/nginx: */
    statistic mode random probability 0.50000000000
KUBE-SEP-UOUQBAIW4Z676WKH  all  --  anywhere    anywhere    /* default/nginx: */

Chain KUBE-SEP-UOUQBAIW4Z676WKH (1 references)  ❸
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  10.244.0.8    anywhere             /* default/nginx: */
DNAT       tcp  --  anywhere           anywhere             /* default/nginx: */
    tcp to:10.244.0.8:80

Chain KUBE-SEP-VUJFIIOGYVVPH7Q4 (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  10.244.0.108    anywhere           /* default/nginx: */
DNAT       tcp  --  anywhere             anywhere           /* default/nginx: */
    tcp to:10.244.0.108:80

Chain KUBE-SEP-Y42457KCQHG7FFWI (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  10.244.0.6           anywhere     /* default/nginx: */
DNAT       tcp  --  anywhere             anywhere          /* default/nginx: */
    tcp to:10.244.0.6:80
```

❶ This is the top-level chain. It has rules for all the Services in the cluster. Notice how the KUBE-SVC-4N57TFCL4MD7ZTDA rule specifies a destination IP of 10.97.85.96. This is the nginx Service's ClusterIP.

❷ This is the chain of the nginx Service. Notice how there is a rule for each Service endpoint with a given probability of matching the rule.

❸ This chain corresponds to one of the Service endpoints. (SEP stands for Service endpoint.) The last rule in this chain is the one that performs DNAT to forward the packet to the endpoint (or Pod).

NodePort and LoadBalancer Services. When it comes to NodePort and LoadBalancer Services, kube-proxy configures iptables rules similar to those used for ClusterIP Services. The main difference is that the rules match packets based on their destination port number. If they match, the rule sends the packet to the Service-specific chain where DNAT happens. The snippet below shows the iptables rules for a NodePort Service called `nginx` listening on port 31767.

```
$ iptables --list --table nat
Chain KUBE-NODEPORTS (1 references) ❶
target     prot opt source               destination
KUBE-MARK-MASQ  tcp  --  anywhere             anywhere      /* default/nginx: */
    tcp dpt:31767
KUBE-SVC-4N57TFCL4MD7ZTDA  tcp  --  anywhere     anywhere   /* default/nginx: */
    tcp dpt:31767 ❷
```

❶ Kube-proxy programs iptables rules for NodePort Services in the `KUBE-NODEPORTS` chain.

❷ If the packet has `tcp: 31767` as the destination port, it is sent to the Service-specific chain. This chain is the Service-specific chain we saw in callout 2 in the previous code snippet.

In addition to programming the iptables rules, kube-proxy opens the port assigned to the NodePort Service and holds it open. Holding on to the port has no function from a routing perspective. It merely prevents other processes from claiming it.

A key consideration to make when using NodePort and LoadBalancer Services is the Service's external traffic policy setting. The external traffic policy determines whether the Service routes external traffic to node-local endpoints (`externalTrafficPolicy: Local`) or cluster-wide endpoints (`externalTrafficPolicy: Cluster`). Each policy has benefits and trade-offs, as discussed next.

When the policy is set to `Local`, the Service routes traffic to endpoints (Pods) running on the node receiving the traffic. Routing to a local endpoint has two important benefits. First, there is no SNAT involved so the source IP is preserved, making it available to the workload. And second, there is no additional network hop that you would otherwise incur when forwarding traffic to another node. With that said, the `Local` policy also has downsides. Mainly, traffic that reaches a node that lacks Service endpoints is dropped. For this reason, the `Local` policy is usually combined with an external load balancer that health-checks the nodes. When the node doesn't have an endpoint for the Service, the load balancer does not send traffic to the node, given

that the health check fails. Figure 6-6 illustrates this functionality. Another downside of the `Local` policy is the potential for unbalanced application load. For example, if a node has three Service endpoints, each endpoint receives 33% of the traffic. If another node has a single endpoint, it receives 100% of the traffic. This imbalance can be mitigated by spreading the Pods with anti-affinity rules or using a DaemonSet to schedule the Pods.

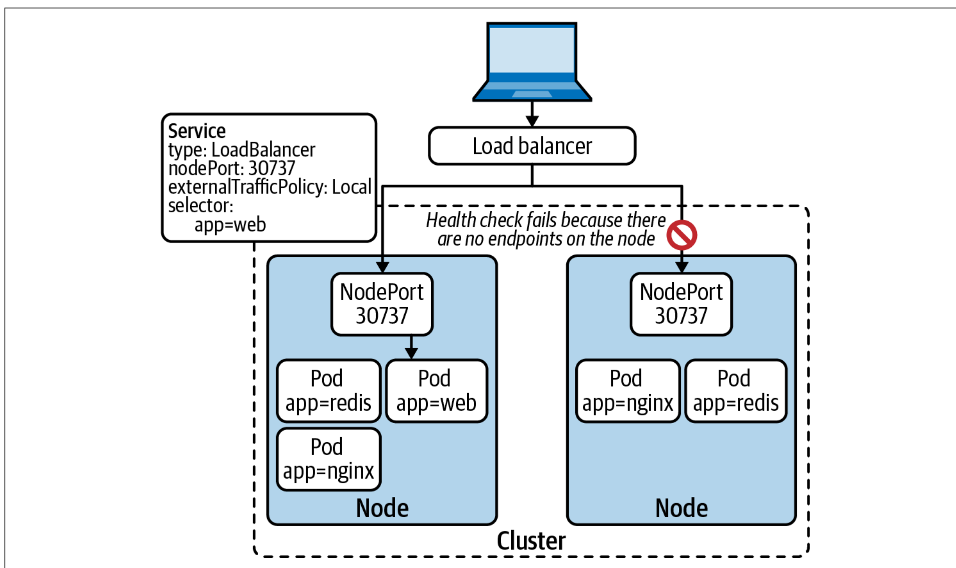


Figure 6-6. LoadBalancer Service with `Local` external traffic policy. The external load balancer runs health checks against the nodes. Any node that does not have a Service endpoint is removed from the load balancer's backend pool.

If you have a Service that handles a ton of external traffic, using the `Local` external policy is usually the right choice. However, if you do not have a load balancer at your disposal, you should use the `Cluster` external traffic policy. With this policy, traffic is load balanced across all endpoints in the cluster, as shown in Figure 6-7. As you can imagine, the load balancing results in the loss of the Source IP due to SNAT. It can also result in an additional network hop. However, the `Cluster` policy does not drop external traffic, regardless of where the endpoint Pods are running.

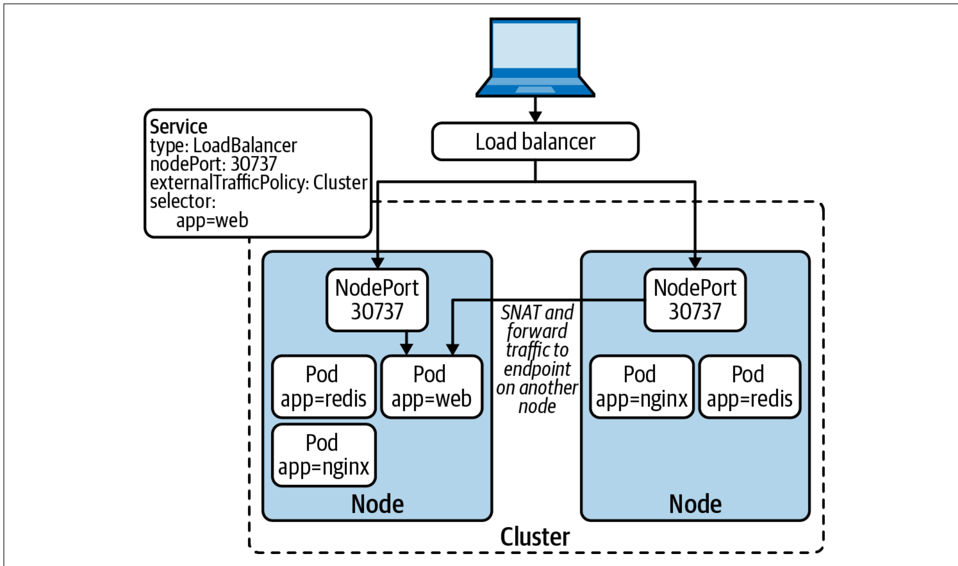


Figure 6-7. LoadBalancer Service with `Cluster` external traffic policy. Nodes that do not have node-local endpoints forward the traffic to an endpoint on another node.

Connection tracking (conntrack). When the kernel's networking stack performs DNAT on a packet destined to a Service, it adds an entry to the connection tracking (conntrack) table. The table tracks the translation performed so that it is applied to any additional packet destined to the same Service. The table is also used to remove the NAT from response packets before sending them to the source Pod.

Each entry in the table maps the pre-NAT protocol, source IP, source port, destination IP, and destination port onto the post-NAT protocol, source IP, source port, destination IP, and destination port. (Entries include additional information but are not relevant in this context.) Figure 6-8 depicts a table entry that tracks the connection from a Pod (`192.168.0.9`) to a Service (`10.96.0.14`). Notice how the destination IP and port change after the DNAT.

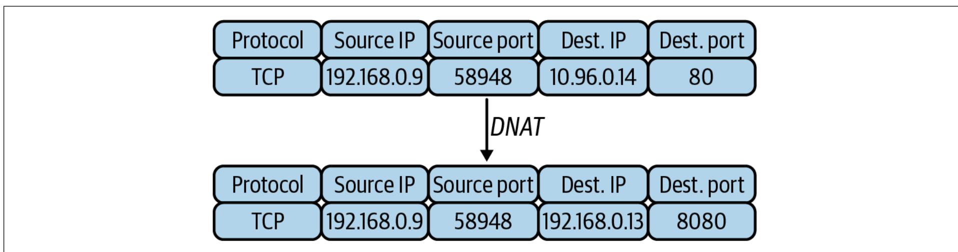
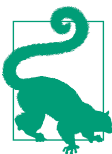


Figure 6-8. Connection tracking (conntrack) table entry that tracks the connection from a Pod (`192.168.0.9`) to a Service (`10.96.0.14`).



When the conntrack table fills up, the kernel starts dropping or rejecting connections, which can be problematic for some applications. If you are running workloads that handle many connections and notice connection issues, you may need to tune the maximum size of the conntrack table on your nodes. More importantly, you should monitor the conntrack table utilization and alert when the table is close to being full.

Masquerade. You may have noticed that we glossed over the `KUBE-MARK-MASQ` iptables rules listed in the previous examples. These rules are in place for packets that arrive at a node from outside the cluster. To route such packets properly, the Service fabric needs to masquerade/source NAT the packets when forwarding them to another node. Otherwise, response packets will contain the IP address of the Pod that handled the request. The Pod IP in the packet would cause a connection issue, as the client initiated the connection to the node and not the Pod.

Masquerading is also used to egress from the cluster. When Pods connect to external services, the source IP must be the IP address of the node where the Pod is running instead of the Pod IP. Otherwise, the network would drop response packets because they would have the Pod IP as the destination IP address.

Performance concerns. The iptables mode has served and continues to serve Kubernetes clusters well. With that said, you should be aware of some performance and scalability limitations, as these can arise in large cluster deployments.

Given the structure of the iptables rules and how they work, whenever a Pod establishes a new connection to a Service, the initial packet traverses the iptables rules until it matches one of them. In the worst-case scenario, the packet needs to traverse the entire collection of iptables rules.

The iptables mode suffers from O(n) time complexity when it processes packets. In other words, the iptables mode scales linearly with the number of Services in the cluster. As the number of Services grows, the performance of connecting to Services gets worse.

Perhaps more important, updates to the iptables rules also suffer at large scale. Because iptables rules are not incremental, kube-proxy needs to write out the entire table for every update. In some cases, these updates can even take minutes to complete, which risks sending traffic to stale endpoints. Furthermore, kube-proxy needs to hold the iptables lock (`/run/xtables.lock`) during these updates, which can cause contention with other processes that need to update the iptables rules, such as CNI plug-ins.

Linear scaling is an undesirable quality of any system. With that said, based on tests performed by the Kubernetes community, you should not notice any performance degradation unless you are running clusters with tens of thousands of Services. If you are operating at that scale, however, you might benefit from the IPVS mode in kube-proxy, which we'll discuss in the following section.

Rolling Updates and Service Reconciliation

An interesting problem with Services is unexpected request errors during application rolling updates. While this issue is less common in development environments, it can crop up in production clusters that are hosting many workloads.

The crux of the problem is the distributed nature of Kubernetes. As we've discussed in this chapter, multiple components work together to make Services work in Kubernetes, mainly the Endpoints controller and kube-proxy.

When a Pod is deleted, the following happens simultaneously:

- The kubelet initiates the Pod shutdown sequence. It sends a SIGTERM signal to the workload and waits until it terminates. If the process continues to run after the graceful shutdown period, the kubelet sends a SIGKILL to terminate the workload forcefully.
- The Endpoints controller receives a Pod deletion watch event, which triggers the removal of the Pod IP address from the Endpoints resource. Once the Endpoints resource is updated, kube-proxy removes the IP address from the iptables rules (or IPVS virtual service).

This is a classic distributed system race. Ideally, the Endpoints controller and kube-proxy finish their updates before the Pod exits. However, ordering is not guaranteed, given that these workflows are running concurrently. There is a chance that the workload exits (and thus stops accepting requests) before kube-proxy on each node removes the Pod from the list of active endpoints. When this happens, in-flight requests fail because they get routed to a Pod that is no longer running.

To solve this, Kubernetes would have to wait until all kube-proxies finish updating endpoints before stopping workloads, but this is not feasible. For example, how would it handle the case of a node becoming unavailable? With that said, we've used SIGTERM handlers and `sleep` pre-stop hooks to mitigate this issue in practice.

Kube-proxy: IP Virtual Server (IPVS) mode

IPVS is a load balancing technology built into the Linux kernel. Kubernetes added support for IPVS in kube-proxy to address the scalability limitations and performance issues of the iptables mode.

As discussed in the previous section, the iptables mode uses iptables rules to implement Kubernetes Services. The iptables rules are stored in a list, which packets need to traverse in its entirety in the worst-case scenario. IPVS does not suffer from this problem because it was originally designed for load balancing use cases.

The IPVS implementation in the Linux kernel uses hash tables to find the destination of a packet. Instead of traversing the list of Services when a new connection is established, IPVS immediately finds the destination Pod based on the Service IP address.

Let's discuss how kube-proxy in IPVS mode handles each of the Kubernetes Service types.

ClusterIP Services. When handling Services that have a ClusterIP, kube-proxy in `ipvs` mode does a couple of things. First, it adds the IP address of the ClusterIP Service to a dummy network interface on the node called `kube-ipvs0`, as shown in the following snippet:

```
$ ip address show dev kube-ipvs0
28: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN group default
    link/ether 96:96:1b:36:32:de brd ff:ff:ff:ff:ff:ff
    inet 10.110.34.183/32 brd 10.110.34.183 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
    inet 10.96.0.10/32 brd 10.96.0.10 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
    inet 10.96.0.1/32 brd 10.96.0.1 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
```

After updating the dummy interface, kube-proxy creates an IPVS virtual service with the IP address of the ClusterIP Service. Finally, for each Service endpoint, it adds an IPVS real server to the IPVS virtual service. The following snippet shows the IPVS virtual service and real servers for a ClusterIP Service with three endpoints:

```
$ ipvsadm --list --numeric --tcp-service 10.110.34.183:80
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.110.34.183:80 rr ❶
  -> 192.168.89.153:80            Masq    1      0          0 ❷
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
```

❶ This is the IPVS virtual service. Its IP address is the IP address of the ClusterIP Service.

❷ This is one of the IPVS real servers. It corresponds to one of the Service endpoints (Pods).

NodePort and LoadBalancer Services. For NodePort and LoadBalancer Services, kube-proxy creates an IPVS virtual service for the Service's cluster IP. Kube-proxy also

creates an IPVS virtual service for each of the node's IP addresses and the loopback address. For example, the following snippet shows a listing of the IPVS virtual services created for a NodePort Service listening on TCP port 30737:

```
ipvsadm --list --numeric
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.0.99.67:30737 rr ❶
  -> 192.168.89.153:80            Masq    1      0          0
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
TCP  10.110.34.183:80 rr ❷
  -> 192.168.89.153:80            Masq    1      0          0
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
TCP  127.0.0.1:30737 rr ❸
  -> 192.168.89.153:80            Masq    1      0          0
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
TCP  192.168.246.64:30737 rr ❹
  -> 192.168.89.153:80            Masq    1      0          0
  -> 192.168.89.154:80            Masq    1      0          0
  -> 192.168.89.155:80            Masq    1      0          0
```

❶ IPVS virtual service listening on the node's IP address.

❷ IPVS virtual service listening on the Service's cluster IP address.

❸ IPVS virtual service listening on `localhost`.

❹ IPVS virtual service listening on a secondary network interface on the node.

Running without kube-proxy

Historically, kube-proxy has been a staple in all Kubernetes deployments. It is a vital component that makes Kubernetes Services work. As the community evolves, however, we could start seeing Kubernetes deployments that do not have kube-proxy running. How is this possible? What handles Services instead?

With the advent of extended Berkeley Packet Filters (eBPF), CNI plug-ins such as Cilium and Calico can absorb kube-proxy's responsibilities. Instead of handling Services with iptables or IPVS, the CNI plug-ins program Services right into the Pod networking data plane. Using eBPF improves the performance and scalability of Services in Kubernetes, given that the eBPF implementation uses hash tables for endpoint lookups. It also improves Service update processing, as it can handle individual Service updates efficiently.

Removing the need for kube-proxy and optimizing Service routing is a worthy feat, especially for those operating at scale. However, it is still early days when it comes to running these solutions in production. For example, the Cilium implementation requires newer kernel versions to support a kube-proxy-less deployment (at the time of writing, the latest Cilium version is `v1.8`). Similarly, the Calico team discourages the use of eBPF in production because it is still in tech preview. (At the time of writing, the latest calico version is `v3.15.1`.) Over time, we expect to see kube-proxy replacements become more common. Cilium even supports running its proxy replacement capabilities alongside other CNI plug-ins (referred to as CNI chaining).

Service Discovery

Service discovery provides a mechanism for applications to discover services that are available on the network. While not a *routing* concern, service discovery is intimately related to Kubernetes Services.

Platform teams may wonder whether they need to introduce a dedicated service discovery system to a cluster, such as Consul. While possible, it is typically not necessary, as Kubernetes offers service discovery to all workloads running in the cluster. In this section, we will discuss the different service discovery mechanisms available in Kubernetes: DNS-based service discovery, API-based service discovery, and environment variable-based service discovery.

Using DNS

Kubernetes provides service discovery over DNS to workloads running inside the cluster. Conformant Kubernetes deployments run a DNS server that integrates with the Kubernetes API. The most common DNS server in use today is CoreDNS, an open source, extensible DNS server.

CoreDNS watches resources in the Kubernetes API server. For each Kubernetes Service, CoreDNS creates a DNS record with the following format: `<service-name>.<namespace-name>.svc.cluster.local`. For example, a Service called `nginx` in the `default` Namespace gets the DNS record `nginx.default.svc.cluster.local`. But how can Pods use these DNS records?

To enable DNS-based service discovery, Kubernetes configures CoreDNS as the DNS resolver for Pods. When setting up a Pod's sandbox, the kubelet writes an */etc/resolv.conf* that specifies CoreDNS as the nameserver and injects the config file into the container. The */etc/resolv.conf* file of a Pod looks something like this:

```
$ cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

Given this configuration, Pods send DNS queries to CoreDNS whenever they try to connect to a Service by name.

Another interesting trick in the resolver configuration is the use of `ndots` and `search` to simplify DNS queries. When a Pod wants to reach a Service that exists in the same Namespace, it can use the Service's name as the domain name instead of the fully qualified domain name (`nginx.default.svc.cluster.local`):

```
$ nslookup nginx
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   nginx.default.svc.cluster.local
Address: 10.110.34.183
```

Similarly, when a Pod wants to reach a Service in another Namespace, it can do so by appending the Namespace name to the Service name:

```
$ nslookup nginx.default
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   nginx.default.svc.cluster.local
Address: 10.110.34.183
```

One thing to consider with the `ndots` configuration is its impact on applications that communicate with services outside of the cluster. The `ndots` parameter specifies how many dots must appear in a domain name for it to be considered an absolute or fully qualified name. When resolving a name that's not fully qualified, the system attempts various lookups using the items in the `search` parameter, as seen in the following example. Thus, when applications resolve cluster-external names that are not fully qualified, the resolver consults the cluster DNS server with multiple futile requests before attempting to resolve the name as an absolute name. To avoid this issue, you can use fully qualified domain names in your applications by adding a `.` at the end of the name. Alternatively, you can tune the DNS configuration of the Pod via the `dnsConfig` field in the Pod's specification.

The following snippet shows the impact of the `ndots` configuration on Pods that resolve external names. Notice how resolving a name that has less dots than the configured `ndots` results in multiple DNS queries, while resolving an absolute name results in a single query:

```
$ nslookup -type=A google.com -debug | grep QUESTIONS -A1 ❶
  QUESTIONS:
google.com.default.svc.cluster.local, type = A, class = IN
--
  QUESTIONS:
google.com.svc.cluster.local, type = A, class = IN
--
```

```
  QUESTIONS:
google.com.cluster.local, type = A, class = IN
--
  QUESTIONS:
google.com, type = A, class = IN

$ nslookup -type=A -debug google.com. | grep QUESTIONS -A1 ❷
  QUESTIONS:
google.com, type = A, class = IN
```

❶ Attempt to resolve a name with less than 5 dots (not fully qualified). The resolver performs multiple lookups, one per item in the `search` field of */etc/resolv.conf*.

❷ Attempt to resolve a fully qualified name. The resolver performs a single lookup.

Overall, service discovery over DNS is extremely useful, as it lowers the barrier for applications to interact with Kubernetes Services.

Using the Kubernetes API

Another way to discover Services in Kubernetes is by using the Kubernetes API. The community maintains various Kubernetes client libraries in different languages, including Go, Java, Python, and others. Some application frameworks, such as Spring, also support service discovery through the Kubernetes API.

Using the Kubernetes API for service discovery can be useful in specific scenarios. For example, if your applications need to be aware of Service endpoint changes as soon as they happen, they would benefit from watching the API.

The main downside of performing service discovery through the Kubernetes API is that you tightly couple the application to the underlying platform. Ideally, applications should be unaware of the platform. If you do choose to use the Kubernetes API for service discovery, consider building an interface that abstracts the Kubernetes details away from your business logic.

Using environment variables

Kubernetes injects environment variables into Pods to facilitate service discovery. For each Service, Kubernetes sets multiple environment variables according to the Service definition. The environment variables for an `nginx` ClusterIP Service listening on port 80 look as follows:

```
NGINX_PORT_80_TCP_PORT=80
NGINX_SERVICE_HOST=10.110.34.183
NGINX_PORT=tcp://10.110.34.183:80
NGINX_PORT_80_TCP=tcp://10.110.34.183:80
NGINX_PORT_80_TCP_PROTO=tcp
NGINX_SERVICE_PORT=80
NGINX_PORT_80_TCP_ADDR=10.110.34.183
```

The downside to this approach is that environment variables cannot be updated without restarting the Pod. Thus, Services must be in place before the Pod starts up.

DNS Service Performance

As mentioned in the previous section, offering DNS-based service discovery to workloads on your platform is crucial. As the size of your cluster and number of applications grows, the DNS service can become a bottleneck. In this section, we will discuss techniques you can use to provide a performant DNS service.

DNS cache on each node

The Kubernetes community maintains a DNS cache add-on called NodeLocal DNSCache. The add-on runs a DNS cache on each node to address multiple problems. First, the cache reduces the latency of DNS lookups, given that workloads get their answers from the local cache (assuming a cache hit) instead of reaching out to the DNS server (potentially on another node). Second, the load on the CoreDNS servers goes down, as workloads are leveraging the cache most of the time. Finally, in the case of a cache miss, the local DNS cache upgrades the DNS query to TCP when reaching out to the central DNS service. Using TCP instead of UDP improves the reliability of the DNS query.

The DNS cache runs as a DaemonSet on the cluster. Each replica of the DNS cache intercepts the DNS queries that originate from their node. There's no need to change application code or configuration to use the cache. The node-level architecture of the NodeLocal DNSCache add-on is depicted in Figure 6-9.

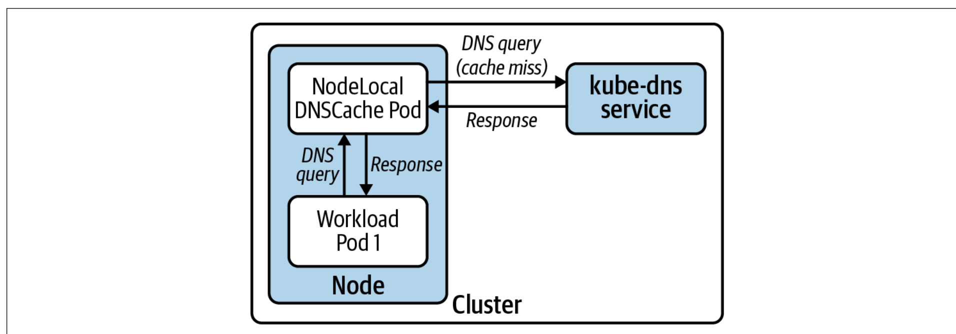


Figure 6-9. Node-level architecture of the NodeLocal DNSCache add-on. The DNS cache intercepts DNS queries and responds immediately if there's a cache hit. In the case of a cache miss, the DNS cache forwards the query to the cluster DNS service.

Auto-scaling the DNS server deployment

In addition to running the node-local DNS cache in your cluster, you can automatically scale the DNS Deployment according to the size of the cluster. Note that this strategy does not leverage the Horizontal Pod Autoscaler. Instead, it uses the cluster Proportional Autoscaler, which scales workloads based on the number of nodes in the cluster.

The Cluster Proportional Autoscaler runs as a Pod in the cluster. It has a configuration flag to set the workload that needs autoscaling. To autoscale DNS, you must set the target flag to the CoreDNS (or kube-dns) Deployment. Once running, the autoscaler polls the API server every 10 seconds (by default) to get the number of nodes and CPU cores in the cluster. Then, it adjusts the number of replicas in the CoreDNS Deployment if necessary. The desired number of replicas is governed by a configurable replicas-to-nodes ratio or replicas-to-cores ratio. The ratios to use depend on your workloads and how DNS-intensive they are.

In most cases, using node-local DNS cache is sufficient to offer a reliable DNS service. However, autoscaling DNS is another strategy you can use when autoscaling clusters with a wide-enough range of minimum and maximum nodes.

Ingress

As we've discussed in Chapter 5, workloads running in Kubernetes are typically not accessible from outside the cluster. This is not a problem if your applications do not have external clients. Batch workloads are a great example of such applications. Realistically, however, most Kubernetes deployments host web services that do have end users.

Ingress is an approach to exposing services running in Kubernetes to clients outside of the cluster. Even though Kubernetes does not fulfill the Ingress API out of the box, it is a staple in any Kubernetes-based platform. It is not uncommon for off-the-shelf Kubernetes applications and cluster add-ons to expect that an Ingress controller is running in the cluster. Moreover, your developers will need it to be able to run their applications successfully in Kubernetes.

This section aims to guide you through the considerations you must make when implementing Ingress in your platform. We will review the Ingress API, the most common ingress traffic patterns that you will encounter, and the crucial role of Ingress controllers in a Kubernetes-based platform. We will also discuss different ways to deploy Ingress controllers and their trade-offs. Finally, we will address common challenges you can run into and explore helpful integrations with other tools in the ecosystem.

The Case for Ingress

Kubernetes Services already provide ways to route traffic to Pods, so why would you need an additional strategy to achieve the same thing? As much as we are fans of keeping platforms simple, the reality is that Services have important limitations and downsides:

Limited routing capabilities
: Services route traffic according to the destination IP and port of incoming requests. This can be useful for small and relatively simple applications, but it quickly breaks down for more substantial, microservices-based applications. These kinds of applications require smarter routing features and other advanced capabilities.

Cost
: If you are running in a cloud environment, each LoadBalancer Service in your cluster creates an external load balancer, such as an ELB in the case of AWS. Running a separate load balancer for each Service in your platform can quickly become cost-prohibitive.

Ingress addresses both these limitations. Instead of being limited to load balancing at layer 3/4 of the OSI model, Ingress provides load balancing and routing capabilities at layer 7. In other words, Ingress operates at the application layer, which results in more advanced routing features.

Another benefit of Ingress is that it removes the need to have multiple load balancers or entry points into the platform. Because of the advanced routing capabilities available in Ingress, such as the ability to route HTTP requests based on the `Host` header, you can route all the service traffic to a single entry point and let the Ingress controller take care of demultiplexing the traffic. This dramatically reduces the cost of bringing traffic into your platform.

The ability to have a single ingress point into the platform also reduces the complexity of noncloud deployments. Instead of potentially having to manage multiple external load balancers with a multitude of NodePort Services, you can operate a single external load balancer that routes traffic to the Ingress controllers.

Even though Ingress solves most of the downsides related to Kubernetes Services, the latter are still needed. Ingress controllers themselves run inside the platform and thus need to be exposed to clients that exist outside. And you can use a Service (either a NodePort or LoadBalancer) to do so. Besides, most Ingress controllers shine when it comes to load balancing HTTP traffic. If you want to be able to host applications that use other protocols, you might have to use Services alongside Ingress, depending on the capabilities of your Ingress controller.

The Ingress API

The Ingress API enables application teams to expose their services and configure request routing according to their needs. Because of Ingress's focus on HTTP routing, the Ingress API resource provides different ways to route traffic according to the properties of incoming HTTP requests.

A common routing technique is routing traffic according to the `Host` header of HTTP requests. For example, given the following Ingress configuration, HTTP requests with the `Host` header set to `bookhotels.com` are routed to one service, while requests destined to `bookflights.com` are routed to another:

```yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hotels-ingress
spec:
  rules:
  - host: bookhotels.com
    http:
      paths:
      - path: /
        backend:
          serviceName: hotels
          servicePort: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flights-ingress
spec:
  rules:
  - host: bookflights.com
    http:
      paths:
      - path: /
        backend:
          serviceName: flights
          servicePort: 80
```

Hosting applications on specific subdomains of a cluster-wide domain name is a common approach we encounter in Kubernetes. In this case, you assign a domain name to the platform, and each application gets a subdomain. Keeping with the travel theme in the previous example, an example of subdomain-based routing for a travel booking application could have `hotels.cluster1.useast.example.com` and `flights.cluster1.useast.example.com`. Subdomain-based routing is one of the best strategies you can employ. It also enables other interesting use cases, such as hosting tenants of a software-as-a-service (SaaS) application on tenant-specific

domain names (`tenantA.example.com` and `tenantB.example.com`, for example). We will further discuss how to implement subdomain-based routing in a later section.

Ingress Configuration Collisions and How to Avoid Them

The Ingress API is prone to configuration collisions in multiteam or multitenant clusters. The primary example is different teams trying to use the same domain name to expose their applications. Consider a scenario where an application team creates an Ingress resource with the host set to `app.bearcanoe.com`. What happens when another team creates an Ingress with the same host? The Ingress API does not specify how to handle this scenario. Instead, it is up to the Ingress controller to decide what to do. Some controllers merge the configuration when possible, while others reject the new Ingress resource and log an error message. In any case, overlapping Ingress resources can result in surprising behavior, and even outages!

Usually, we tackle this problem in one of two ways. The first is using an admission controller that validates the incoming Ingress resource and ensures the hostname is unique across the cluster. We've built many of these admission controllers over time while working in the field. These days, we use the Open Policy Agent (OPA) to handle this concern. The OPA community even maintains a policy for this use case.

The Contour Ingress controller approaches this with a different solution. The HTTPProxy Custom Resource handles this use case with *root* HTTPProxy resources and *inclusion*. In short, a root HTTPProxy specifies the host and then *includes* other HTTPProxy resources that are hosted under that domain. The idea is that the operator manages root HTTPProxy resources and assigns them to specific teams. For example, the operator would create a root HTTPProxy with the host `app1.bearcanoe.com` and *include* all HTTPProxy resources in the `app1` Namespace. See Contour's documentation for more details.

The Ingress API supports features beyond host-based routing. Through the evolution of the Kubernetes project, Ingress controllers extended the Ingress API. Unfortunately, these extensions were made using annotations instead of evolving the Ingress resource. The problem with using annotations is that they don't have a schema. This can result in a poor user experience, as there is no way for the API server to catch misconfigurations. To address this issue, some Ingress controllers provide Custom Resource Definitions (CRDs). These resources have well-defined APIs offering features otherwise not available through Ingress. Contour, for example, provides an `HTTPProxy` custom resource. While leveraging these CRDs gives you access to a broader array of features, you give up the ability to swap Ingress controllers if necessary. In other words, you "lock" yourself into a specific controller.

Ingress Controllers and How They Work

If you can recall the first time you played with Kubernetes, you probably ran into a puzzling scenario with Ingress. You downloaded a bunch of sample YAML files that included a Deployment and an Ingress and applied them to your cluster. You noticed that the Pod came up just fine, but you were not able to reach it. The Ingress resource was essentially doing nothing. You probably wondered, What's going on here?

Ingress is one of those APIs in Kubernetes that are left to the platform builder to implement. In other words, Kubernetes exposes the Ingress interface and expects another component to provide the implementation. This component is commonly called an *Ingress controller*.

An Ingress controller is a platform component that runs in the cluster. The controller is responsible for watching the Ingress API and acting according to the configuration defined in Ingress resources. In most implementations, the Ingress controller is paired with a reverse proxy, such as NGINX or Envoy. This two-component architecture is comparable to other software-defined networking systems, in that the controller is the *control plane* of the Ingress controller, while the proxy is the *data plane* component. Figure 6-10 shows the control plane and data plane of an Ingress controller.

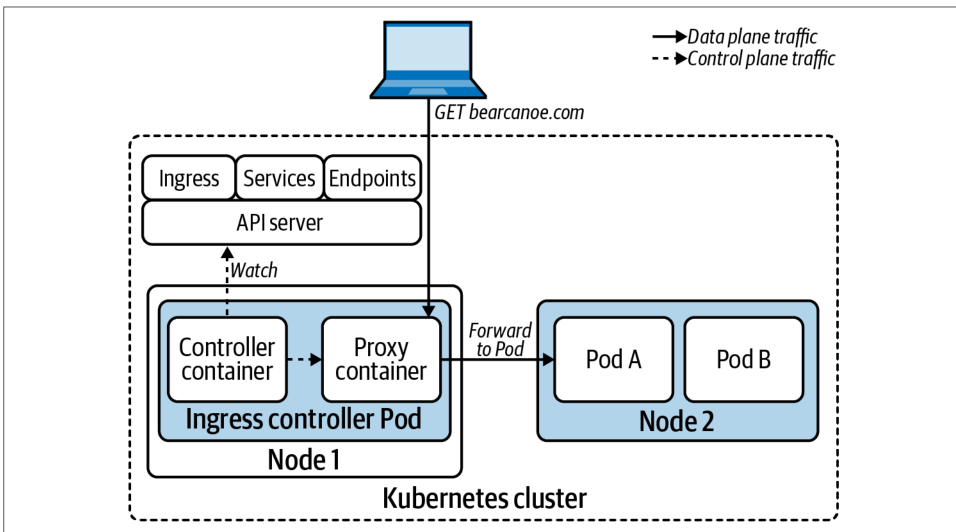


Figure 6-10. The Ingress controller watches various resources in the API server and configures the proxy accordingly. The proxy handles incoming traffic and forwards it to Pods, according to the Ingress configuration.

The control plane of the Ingress controller connects to the Kubernetes API and watches a variety of resources, such as Ingress, Services, Endpoints, and others.

Whenever these resources change, the controller receives a watch notification and configures the data plane to act according to the desired state declared in the Kubernetes API.

The data plane handles the routing and load balancing of network traffic. As mentioned before, the data plane is usually implemented with an off-the-shelf proxy.

Because the Ingress API builds on top of the Service abstraction, Ingress controllers have a choice between forwarding traffic through Services or sending it directly to Pods. Most Ingress controllers opt for the latter. They don't use the Service resource, other than to validate that the Service referenced in the Ingress resource exists. When it comes to routing, most controllers forward traffic to the Pod IP addresses listed in the corresponding Endpoints object. Routing traffic directly to the Pod bypasses the Service layer, which reduces latency and adds different load balancing strategies.

Ingress Traffic Patterns

A great aspect of Ingress is that each application gets to configure routing according to its needs. Typically, each application has different requirements when it comes to handling incoming traffic. Some might require TLS termination at the edge. Some might want to handle TLS themselves, while others might not support TLS at all (hopefully, this is not the case).

In this section, we will explore the common ingress traffic patterns that we have encountered. This should give you an idea of what Ingress can provide to your developers and how Ingress can fit into your platform offering.

HTTP proxying

HTTP proxying is the bread-and-butter of Ingress. This pattern involves exposing one or more HTTP-based services and routing traffic according to the HTTP requests' properties. We have already discussed routing based on the `Host` header. Other properties that can influence routing decisions include the URL path, the request method, request headers, and more, depending on the Ingress controller.

The following Ingress resource exposes the `app1` Service at `app1.example.com`. Any incoming request that has a matching `Host` HTTP header is sent to an `app1` Pod.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app1
spec:
  rules:
  - host: app1.example.com
    http:
      paths:
      - path: /
```

```
          backend:
            serviceName: app1
            servicePort: 80
```

Once applied, the preceding configuration results in the data plane flow depicted in Figure 6-11.

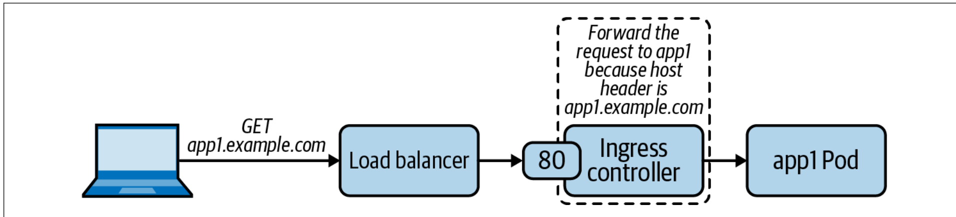


Figure 6-11. Path of an HTTP request from the client to the target Pod through the Ingress controller.

HTTP proxying with TLS

Supporting TLS encryption is table-stakes for Ingress controllers. This ingress traffic pattern is the same as HTTP proxying from a routing perspective. However, clients communicate with the Ingress controller over a secure TLS connection instead of plain-text HTTP.

The following example shows an Ingress resource that exposes `app1` with TLS. The controller gets the TLS serving certificate and key from the referenced Kubernetes Secret.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app1
spec:
  tls:
  - hosts:
      - app1.example.com
    secretName: app1-tls-cert
  rules:
  - host: app1.example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: app1
          servicePort: 443
```

Ingress controllers support different configurations when it comes to the connection between the Ingress controller and the backend service. The connection between the external client and the controller is secure (TLS), while the connection between the Ingress controller and the backend application does not have to be. Whether the

connection between the controller and the backend is secure depends on whether the application is listening for TLS connections. By default, most Ingress controllers terminate TLS and forward requests over an unencrypted connection, as depicted in Figure 6-12.

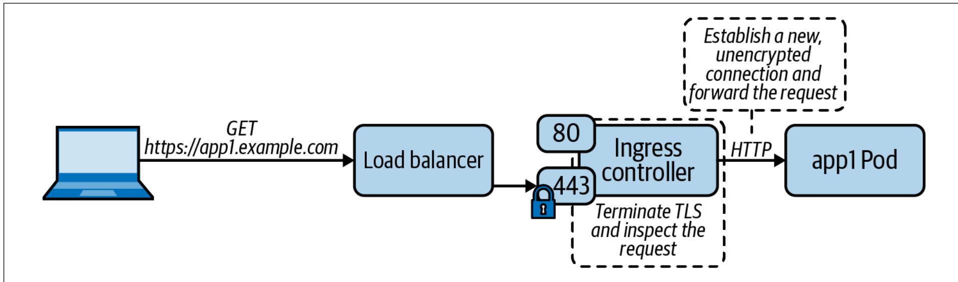


Figure 6-12. Ingress controller handling an HTTPS request by terminating TLS and forwarding the request to the backend Pod over an unencrypted connection.

In the case where a secure connection to the backend is required, the Ingress controller terminates the TLS connection at the edge and establishes a new TLS connection with the backend (illustrated in Figure 6-13). The reestablishment of the TLS connection is sometimes not appropriate for certain applications, such as those that need to perform the TLS handshake with their clients. In these situations, TLS passthrough, which we will discuss further later, is a viable alternative.

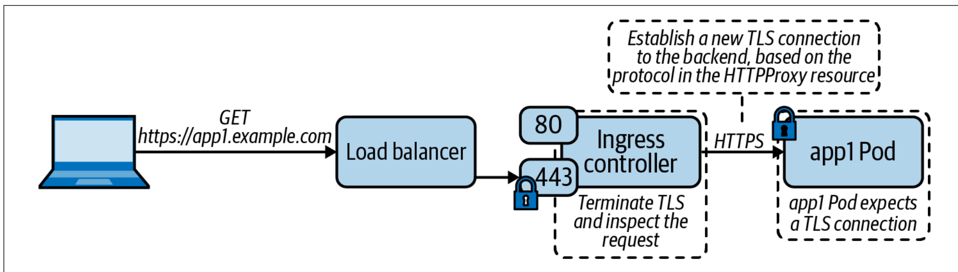


Figure 6-13. Ingress controller terminating TLS and establishing a new TLS connection with the backend Pod when handling HTTPS requests.

Layer 3/4 proxying

Even though the Ingress API's primary focus is layer 7 proxying (HTTP traffic), some Ingress controllers can proxy traffic at layer 3/4 (TCP/UDP traffic). This can be useful if you need to expose applications that do not speak HTTP. When evaluating Ingress controllers, you must keep this in mind, as support for layer 3/4 proxying varies across controllers.

The main challenge with proxying TCP or UDP services is that Ingress controllers listen on a limited number of ports, usually 80 and 443. As you can imagine, exposing

different TCP or UDP services on the same port is impossible without a strategy to distinguish the traffic. Ingress controllers solve this problem in different ways. Some, such as Contour, support proxying of only TLS encrypted TCP connections that use the Server Name Indication (SNI) TLS extension. The reason for this is that Contour needs to know where the traffic is headed. And when using SNI, the target domain name is available (unencrypted) in the ClientHello message of the TLS handshake. Because TLS and SNI are dependent on TCP, Contour does not support UDP proxying.

The following is a sample HTTPProxy Custom Resource, which is supported by Contour. Layer 3/4 proxying is one of those cases where a Custom Resource provides a better experience than the Ingress API:

```
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: tcp-proxy
spec:
  virtualhost:
    fqdn: tcp.bearcanoe.com
    tls:
      secretName: secret
  tcpproxy:
    services:
    - name: tcp-app
      port: 8080
```

With the preceding configuration, Contour reads the server name in the SNI extension and proxies the traffic to the backend TCP service. Figure 6-14 illustrates this capability.

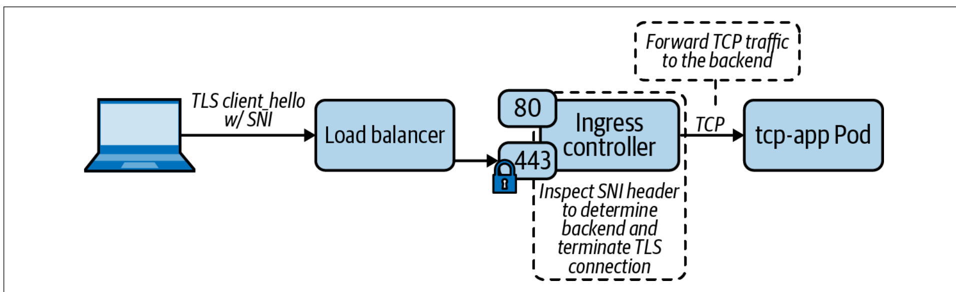


Figure 6-14. The Ingress controller inspects the SNI header to determine the backend, terminates the TLS connection, and forwards the TCP traffic to the Pod.

Other Ingress controllers expose configuration parameters that you can use to tell the underlying proxy to bind additional ports for layer 3/4 proxying. You then map these additional ports to specific services running in the cluster. This is the approach that the community-led NGINX Ingress controller takes for layer 3/4 proxying.

A common use case of layer 3/4 proxying is TLS passthrough. TLS passthrough involves an application that exposes a TLS endpoint and the need to handle the TLS handshake directly with the client. As we discussed in the "HTTP proxying with TLS" pattern, the Ingress controller usually terminates the client-facing TLS connection. The TLS termination is necessary so that the Ingress controller can inspect the HTTP request, which would otherwise be encrypted. However, with TLS passthrough, the Ingress controller does not terminate TLS and instead proxies the secure connection to a backend Pod. Figure 6-15 depicts TLS passthrough.

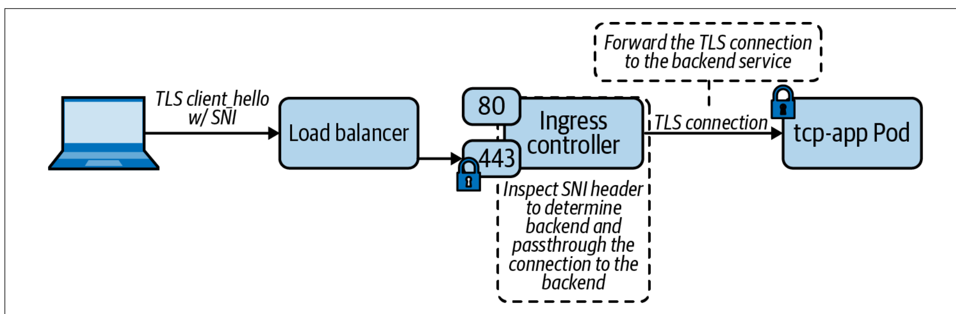


Figure 6-15. When TLS passthrough is enabled, the Ingress controller inspects the SNI header to determine the backend and forwards the TLS connection accordingly.

Choosing an Ingress Controller

There are several Ingress controllers that you can choose from. In our experience, the NGINX Ingress controller is one of the most commonly used. However, that does not mean it is best for your application platform. Other choices include Contour, HAProxy, Traefik, and more. In keeping with this book's theme, our goal is not to tell you which to use. Instead, we aim to equip you with the information you need to make this decision. We will also highlight significant trade-offs where applicable.

Stepping back a bit, the primary goal of an Ingress controller is to handle application traffic. Thus, it is natural to turn to the applications as the primary factor when selecting an Ingress controller. Specifically, what are the features and requirements that your applications need? The following is a list of criteria that you can use to evaluate Ingress controllers from an application support perspective:

- Do applications expose HTTPS endpoints? Do they need to handle the TLS handshake with the client directly, or is it okay to terminate TLS at the edge?
- What SSL ciphers do the applications use?
- Do applications need session affinity or sticky sessions?

- Do applications need advanced request routing capabilities, such as HTTP header-based routing, cookie-based routing, HTTP method-based routing, and others?
- Do applications have different load balancing algorithm requirements, such as round-robin, weighted least request, or random?
- Do applications need support for Cross-Origin Resource Sharing (CORS)?
- Do applications offload authentication concerns to an external system? Some Ingress controllers provide authentication features that you can leverage to provide a common authentication mechanism across applications.
- Are there any applications that need to expose TCP or UDP endpoints?
- Does the application need the ability to rate-limit incoming traffic?

In addition to application requirements, another crucial consideration to make is your organization's experience with the data plane technology. If you are already intimately familiar with a specific proxy, it is usually a safe bet to start there. You will already have a good understanding of how it works, and more importantly, you will know its limitations and how to troubleshoot it.

Supportability is another critical factor to consider. Ingress is an essential component of your platform. It exists right in the middle of your customers and the services they are trying to reach. When things go wrong with your Ingress controller, you want to have access to the support you need when facing an outage.

Finally, remember that you can run multiple Ingress controllers in your platform using Ingress classes. Doing so increases the complexity and management of your platform, but it is necessary in some cases. The higher the adoption of your platform and the more production workloads you are running, the more features they will demand from your Ingress tier. It is entirely possible that you will end up having a set of requirements that cannot be fulfilled with a single Ingress controller.

Ingress Controller Deployment Considerations

Regardless of the Ingress controller, there is a set of considerations that you should keep in mind when it comes to deploying and operating the Ingress tier. Some of these considerations can also have an impact on the applications running on the platform.

Dedicated Ingress nodes

Dedicating (or reserving) a set of nodes to run the Ingress controller and thus serve as the cluster's "edge" is a pattern that we have found very successful. Figure 6-16 illustrates this deployment pattern. At first, it might seem wasteful to use dedicated ingress nodes. However, our philosophy is, if you can afford to run dedicated control

plane nodes, you can probably afford to dedicate nodes to the layer that is in the critical path for all workloads on the cluster. Using a dedicated node pool for Ingress brings considerable benefits.

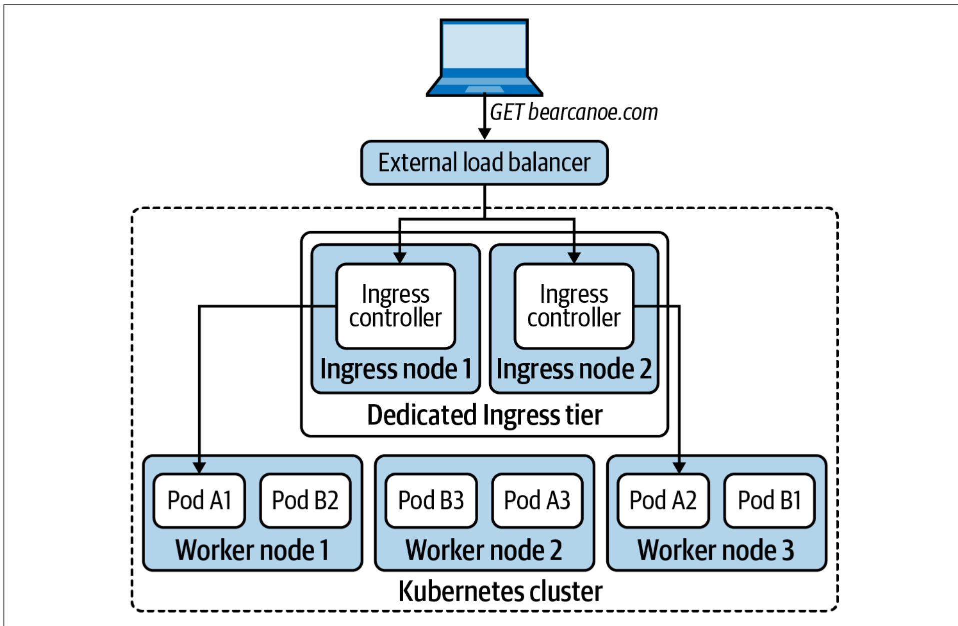


Figure 6-16. Dedicated ingress nodes are reserved for the Ingress controller. The ingress nodes serve as the "edge" of the cluster or the Ingress tier.

The primary benefit is resource isolation. Even though Kubernetes has support for configuring resource requests and limits, we have found that platform teams can struggle with getting those parameters right. This is especially true when the platform team is at the beginning of their Kubernetes journey and is unaware of the implementation details that underpin resource management (e.g., the Completely Fair Scheduler, cgroups). Furthermore, at the time of writing, Kubernetes does not support resource isolation for network I/O or file descriptors, making it challenging to guarantee the fair sharing of these resources.

Another reason for running Ingress controllers on dedicated nodes is compliance. We have encountered that a large number of organizations have pre-established firewall rules and other security practices that can be incompatible with Ingress controllers. Dedicated ingress nodes are useful in these environments, as it is typically easier to get exceptions for a subset of cluster nodes instead of all of them.

Finally, limiting the number of nodes that run the Ingress controller can be helpful in bare-metal or on-premises installations. In such deployments, the Ingress tier is fronted by a hardware load balancer. In most cases, these are traditional load balancers

that lack APIs and must be statically configured to route traffic to a specific set of backends. Having a small number of ingress nodes eases the configuration and management of these external load balancers.

Overall, dedicating nodes to Ingress can help with performance, compliance, and managing external load balancers. The best approach to implement dedicated ingress nodes is to label and taint the ingress nodes. Then, deploy the Ingress controller as a DaemonSet that (1) tolerates the taint, and (2) has a node selector that targets the ingress nodes. With this approach, ingress node failures must be accounted for, as Ingress controllers will not run on nodes other than those reserved for Ingress. In the ideal case, failed nodes are automatically replaced with new nodes that can continue handling Ingress traffic.

Binding to the host network

To optimize the ingress traffic path, you can bind your Ingress controller to the underlying host's network. By doing so, incoming requests bypass the Kubernetes Service fabric and reach the Ingress controller directly. When enabling host networking, ensure that the Ingress controller's DNS policy is set to `ClusterFirstWithHostNet`. The following snippet shows the host networking and DNS policy settings in a Pod template:

```
spec:
  containers:
  - image: nginx
    name: nginx
  dnsPolicy: ClusterFirstWithHostNet
  hostNetwork: true
```

While running the Ingress controller directly on the host network can increase performance, you must keep in mind that doing so removes the network namespace boundary between the Ingress controller and the node. In other words, the Ingress controller has full access to all network interfaces and network services available on the host. This has implications on the Ingress controller's threat model. Namely, it lowers the bar for an adversary to perform lateral movement in the case of a data plane proxy vulnerability. Additionally, attaching to the host network is a privileged operation. Thus, the Ingress controller needs elevated privileges or exceptions to run as a privileged workload.

Even then, we've found that binding to the host network is worth the trade-off and is usually the best way to expose the platform's Ingress controllers. The ingress traffic arrives directly at the controller's gate, instead of traversing the Service stack (which can be suboptimal, as discussed in "Kubernetes Services" on page 128).

Ingress controllers and external traffic policy

Unless configured properly, using a Kubernetes Service to expose the Ingress controller impacts the performance of the Ingress data plane.

If you recall from "Kubernetes Services" on page 128, a Service's external traffic policy determines how to handle traffic that's coming from outside the cluster. If you are using a NodePort or LoadBalancer Service to expose the Ingress controller, ensure that you set the external traffic policy to `Local`.

Using the `Local` policy avoids unnecessary network hops, as the external traffic reaches the local Ingress controller instead of hopping to another node. Furthermore, the `Local` policy doesn't use SNAT, which means the client IP address is visible to applications handling the requests.

Spread Ingress controllers across failure domains

To ensure the high-availability of your Ingress controller fleet, use Pod anti-affinity rules to spread the Ingress controllers across different failure domains.

DNS and Its Role in Ingress

As we have discussed in this chapter, applications running on the platform share the ingress data plane, and thus share that single entry point into the platform's network. As requests come in, the Ingress controller's primary responsibility is to disambiguate traffic and route it according to the Ingress configuration.

One of the primary ways to determine the destination of a request is by the target hostname (the `Host` header in the case of HTTP or SNI in the case of TCP), turning DNS into an essential player of your Ingress implementation. We will discuss two of the main approaches that are available when it comes to DNS and Ingress.

Wildcard DNS record

One of the most successful patterns we continuously use is to assign a domain name to the environment and slice it up by assigning subdomains to different applications. We sometimes call this "subdomain-based routing." The implementation of this pattern involves creating a wildcard DNS record (e.g., `*.bearcanoe.com`) that resolves to the Ingress tier of the cluster. Typically, this is a load balancer that is in front of the Ingress controllers.

There are several benefits to using a wildcard DNS record for your Ingress controllers:

- Applications can use any path under their subdomain, including the root path (`/`). Developers don't have to spend engineering hours to make their apps work

on subpaths. In some cases, applications expect to be hosted at the root path and do not work otherwise.

- The DNS implementation is relatively straightforward. There is no integration necessary between Kubernetes and your DNS provider.
- The single wildcard DNS record removes DNS propagation concerns that could arise when using different domain names for each application.

Kubernetes and DNS integration

An alternative to using a wildcard DNS record is to integrate your platform with your DNS provider. The Kubernetes community maintains a controller that offers this integration called external-dns. If you are using a DNS provider that is supported, consider using this controller to automate the creation of domain names.

As you might expect from a Kubernetes controller, external-dns continuously reconciles the DNS records in your upstream DNS provider and the configuration defined in Ingress resources. In other words, external-dns creates, updates, and deletes DNS records according to changes that happen in the Ingress API. External-dns needs two pieces of information to configure the DNS records, both of which are part of the Ingress resource: the desired hostname, which is in the Ingress specification, and the target IP address, which is available in the status field of the Ingress resource.

Integrating the platform with your DNS provider can be useful if you need to support multiple domain names. The controller takes care of automatically creating DNS records as needed. However, it is important to keep the following trade-offs in mind:

- You have to deploy an additional component (external-dns) into your cluster. An additional add-on brings about more complexity into your deployments, given that you have to operate, maintain, monitor, version, and upgrade one more component in your platform.
- If external-dns does not support your DNS provider, you have to develop your own controller. Building and maintaining a controller requires engineering effort that could be spent on higher-value efforts. In these situations, it is best to simply implement a wildcard DNS record.

Handling TLS Certificates

Ingress controllers need certificates and their corresponding private keys to serve applications over TLS. Depending on your Ingress strategy, managing certificates can be cumbersome. If your cluster hosts a single domain name and implements subdomain-based routing, you can use a single wildcard TLS certificate. In some cases, however, clusters host applications across a variety of domains, making it challenging to manage certificates efficiently. Furthermore, your security team might frown upon the usage of wildcard certificates. In any case, the Kubernetes community

has rallied around a certificate management add-on that eases the minting and management of certificates. The add-on is aptly called cert-manager.

Cert-manager is a controller that runs in your cluster. It installs a set of CRDs that enable declarative management of Certificate Authorities (CAs) and Certificates via the Kubernetes API. More importantly, it supports different certificate issuers, including ACME-based CAs, HashiCorp Vault, Venafi, etc. It also offers an extension point to implement custom issuers, when necessary.

The certificate minting features of cert-manager revolve around issuers and certificates. Cert-manager has two issuer Custom Resources. The Issuer resource represents a CA that signs certificates in a specific Kubernetes Namespace. If you want to issue certificates across all Namespaces, you can use the ClusterIssuer resource. The following is a sample ClusterIssuer definition that uses a private key stored in a Kubernetes Secret named `platform-ca-key-pair`:

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: prod-ca-issuer
spec:
  ca:
    secretName: platform-ca-key-pair
```

The great thing about cert-manager is that it integrates with the Ingress API to automatically mint certificates for Ingress resources. For example, given the following Ingress object, cert-manager automatically creates a certificate key pair suitable for TLS:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    cert-manager.io/cluster-issuer: prod-ca-issuer ❶
  name: bearcanoe-com
spec:
  tls:
  - hosts:
    - bearcanoe.com
    secretName: bearcanoe-cert-key-pair ❷
  rules:
  - host: bearcanoe.com
    http:
      paths:
      - path: /
        backend:
          serviceName: nginx
          servicePort: 80
```

❶ The `cert-manager.io/cluster-issuer` annotation tells cert-manager to use the `prod-ca-issuer` to mint the certificate.

❷ Cert-manager stores the certificate and private key in a Kubernetes Secret called `bearcanoe-cert-key-pair`.

Behind the scenes, cert-manager handles the certificate request process, which includes generating a private key, creating a certificate signing request (CSR), and submitting the CSR to the CA. Once the issuer mints the certificate, cert-manager stores it in the `bearcanoe-cert-key-pair` certificate. The Ingress controller can then pick it up and start serving the application over TLS. Figure 6-17 depicts the process in more detail.

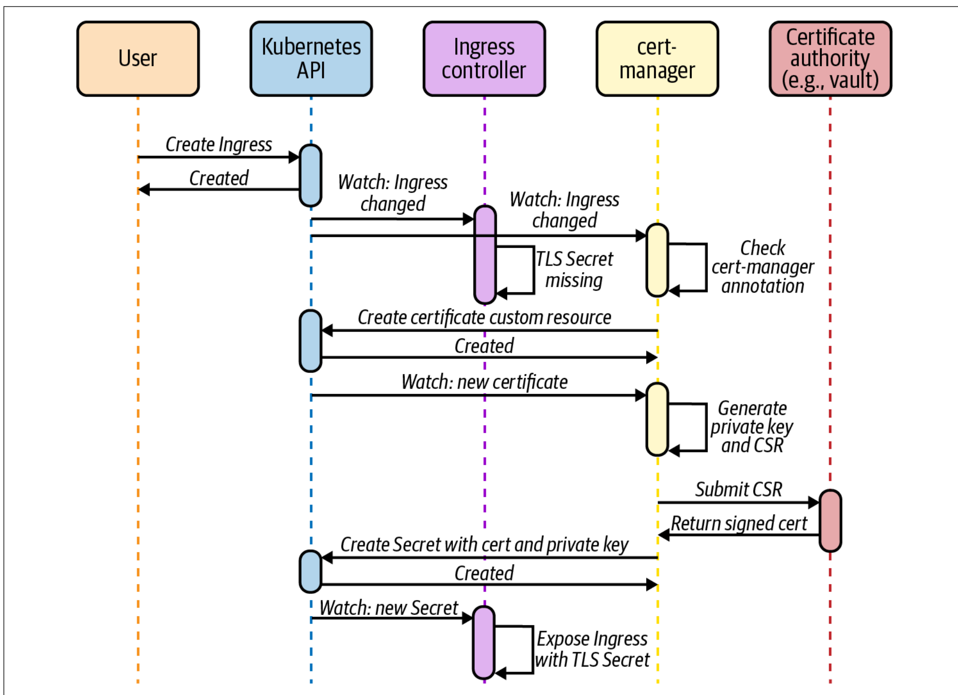


Figure 6-17. Cert-manager watches the Ingress API and requests a certificate from a Certificate Authority when the Ingress resource has the `cert-manager.io/cluster-issuer` annotation.

As you can see, cert-manager simplifies certificate management on Kubernetes. Most platforms we've encountered use cert-manager in some capacity. If you leverage cert-manager in your platform, consider using an external system such as Vault as the CA. Integrating cert-manager with an external system instead of using a CA backed by a Kubernetes Secret is a more robust and secure solution.

Service Mesh

As the industry continues to adopt containers and microservices, service meshes have gained immense popularity. While the term "service mesh" is relatively new, the concepts that it encompasses are not. Service meshes are a rehash of preexisting ideas in service routing, load balancing, and telemetry. Before the rise of containers and Kubernetes, hyperscale internet companies implemented service mesh precursors as they ran into challenges with microservices. Twitter, for example, created Finagle, a Scala library that all its microservices embedded. It handled load balancing, circuit breaking, automatic retries, telemetry, and more. Netflix developed Hystrix, a similar library for Java applications.

Containers and Kubernetes changed the landscape. Service meshes are no longer language-specific libraries like their precursors. Today, service meshes are distributed systems themselves. They consist of a control plane that configures a collection of proxies that implement the data plane. The routing, load balancing, telemetry, and other capabilities are built into the proxy instead of the application. The move to the proxy model has enabled even more apps to take advantage of these features, as there's no need to make code changes to participate in a mesh.

Service meshes provide a broad set of features that can be categorized across three pillars:

Routing and reliability
: Advanced traffic routing and reliability features such as traffic shifting, traffic mirroring, retries, and circuit breaking.

Security
: Identity and access control features that enable secure communication between services, including identity, certificate management, and mutual TLS (mTLS).

Observability
: Automated gathering of metrics and traces from all the interactions happening in the mesh.

Throughout the rest of this chapter, we are going to discuss service mesh in more detail. Before we do so, however, let's return to this book's central theme and ask "Do we need a service mesh?" Service meshes have risen in popularity as some organizations see them as a golden bullet to implement the aforementioned features. However, we have found that organizations should carefully consider the impact of adopting a service mesh.

When (Not) to Use a Service Mesh

A service mesh can provide immense value to an application platform and the applications that run atop. It offers an attractive feature set that your developers will

appreciate. At the same time, a service mesh brings a ton of complexity that you must deal with.

Kubernetes is a complex distributed system. Up to this point in the book, we have touched on some of the building blocks you need to create an application platform atop Kubernetes, and there are still a bunch of chapters left. The reality is that building a successful Kubernetes-based application platform is a lot of work. Keep this in mind when you are considering a service mesh. Tackling a service mesh implementation while you are beginning your Kubernetes journey will slow you down, if not take you down the path to failure.

We have seen these cases firsthand while working in the field. We have worked with platform teams who were blinded by the shiny features of a service mesh. Granted, those features would make their platform more attractive to developers and thus increase the platform's adoption. However, timing is important. Wait until you gain operational experience in production before thinking about service mesh.

Perhaps more critical is for you to understand your requirements or the problems you are trying to solve. Putting the cart before the horse will not only increase the chances of your platform failing but also result in wasted engineering effort. A fitting example of this mistake is an organization that dove into service mesh while developing a Kubernetes-based platform that was not yet in production. "We want a service mesh because we need everything it provides," they said. Twelve months later, the only feature they were using was the mesh's Ingress capabilities. No mutual TLS, no fancy routing, no tracing. Just Ingress. The engineering effort to get a dedicated Ingress controller ready for production is far less than a full-featured mesh implementation. There's something to be said for getting a minimum viable product into production and then iterating to add features moving forward.

After reading this, you might feel like we think there's no place for a service mesh in an application platform. Quite the opposite. A service mesh can solve a ton of problems if you have them, and it can bring a ton of value if you take advantage of it. In the end, we have found that a successful service mesh implementation boils down to timing it right and doing it for the right reasons.

The Service Mesh Interface (SMI)

Kubernetes provides interfaces for a variety of pluggable components. These interfaces include the Container Runtime Interface (CRI), the Container Networking Interface (CNI), and others. As we've seen throughout the book, these interfaces are what makes Kubernetes such an extensible foundation. Service mesh is slowly but surely becoming an important ingredient of a Kubernetes platform. Thus, the service mesh community collaborated to build the Service Mesh Interface, or SMI.

Similar to the other interfaces we've already discussed, the SMI specifies the interaction between Kubernetes and a service mesh. With that said, the SMI is different than other Kubernetes interfaces in that it is not part of the core Kubernetes project. Instead, the SMI project leverages CRDs to specify the interface. The SMI project also houses libraries to implement the interface, such as the SMI SDK for Go.

The SMI covers the three pillars we discussed in the previous section with a set of CRDs. The Traffic Split API is concerned with routing and splitting traffic across a number of services. It enables percent-based traffic splitting, which enables different deployment scenarios such as blue-green deployments and A/B testing. The following snippet is an example of a TrafficSplit that performs a canary deployment of the "flights" web service:

```
apiVersion: split.smi-spec.io/v1alpha3
kind: TrafficSplit
metadata:
  name: flights-canary
  namespace: bookings
spec:
  service: flights ❶
  backends: ❷
  - service: flights-v1
    weight: 70
  - service: flights-v2
    weight: 30
```

❶ The top-level Service that clients connect to (i.e., `flights.bookings.cluster.svc.local`).

❷ The backend Services that receive the traffic. The v1 version receives 70% of traffic and the v2 version receives the rest.

The Traffic Access Control and Traffic Specs APIs work together to implement security features such as access control. The Traffic Access Control API provides CRDs to control the service interactions that are allowed in the mesh. With these CRDs, developers can specify access control policy that determines which services can talk to each other and under what conditions (list of allowed HTTP methods, for example). The Traffic Specs API offers a way to describe traffic, including an `HTTPRouteGroup` CRD for HTTP traffic and a `TCPRoute` for TCP traffic. Together with the Traffic Access Control CRDs, these apply policy at the application level.

For example, the following HTTPRouteGroup and TrafficTarget allow all requests from the bookings service to the payments service. The HTTPRouteGroup resource describes the traffic, while the TrafficTarget specifies the source and destination services:

```
apiVersion: specs.smi-spec.io/v1alpha3
kind: HTTPRouteGroup
```

```
metadata:
  name: payment-processing
  namespace: payments
spec:
  matches:
  - name: everything ❶
    pathRegex: ".*"
    methods: ["*"]
---
apiVersion: access.smi-spec.io/v1alpha2
kind: TrafficTarget
metadata:
  name: allow-bookings
  namespace: payments
spec:
  destination: ❷
    kind: ServiceAccount
    name: payments
    namespace: payments
    port: 8080
  rules: ❸
  - kind: HTTPRouteGroup
    name: payment-processing
    matches:
    - everything
  sources: ❹
  - kind: ServiceAccount
    name: flights
    namespace: bookings
```

❶ Allow all requests in this HTTPRouteGroup.

❷ The destination service. In this case, the Pods using the `payments` Service Account in the `payments` Namespace.

❸ The HTTPRouteGroups that control the traffic between the source and destination services.

❹ The source service. In this case, the Pods using the `flights` Service Account in the `bookings` Namespace.

Finally, the Traffic Metrics API provides the telemetry functionality of a service mesh. This API is somewhat different than the rest in that it defines outputs instead of mechanisms to provide inputs. The Traffic Metrics API defines a standard to expose service metrics. Systems that need these metrics, such as monitoring systems, autoscalers, dashboards, and others, can consume them in a standardized fashion. The following snippet shows an example TrafficMetrics resource that exposes metrics for traffic between two Pods:

```yaml
apiVersion: metrics.smi-spec.io/v1alpha1
kind: TrafficMetrics
resource:
  name: flights-19sk18sj11-a9od2
  namespace: bookings
  kind: Pod
edge:
  direction: to
  side: client
  resource:
    name: payments-ks8xoa999x-xkop0
    namespace: payments
    kind: Pod
timestamp: 2020-08-09T01:07:23Z
window: 30s
metrics:
- name: p99_response_latency
  unit: seconds
  value: 13m
- name: p90_response_latency
  unit: seconds
  value: 7m
- name: p50_response_latency
  unit: seconds
  value: 3m
- name: success_count
  value: 100
- name: failure_count
  value: 0
```

The SMI is one of the newest interfaces in the Kubernetes community. While still under development and iteration, it paints the picture of where we are headed as a community. As with other interfaces in Kubernetes, the SMI enables platform builders to offer a service mesh using portable and provider-agnostic APIs, further increasing the value, flexibility, and power of Kubernetes.

The Data Plane Proxy

The data plane of a service mesh is a collection of proxies that connect services together. The Envoy proxy is one of the most popular service proxies in the cloud native ecosystem. Originally developed at Lyft, it has quickly become a prevalent building block in cloud native systems since it was open sourced in late 2016.

Envoy is used in Ingress controllers (Contour), API gateways (Ambassador, Gloo), and, you guessed it, service meshes (Istio, OSM).

One of the reasons why Envoy is such a good building block is its support for dynamic configuration over a gRPC/REST API. Open source proxies that predate Envoy were not designed for environments as dynamic as Kubernetes. They used static configuration files and required restarts for configuration changes to take effect.

Envoy, on the other hand, offers the xDS (* discovery service) APIs for dynamic configuration (depicted in Figure 6-18). It also supports hot restarts, which allow Envoy to reinitialize without dropping any active connections.

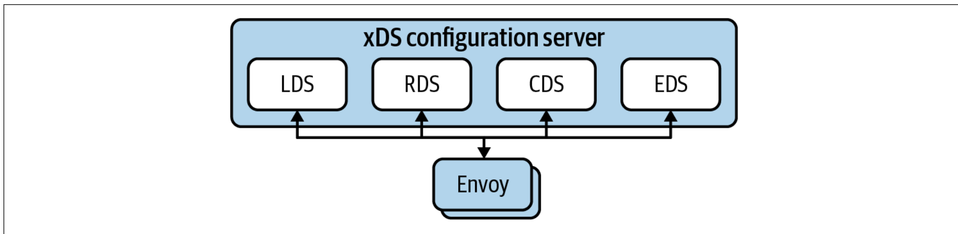


Figure 6-18. Envoy supports dynamic configuration via the XDS APIs. Envoy connects to a configuration server and requests its configuration using LDS, RDS, EDS, CDS, and other xDS APIs.

Envoy's xDS is a collection of APIs that includes the Listener Discovery Service (LDS), the Cluster Discovery Service (CDS), the Endpoints Discovery Service (EDS), the Route Discovery Service (RDS), and more. An Envoy *configuration server* implements these APIs and behaves as the source of dynamic configuration for Envoy. During startup, Envoy reaches out to a configuration server (typically over gRPC) and subscribes to configuration changes. As things change in the environment, the configuration server streams changes to Envoy. Let's review the xDS APIs in more detail.

The LDS API configures Envoy's *Listeners*. Listeners are the entry point into the proxy. Envoy can open multiple Listeners that clients can connect to. A typical example is listening on ports 80 and 443 for HTTP and HTTPS traffic.

Each Listener has a set of filter chains that determine how to handle incoming traffic. The HTTP connection manager filter leverages the RDS API to obtain routing configuration. The routing configuration tells Envoy how to route incoming HTTP requests. It provides details around virtual hosts and request matching (path-based, header-based, and others).

Each route in the routing configuration references a *Cluster*. A cluster is a collection of *Endpoints* that belong to the same service. Envoy discovers Clusters and Endpoints using the CDS and EDS APIs, respectively. Interestingly enough, the EDS API does not have an Endpoint object per se. Instead, Endpoints are assigned to clusters using *ClusterLoadAssignment* objects.

While digging into the details of the xDS APIs merits its own book, we hope the preceding overview gives you an idea of how Envoy works and its capabilities. To summarize, listeners bind to ports and accept connections from clients. Listeners have filter chains that determine what to do with incoming connections. For example, the HTTP filter inspects requests and maps them to clusters. Each cluster has one or

more endpoints that end up receiving and handling the traffic. Figure 6-19 shows a graphical representation of these concepts and how they relate to each other.

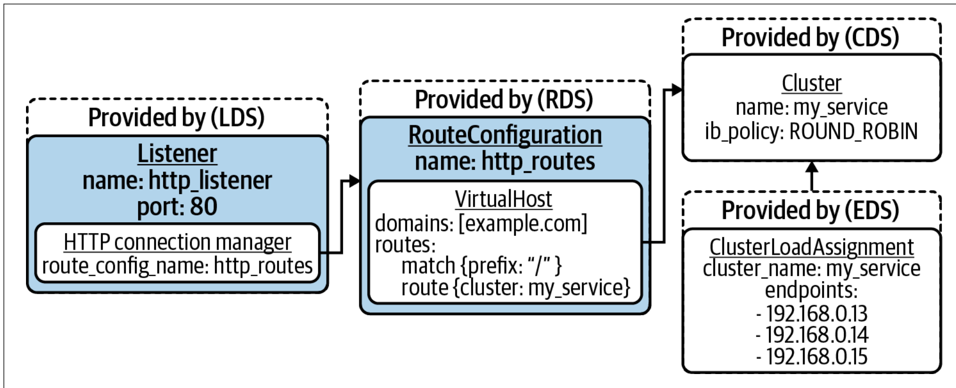


Figure 6-19. Envoy configuration with a Listener that binds to port 80. The Listener has an HTTP connection manager filter that references a routing configuration. The routing config matches requests with / prefix and forwards requests to the `my_service` cluster, which has three endpoints.

Service Mesh on Kubernetes

In the previous section, we discussed how the data plane of a service mesh provides connectivity between services. We also talked about Envoy as a data plane proxy and how it supports dynamic configuration through the xDS APIs. To build a service mesh on Kubernetes, we need a control plane that configures the mesh's data plane according to what's happening inside the cluster. The control plane needs to understand Services, Endpoints, Pods, etc. Furthermore, it needs to expose Kubernetes Custom Resources that developers can use to configure the service mesh.

One of the most popular service mesh implementations for Kubernetes is Istio. Istio implements a control plane for an Envoy-based service mesh. The control plane is implemented in a component called istiod, which itself has three primary subcomponents: Pilot, Citadel, and Galley. Pilot is an Envoy configuration server. It implements the xDS APIs and streams the configuration to the Envoy proxies running alongside the applications. Citadel is responsible for certificate management inside the mesh. It mints certificates that are used to establish service identity and mutual TLS. Finally, Galley interacts with external systems such as Kubernetes to obtain configuration. It abstracts the underlying platform and translates configuration for the other istiod components. Figure 6-20 shows the interactions between the Istio control plane components.

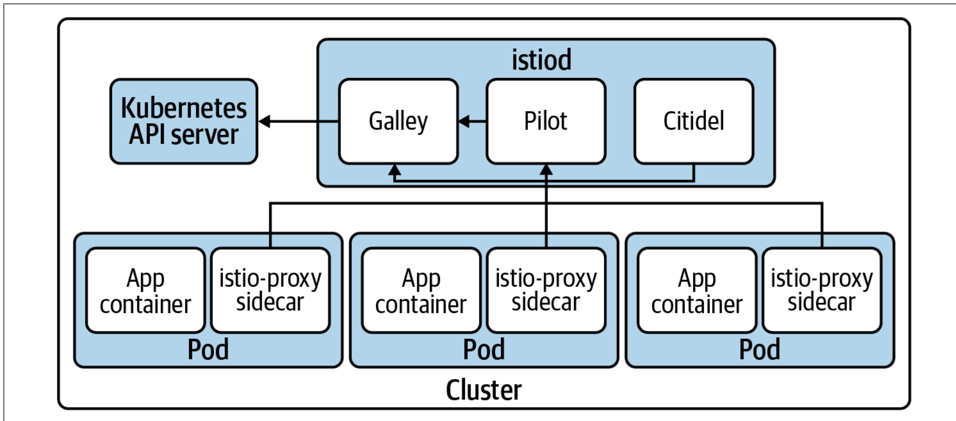


Figure 6-20. Istio control plane interactions.

Istio provides other capabilities besides configuring the data plane of the service mesh. First, Istio includes a mutating admission webhook that injects an Envoy sidecar into Pods. Every Pod that participates in the mesh has an Envoy sidecar that handles all the incoming and outgoing connections. The mutating webhook improves the developer experience on the platform, given that developers don't have to manually add the sidecar proxy to all of their application deployment manifests. The platform injects the sidecar automatically with both an opt-in and opt-out model. With that said, merely injecting the Envoy proxy sidecar alongside the workload does not mean the workload will automatically start sending traffic through Envoy. Thus, Istio uses an init-container to install iptables rules that intercept the Pod's network traffic and routes it to Envoy. The following snippet (trimmed for brevity) shows the Istio init-container configuration:

```
...
initContainers:
- args:
  - istio-iptables
  - --envoy-port ❶
  - "15001"
  - --inbound-capture-port ❷
  - "15006"
  - --proxy-uid
  - "1337"
  - --istio-inbound-interception-mode
  - REDIRECT
  - --istio-service-cidr ❸
  - '*'
  - --istio-inbound-ports ❹
  - '*'
  - --istio-local-exclude-ports
  - 15090,15021,15020
  image: docker.io/istio/proxyv2:1.6.7
```

```
    imagePullPolicy: Always
    name: istio-init
...
```

❶ Istio installs an iptables rule that captures all outbound traffic and sends it to Envoy at this port.

❷ Istio installs an iptables rule that captures all inbound traffic and sends it to Envoy at this port.

❸ List of CIDRs to redirect to Envoy. In this case, we are redirecting all CIDRs.

❹ List of ports to redirect to Envoy. In this case, we are redirecting all ports.

Now that we've discussed Istio's architecture, let's discuss some of the service mesh features that are typically used. One of the more common requirements we run into in the field is service authentication and encryption of service-to-service traffic. This feature is covered by the Traffic Access Control APIs in the SMI. Istio and most service mesh implementations use mutual TLS to achieve this. In Istio's case, mutual TLS is enabled by default for all services that are participating in the mesh. The workload sends unencrypted traffic to the sidecar proxy. The sidecar proxy upgrades the connection to mTLS and sends it along to the sidecar proxy on the other end. By default, the services can still receive non-TLS traffic from other services outside of the mesh. If you want to enforce mTLS for all interactions, Istio supports a `STRICT` mode that configures all services in the mesh to accept only TLS-encrypted requests. For example, you can enforce strict mTLS at the cluster level with the following configuration in the `istio-system` Namespace:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
```

Traffic management is another key concern handled by a service mesh. Traffic management is captured in the Traffic Split API of the SMI, even though Istio's traffic management features are more advanced. In addition to traffic splitting or shifting, Istio supports fault injection, circuit breaking, mirroring, and more. When it comes to traffic shifting, Istio uses two separate Custom Resources for configuration: VirtualService and DestinationRule.

- The *VirtualService* resource creates services in the mesh and specifies how traffic is routed to them. It specifies the hostname of the service and rules that control the destination of the requests. For example, the VirtualService can send 90% of

traffic to one destination and send the rest to another. Once the VirtualService evaluates the rules and chooses a destination, it sends the traffic along to a specific subset of a *DestinationRule*.

- The *DestinationRule* resource lists the "real" backends that are available for a given Service. Each backend is captured in a separate subset. Each subset can have its own routing configuration, such as load balancing policy, mutual TLS mode, and others.

As an example, let's consider a scenario where we want to slowly roll out version 2 of a service. We can use the following DestinationRule and VirtualService to achieve this. The DestinationRule creates two service subsets: v1 and v2. The VirtualService references these subsets. It sends 90% of traffic to the v1 subset and 10% of the traffic to the v2 subset:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: flights
spec:
  host: flights
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: flights
spec:
  hosts:
  - flights
  http:
  - route:
    - destination:
        host: flights
        subset: v1
      weight: 90
    - destination:
        host: flights
        subset: v2
      weight: 10
```

Service observability is another feature that is commonly sought after. Because there's a proxy between all services in the mesh, deriving service-level metrics is straightforward. Developers get these metrics without having to instrument their applications.

The metrics are exposed in the Prometheus format, which makes them available to a wide range of monitoring systems. The following is an example metric captured by the sidecar proxy (some labels removed for brevity). The metric shows that there have been 7183 successful requests from the flights booking service to the payment processing service:

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_service_name="payments",
  destination_service_namespace="payments",
  destination_version="v1",
  request_protocol="http",
  ...
  response_code="200",
  source_app="bookings",
  source_version="v1",
  source_workload="bookings-v1",
  source_workload_namespace="flights"
} 7183
```

Overall, Istio offers all of the features that are captured in the SMI. However, it does not yet implement the SMI APIs (Istio v1.6). The SMI community maintains an adapter that you can use to make the SMI APIs work with Istio. We discussed Istio mainly because it is the service mesh that we've most commonly encountered in the field. With that said, there are other meshes available in the Kubernetes ecosystem, including Linkerd, Consul Connect, Maesh, and more. One of the things that varies across these implementations is the data plane architecture, which we'll discuss next.

Data Plane Architecture

A service mesh is a highway that services can use to communicate with each other. To get onto this highway, services use a proxy that serves as the on-ramp. Service meshes follow one of two architecture models when it comes to the data plane: the sidecar proxy or the node proxy.

Sidecar proxy

The sidecar proxy is the most common architecture model among the two. As we discussed in the previous section, Istio follows this model to implement its data plane with Envoy proxies. Linkerd uses this approach as well. In essence, service meshes that follow this pattern deploy the proxy inside the workload's Pod, running alongside the service. Once deployed, the sidecar proxy intercepts all the communications into and out of the service, as depicted in Figure 6-21.

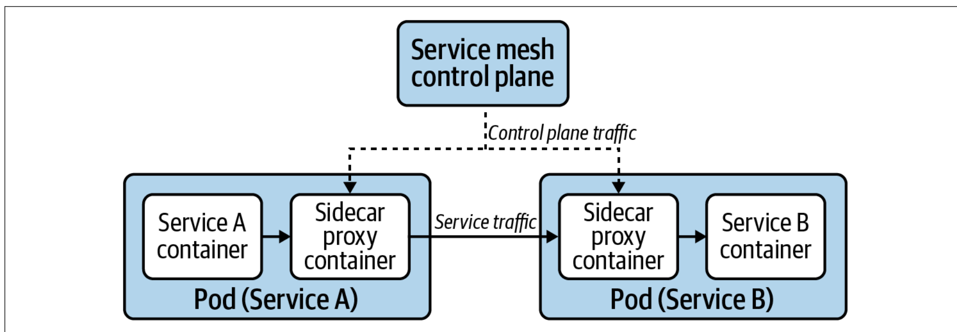


Figure 6-21. Pods participating in the mesh have a sidecar proxy that intercepts the Pod's network traffic.

When compared to the node proxy approach, the sidecar proxy architecture can have greater impact on services when it comes to data plane upgrades. The upgrade involves rolling all the service Pods, as there is no way to upgrade the sidecar without re-creating the Pods.

Node proxy

The node proxy is an alternative data plane architecture. Instead of injecting a sidecar proxy into each service, the service mesh consists of a single proxy running on each node. Each node proxy handles the traffic for all services running on their node, as depicted in Figure 6-22. Service meshes that follow this architecture include Consul Connect and Maesh. The first version of Linkerd used node proxies as well, but the project has since moved to the sidecar model in version 2.

When compared to the sidecar proxy architecture, the node proxy approach can have greater performance impact on services. Because the proxy is shared by all the services on a node, services can suffer from noisy neighbor problems and the proxy can become a network bottleneck.

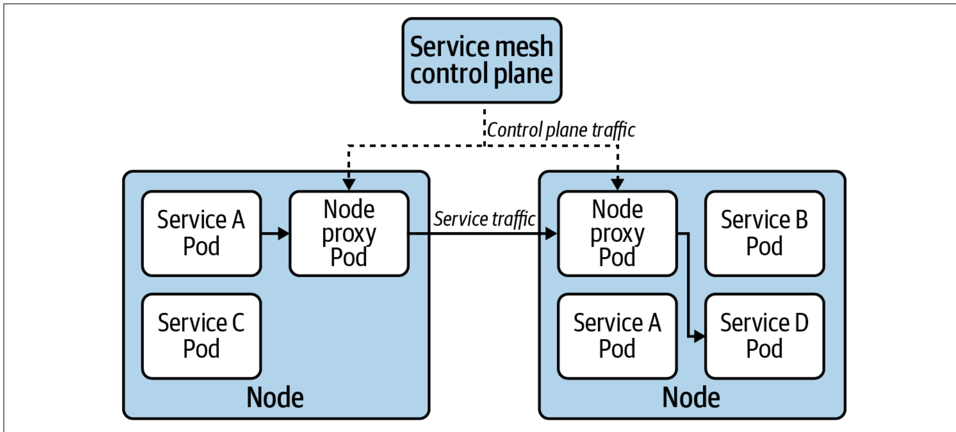


Figure 6-22. The node proxy model involves a single service mesh proxy that handles the traffic for all services on the node.

Adopting a Service Mesh

Adopting a service mesh can seem like a daunting task. Should you deploy it to an existing cluster? How do you avoid affecting workloads that are already running? How can you selectively onboard services for testing?

In this section, we will explore the different considerations you should make when introducing a service mesh to your application platform.

Prioritize one of the pillars

One of the first things to do is to prioritize one of the service mesh pillars. Doing so will allow you to narrow the scope, both from an implementation and testing perspective. Depending on your requirements (which you've established if you're adopting a service mesh, right?), you might prioritize mutual TLS, for example, as the first pillar. In this case, you can focus on deploying the PKI necessary to support this feature. No need to worry about setting up a tracing stack or spending development cycles testing traffic routing and management.

Focusing on one of the pillars enables you to learn about the mesh, understand how it behaves in your platform, and gain operational expertise. Once you feel comfortable, you can implement additional pillars, as necessary. In essence, you will be more successful if you follow a piecemeal deployment instead of a big-bang implementation.

Deploy to a new or an existing cluster?

Depending on your platform's life cycle and topology, you might have a choice between deploying the service mesh to a new, fresh cluster or adding it to an existing cluster. When possible, prefer going down the new cluster route. This eliminates any

potential disruption to applications that would otherwise be running in an existing cluster. If your clusters are ephemeral, deploying the service mesh to a new cluster should be a natural path to follow.

In situations where you must introduce the service mesh into an existing cluster, make sure to perform extensive testing in your development and testing tiers. More importantly, offer an onboarding window that allows development teams to experiment and test their services with the mesh before rolling it out to the staging and production tiers. Finally, provide a mechanism that allows applications to opt into being part of the mesh. A common way to enable the opt-in mechanism is to provide a Pod annotation. Istio, for example, provides an annotation (`sidecar.istio.io/inject`) that determines whether the platform should inject the sidecar proxy into the workload, which is visible in the following snippet:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
      - name: nginx
        image: nginx
```

Handling upgrades

When offering a service mesh as part of your platform, you must have a solid upgrade strategy in place. Keep in mind that the service mesh data plane is in the critical path that connects your services, including your cluster's edge (regardless of whether you are using the mesh's Ingress gateway or another Ingress controller). What happens when there's a CVE that affects the mesh's proxy? How will you handle upgrades effectively? Do not adopt a service mesh without understanding these concerns and having a well-established upgrade strategy.

The upgrade strategy must account for both the control plane and the data plane. The control plane upgrade carries less risk, as the mesh's data plane should continue to function without it. With that said, do not discount control plane upgrades. You should understand the version compatibility between the control plane and the data plane. If possible, follow a canary upgrade pattern, as recommended by the Istio project. Also make sure to review any service mesh Custom Resource Definition (CRD) changes and whether they impact your services.

The data plane upgrade is more involved, given the number of proxies running on the platform and the fact that the proxy is handling service traffic. When the proxy runs as a sidecar, the entire Pod must be re-created to upgrade the proxy as Kubernetes doesn't support in-place upgrades of containers. Whether you do a full data plane upgrade or a slow rollout of the new data plane proxy depends on the reason behind the upgrade. One one hand, if you are upgrading the data plane to handle a vulnerability in the proxy, you must re-create every single Pod that participates in the mesh to address the vulnerability. As you can imagine, this can be disruptive to some applications. If, on the other hand, you are upgrading to take advantage of new features or bug fixes, you can let the new version of the proxy roll out as Pods are created or moved around in the cluster. This slower, less disruptive upgrade results in version sprawl of the proxy, which may be acceptable as long as the service mesh supports it. Regardless of why you are upgrading, always use your development and testing tiers to practice and validate service mesh upgrades.

Another thing to keep in mind is that meshes typically have a narrow set of Kubernetes versions they can support. How does a Kubernetes upgrade affect your service mesh? Does leveraging a service mesh hinder your ability to upgrade Kubernetes as soon as a new version is released? Given that Kubernetes APIs are relatively stable, this should not be the case. However, it is possible and something to keep in mind.

Resource overhead

One of the primary trade-offs of using a service mesh is the resource overhead that it carries, especially in the sidecar architecture. As we've discussed, the service mesh injects a proxy into each Pod in the cluster. To get its job done, the proxy consumes resources (CPU and memory) that would otherwise be available to other services. When adopting a service mesh, you must understand this overhead and whether the trade-off is worth it. If you are running the cluster in a datacenter, the overhead is probably palatable. However, the overhead might prevent you from using a service mesh in edge deployments where resource constraints are tighter.

Perhaps more important, a service mesh introduces latency between services given that the service calls are traversing a proxy on both the source and the destination services. While the proxies used in service meshes are usually highly performant, it is important to understand the latency overhead they introduce and whether your application can function given the overhead.

When evaluating a service mesh, spend time investigating its resource overhead. Even better, run performance tests with your services to understand how the mesh behaves under load.

Certificate Authority for mutual TLS

The identity features of a service mesh are usually based on X.509 certificates. Proxies in the mesh use these certificates to establish mutual TLS (mTLS) connections between services.

Before being able to leverage the mTLS features of a service mesh, you must establish a certificate management strategy. While the mesh is usually responsible for minting the service certificates, it is up to you to determine the Certificate Authority (CA). In most cases, a service mesh uses a self-signed certificate as the CA. However, mature service meshes allow you to bring your own CA, if necessary.

Because the service mesh handles service-to-service communications, using a self-signed CA is adequate. The CA is essentially an implementation detail that is invisible to your applications and their clients. With that said, security teams can disapprove of the use of self-signed CAs. When adopting a service mesh, make sure to bring your security team into the conversation.

If using a self-signed CA for mTLS is not viable, you will have to provide a CA certificate and key that the service mesh can use to mint certificates. Alternatively, you can integrate with an external CA, such as Vault, when an integration is available.

Multicluster service mesh

Some service meshes offer multicluster capabilities that you can use to extend the mesh across multiple Kubernetes clusters. The goal of these capabilities is to connect services running in different clusters through a secure channel that is transparent to the application. Multicluster meshes increase the complexity of your platform. They can have both performance and fault-domain implications that developers might have to be aware of. In any case, while creating multicluster meshes might seem attractive, you should avoid them until you gain the operational knowledge to run a service mesh successfully within a single cluster.

Summary

Service routing is a crucial concern when building an application platform atop Kubernetes. Services provide layer 3/4 routing and load balancing capabilities to applications. They enable applications to communicate with other services in the cluster without worrying about changing Pod IPs or failing cluster nodes. Furthermore, developers can use NodePort and LoadBalancer Services to expose their applications to clients outside of the cluster.

Ingress builds on top of Services to provide richer routing capabilities. Developers can use the Ingress API to route traffic according to application-level concerns, such as the `Host` header of the request or the path that the client is trying to reach. The Ingress API is satisfied by an Ingress controller, which you must deploy before using

Ingress resources. Once installed, the Ingress controller handles incoming requests and routes them according to the Ingress configuration defined in the API.

If you have a large portfolio of microservices-based applications, your developers might benefit from a service mesh's capabilities. When using a service mesh, services communicate with each other through proxies that augment the interaction. Service meshes can provide a variety of features, including traffic management, mutual TLS, access control, automated service metrics gathering, and more. Like other interfaces in the Kubernetes ecosystem, the Service Mesh Interface (SMI) aims to enable platform operators to use a service mesh without tying themselves to specific implementations. However, before adopting a service mesh, ensure that you have the operational expertise in your team to operate an additional distributed system on top of Kubernetes.

CHAPTER 7

Secret Management

In any application stack, we are almost guaranteed to run into secret data. This is the data that applications want to keep, well, secret. Commonly, we associate secrets with credentials. Often these credentials are used to access systems within or external to the cluster, such as databases or message queues. We also run into secret data when using private keys, which may support our application's ability to perform mutual TLS with other applications. These kinds of concerns are covered in Chapter 11. The existence of secrets bring in many operational concerns to consider, such as:

Secret rotation policies
: How long is a secret allowed to remain before it must be changed?

Key (encryption) rotation policies
: Assuming secret data is encrypted at the application layer before being persisted to disk, how long is an encryption key allowed to stay around before it must be rotated?

Secret storage policies
: What requirements must be satisfied in order to store secret data? Do you need to persist secrets to isolated hardware? Do you need your secret management solution to integrate with a hardware security module (HSM)?

Remediation plan
: If secret(s) or encryption key(s) are compromised, how do you plan to remediate? Can your plan or automation be run without impact to applications?

A good starting point is to determine what layer to offer secret management for your applications. Some organizations choose to not solve this at a platform level and instead expect application teams to inject secrets into their applications dynamically. For example, if an organization is running a secret management system such as Vault, applications can talk directly to the API to authenticate and retrieve secrets.

Application frameworks may even offer libraries to talk directly to these systems. For example, Spring offers the spring-vault project to authenticate against Vault, retrieve secrets, and inject their values directly into Java classes. While possible to do at the application layer, many platform teams aspire to offer enterprise-grade secret capabilities as platform services, perhaps in a way that application developers need not be concerned with how the secret got there or what external provider (e.g., Vault) is being used under the hood.

In this chapter we'll dive into how to think about secret data in Kubernetes. We'll start at lower-level layers Kubernetes runs on and work up to the APIs Kubernetes exposes that make secret data available to workloads. Like many topics in this book, you'll find these considerations and recommendations to live on a spectrum—one end of the spectrum includes how secure you're willing to get relative to engineering effort and tolerance for risk, and the other end focuses on what level of abstractions you'd like to provide to developers consuming this platform.

Defense in Depth

Protection of our secret data largely comes down to what depths we're willing to go to make it secure. As much as we'd like to say we always choose the most secure options, the reality is we make sensible decisions that keep us "safe enough" and ideally harden them over time. This comes with risk and technical debt that is quintessential to our work. However, there's no denying that a misjudgment of what is "safe enough" can quickly make us famous, and not in a good way. In the following sections, we'll look at these layers of security and call out some of the most critical points.

Defense can start literally at the physical layer. A prime example is Google. It has multiple whitepapers, and even a video on YouTube, that describe its approach to datacenter security. This includes metal detectors, vehicle barriers capable of stopping a semi truck, and several layers of building security just to get into the datacenter. This attention to detail extends beyond what is live and racked. When drives are retired, Google has authorized staff zero-out the data and then potentially crush and shred the drives. While the subject of physical security is interesting, this book will not go into depth around physical security of your datacenter, but the steps cloud providers take to ensure the security of their hardware are amazing, and that's just the start.

Let's say a human did somehow get access to a disk before it was zeroed out or crushed. Most cloud providers and datacenters are securing their physical disks by ensuring the drive is encrypted at rest. Providers may do this with their own encryption keys and/or they allow customers to provide their own keys, which makes it near impossible for providers to access your unencrypted data. This is a perfect example of defense in depth. We are protected from a physical standpoint, intra-datacenter, and we extend that to encrypting data on the physical disks themselves, closing off further opportunities for bad actors internally to do anything with users' data.

Disk Encryption

Let's take a closer look at the disk encryption domain. There are several ways to encrypt disks. A common method in Linux for full block encryption is leveraging Linux Unified Key System (LUKS). LUKS works in conjunction with the dm-crypt encryption subsystem, which has been available in the Linux kernel since version 2.6. For dedicated storage systems such as vSAN, ceph, or Gluster, each provide one or many means to provide encryption at rest. In cloud providers, the default encryption behavior can vary. For AWS, you should explore its documentation to enable encryption for Elastic Block Storage. AWS offers the ability to enable encryption by default, which we recommend as a best-practice setting. Google Cloud, on the other hand, performs encryption at rest as its default mode. Similar to AWS, it can be configured with the Key Management Service (KMS), which enables you to customize the encryption behavior, such as providing your own encryption keys.

Regardless of the trust you do or don't have for your cloud provider or datacenter operators, we highly recommend encryption at rest as your default practice. Encryption at rest essentially means the data is *stored* encrypted. Not only is this good for mitigating attack vectors, but it provides some protection against possible mistakes. For example, the world of virtual machines has made it trivial to create snapshots of hosts. Snapshots end up like any other file, data that is too easy to accidentally expose to an internal or external network. In the spirit of defense in depth, we should protect ourselves against this scenario where, if we select the wrong button via a UI or field in an API, the leaked data is useless for those without private key access. Figure 7-1 shows the UI for how easily these permissions can be toggled.

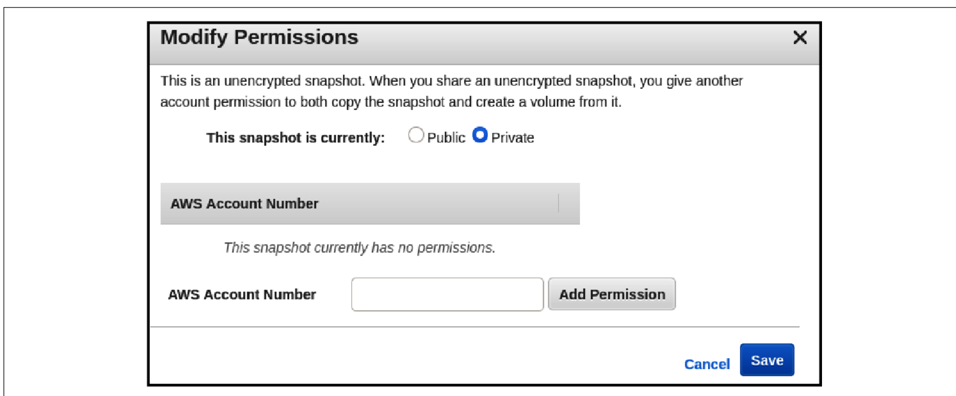


Figure 7-1. Permission setting on an AWS snapshot, as the warning says "making public" will give others access to create a volume from this snapshot and, potentially, access the data.

Transport Security

With a better understanding of encrypted data at rest, how about data that is actively in flight? Without having explored the Kubernetes secret architecture yet, let's look at the paths a secret can take when being transported between services. Figure 7-2 shows some of the interaction points for secrets. The arrows represent the secret moving through the network between hosts.

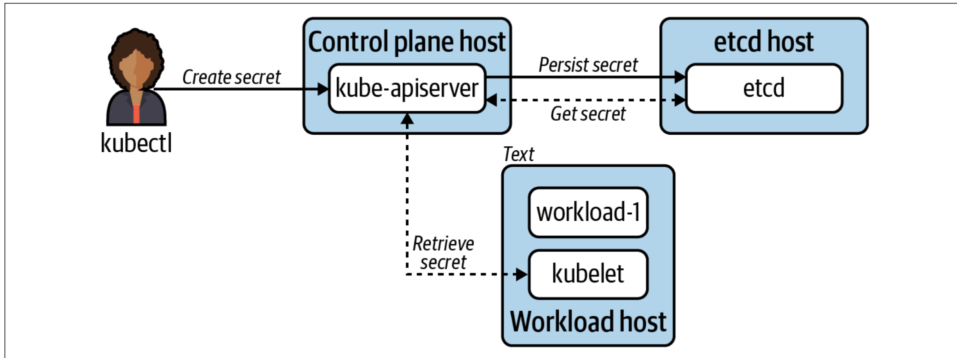


Figure 7-2. Diagram demonstrating points where a secret may pass over the wire.

The figure shows secret data moving across the network to reach different hosts. No matter how strong our encryption at rest strategy is, if any of the interaction points do not communicate over TLS, we have exposed our secret data. As seen Figure 7-2, this includes the human-to-system interaction, kubectl, and the system-to-system interaction, kubelet to API server. In summary, it's crucial communications with the API server and etcd that happen exclusively over TLS. We won't spend much time discussing this need as it is the default for almost every mode of installing or bootstrapping Kubernetes clusters. Often the only configuration you may wish to do is to provide a Certificate Authority (CA) to generate the certificates. However, keep in mind these certificates are internal to Kubernetes system components. With this in mind, you might not need to overwrite the default CA Kubernetes will generate.

Application Encryption

Application encryption is the encryption we perform within our system components or workloads running in Kubernetes. Application encryption can have many layers itself. For example, a workload in Kubernetes can encrypt data before persisting it to Kubernetes, which could then encrypt it, and then persisting it to etcd where it'll be encrypted at the filesystem level. The first two encryption points are considered "application-level." That's a lot of encryption!

While we won't always have encryption or decryption take place at that many levels, there is something to be said about data being encrypted at least once at the application level. Consider what we've talked about thus far: encryption over TLS and encryption at rest. If we'd stopped there, we'd have a decent start. When secret data is in flight, it will be encrypted, and it'll also be encrypted on the physical disk. But what about on the running system? While the bits persisted to disk may be encrypted, if a user were to gain access to the system they would likely be able to read the data! Consider your encrypted desktop computer where you may keep sensitive credentials in a dotfile (we've all done it). If I steal your computer and try to access this data by sledding the drive, I won't be able to get the information I'm after. However, if I succeed at booting your computer *and logging in as you*, I now have full access to this data.

Application encryption is the act of encrypting that data with a key at the userspace level. In this computer example, I could use a (strongly) password protected gpg key to encrypt that dotfile, requiring my user to decrypt it before it can be used. Writing a simple script can automate this process and you're off to the races with a far *deeper* security model. As the attacker logged in as you, even the decryption key is useless because without the password it's just useless bits. The same consideration applies to Kubernetes. Going forward we're going to assume two things are set up in your cluster:

- Encryption at rest is enabled in the filesystem and/or storage systems used by Kubernetes.
- TLS is enabled for all Kubernetes components and etcd.

With this, let's begin our exploration of encryption at the Kubernetes application level.

The Kubernetes Secret API

The Kubernetes Secret API is one of the most-used APIs in Kubernetes. While there are many ways for us to populate the Secret objects, the API provides a consistent means for workloads to interact with secret data. Secret objects heavily resemble ConfigMaps. They also have similar mechanics around how workloads can consume the objects, via environment variables or volume data. Consider the following Secret object:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  dbuser: aGVwdGlvCg==
  dbkey: YmVhcmNhbm9lCg==
```

In the `data` field, `dbuser` and `dbkey` are base64 encoded. All Kubernetes secret data is. If you wish to submit nonencoded string data to the API server, you can use the `stringData` field as follows:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  dbuser: heptio
  dbkey: bearcanoe
```

When applied, the `stringData` will be encoded at the API server and applied to etcd. A common misconception is that Kubernetes is encoding this data as a practice of security. This is not the case. Secret data can contain all kinds of strange characters or binary data. In order to ensure it's stored correctly, it is base64 encoded. By default, the key mechanism for ensuring that Secrets aren't compromised is RBAC. Understanding the implications of RBAC verbs as they pertain to Secrets is crucial to prevent introducing vectors of attack:

`get`
: Retrieve the data of a known secret by its name.

`list`
: Get a list of all secrets and/or *secret data*.

`watch`
: Watch any secret change and/or change to *secret data*.

As you can imagine, small RBAC mistakes, such as giving the user list access, expose every Secret in a Namespace or, worse, the entire cluster if a ClusterRoleBinding is accidentally used. The truth is, in many cases, users don't need any of these permissions. This is because a user's RBAC does not determine what secrets the workload can have access to. Generally the kubelet is responsible for making the secret available to the container(s) in a Pod. In summary, as long as your Pod references a valid secret, the kubelet will make it available through the means you specify. There are a few options to how we expose the secret in the workload, which is covered next.

The Scope of Secrets

The kubelet handling secret retrieval and injection is very convenient. However, it begs the question, How does the kubelet know whether my application should be able to access a secret? The model for workload access to secrets in Kubernetes is very simple, for better or for worse. Secrets are Namespace scoped, meaning that without replicating the Secret across Namespaces, a Pod may reference Secret(s) only in its Namespace. Which also means that Pods may access *any* Secret available in their

Namespace. This is one reason careful consideration of how Namespaces are shared among workloads is critical. If this model is unacceptable, there are ways to add additional checks at the admission control layer, which will be covered in a future chapter.

Secret Consumption Models

For workloads wishing to consume a Secret, there are several choices. The preference on how secret data is ingested may depend on the application. However, there are trade-offs to the approach you choose. In the coming sections, we'll look at three means of consuming secret data in workloads.

Environment variables

Secret data may be injected into environment variables. In the workload YAML, an arbitrary key and reference to the secret may be specified. This can be a nice feature for workloads moving onto Kubernetes that already expect an environment variable by reducing the need to change application codes. Consider the following Pod example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    env:
      - name: USER ❶
        valueFrom:
          secretKeyRef:
            name: mysecret ❷
            key: dbuser ❸
      - name: PASS
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: dbkey
```

❶ The environment variable key that will be available in the application.

❷ The name of the Secret object in Kubernetes.

❸ The key in the Secret object that should be injected into the `USER` variable.

The downside to exposing secrets in environment variables is their inability to be hot reloaded. A change in a Secret object will not be reflected until the Pod is re-created.

This could occur through manual intervention or system events such as the need to reshedule. Additionally, it is worth calling out that some consider secrets in environment variables to be less secure than reading from volume mounts. This point can be debated, but it is fair to call out some of the common opportunities to leak. Namely, when processes or container runtimes are inspected, there may be ways to see the environment variables in plain text. Additionally, some frameworks, libraries, or languages may support debug or crash modes where they dump (spew) environment variables out to the logs. Before using environment variables, these risks should be considered.

Volumes

Alternatively, secret objects may be injected via volumes. In the workload's YAML, a volume is configured where the secret is referenced. The container that the secret should be injected into references that volume using a `volumeMount`:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: creds ❷
      readOnly: true
      mountPath: "/etc/credentials" ❸
  volumes: ❶
  - name: creds
    secret:
      secretName: mysecret
```

❶ Pod-level volumes available for mounting. Name specified must be referenced in the mount.

❷ The volume object to mount into the container filesystem.

❸ Where in the container filesystem the mount is made available.

With this Pod manifest, the secret data is available under */etc/credentials* with each key/value pair in the secret object getting its own file:

```
root@nginx:/# cat /etc/credentials/db
dbkey   dbuser
```

The biggest benefit to the volume approach is that secrets may be updated dynamically, without the Pod restarting. When a change to the secret is seen, the kubelet will reload the secret and it'll show as updated in the container's filesystem. It's important

to call out that the kubelet is using tmpfs, on Linux, to ensure secret data is stored exclusively in memory. We can see this by examining the mount table file on the Linux host:

```
# grep 'secret/creds' secret/creds

tmpfs
/var/lib/kubelet/pods/
e98df9fe-a970-416b-9ddf-bcaff15dff87/volumes/
kubernetes.io~secret/creds tmpfs rw,relatime 0 0
```

If the `nginx` Pod is removed from this host, this mount is discarded. With this model in mind, it's especially important to consider that Secret data should *never* be large in size. Ideally it holds credentials or keys and is never used as a pseudo database.

From an application perspective, a simple watch on the directory or file, then re-injecting values into the application, is all it would take to handle a secret change. No need to understand or communicate with the Kubernetes API server. This is an ideal pattern we've seen success with in many workloads.

Client API Consumption

The last consumption model, Client API Consumption, is not a core Kubernetes feature. This model puts the onus on the application to communicate with the kube-apiserver to retrieve Secret(s) and inject them into the application. There are several frameworks and libraries out there that make communicating with Kubernetes trivial for your application. For Java, Spring's Spring Cloud Kubernetes brings this functionality to Spring applications. It takes the commonly used Spring PropertySource type and enables it to be wired up by connecting to Kubernetes and retreiving Secrets and/or ConfigMaps.

Caution: Carefully Consider This Approach

While consumption of Secret objects directly from the client application is possible, we generally discourage this approach. Previously, we talked about Spring's affinity toward mounting secrets but talking to the API server for ConfigMap. Even in the case of ConfigMaps, it's less than ideal to require each application to communicate directly to the API server. From a philosophical standpoint, when possible, we'd rather the application not be aware of where it's running. Meaning, if we can make it run on a VM as we could on Kubernetes or another container service, that's preferable. Most languages and frameworks have primitives needed to read environment variables and files. The kubelet can ensure our containers get Secret data through these mediums, so why add provider-specific logic into our applications for this use case? Philosophy aside, this is another client that will need to connect and establish a watch on the API server. Not only is this another unnecessary connection, but we now need to ensure the workloads get their own Service Account with accompanying

RBAC to access their Secret object(s). Whereas, in delegating this work to the kubelet, no Service Account is required. In fact, the Service Account (`default`) can and should be disabled altogether!

Now that we have covered consumption of secrets at a workload level, it is time to talk about storing secret data.

Secret Data in etcd

Like most Kubernetes objects, Secrets are stored in etcd. By default, *no* encryption is done at the Kubernetes layer before persisting Secrets to etcd. Figure 7-3 shows the flow of a secret from manifest to etcd.

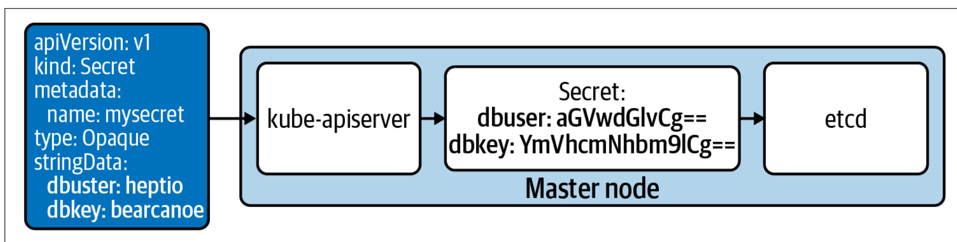


Figure 7-3. Default secret data flow in Kubernetes (the colocated etcd is sometimes run on a separate host).

While Kubernetes did *not* encrypt the secret data, this does not imply access to the data upon gaining hardware access. Remember that encryption at rest can be performed on disks through methods such as Linux Unified Key Setup (LUKS), where physical access to hardware gives you access only to encrypted data. For many cloud providers and enterprise datacenters, this is a default mode of operation. However, should we gain `ssh` access to the server running etcd and a user that has privileges or can escalate to see its filesystem, then we can potentially gain access to the secret data.

For some cases, this default model can be acceptable. etcd can be run external to the Kubernetes API server, ensuring it is separated by at least a hypervisor. In this model, an attacker would likely need to obtain root access to the etcd node, find the data location, and then read the secrets from the etcd database. The other entry point would be an adversary getting root access to the API server, locating the API server and etcd certs, then impersonating the API server in communicating with etcd to read secrets. Both cases assume potentially other breaches. For example, the attacker would have had to gain access to the internal network or subnet that is running the control-plane components. Additionally, they'd need to get the appropriate key to `ssh` into the node. Frankly, it's far more likely that an RBAC mistake or application compromise would expose a secret before this case.

To better understand the threat model, let's work through an example of how an attacker could gain access to Secrets. Let's consider the case where an attacker SSHs and gains root access to the kube-apiserver node. The attacker could set up a script as follows:

```
#!/bin/bash

# Change this based on location of etcd nodes
ENDPOINTS='192.168.3.43:2379'

ETCDCTL_API=3 etcdctl \
  --endpoints=${ENDPOINTS} \
  --cacert="/etc/kubernetes/pki/etcd/ca.crt" \
  --cert="/etc/kubernetes/pki/apiserver-etcd-client.crt" \
  --key="/etc/kubernetes/pki/apiserver-etcd-client.key" \
  ${@}
```

The certificate and key locations seen in this snippet are the default when Kubernetes was bootstrapped by kubeadm, which is also used by many tools such as cluster-api. etcd stores the secret data within the directory */registry/secrets/${NAMESPACE}/${SECRET_NAME}*. Using this script to get a secret, named `login1`, would look as follows:

```
# ./etcctl-script get /registry/secrets/default/login1

/registry/secrets/default/login1
k8s


v1Secret

login1default"*$6c991b48-036c-48f8-8be3-58175913915c2bB
0kubectl.kubernetes.io/last-applied-configuration{"apiVersion":"v1","data":
{"dbkey":"YmVhcmNhbm9lCg==","dbuser":"aGVwdGlvCg=="},"kind":"Secret",
"metadata":{"annotations":{},"name":"login1","namespace":"default"},
"type":"Opaque"}
z
dbkey
bearcanoe

dbuserheptio
Opaque"
```

With this, we have successfully compromised the secret `login1`.

Even though storing Secrets without encryption can be acceptable, many platform operators choose not to stop here. Kubernetes supports a few ways to encrypt the data within etcd, furthering the depth of your defense with respect to secrets. These include models to support encryption at rest (where encryption occurs at the Kubernetes layer) before it rests in etcd. These models include static-key encryption and envelope encryption.

Static-Key Encryption

The Kubernetes API server supports encrypting secrets at rest. This is achieved by providing the Kubernetes API server with an encryption key, which it will use to encrypt all secret objects before persisting them to etcd. Figure 7-4 shows the flow of a secret when static-key encryption is in play.

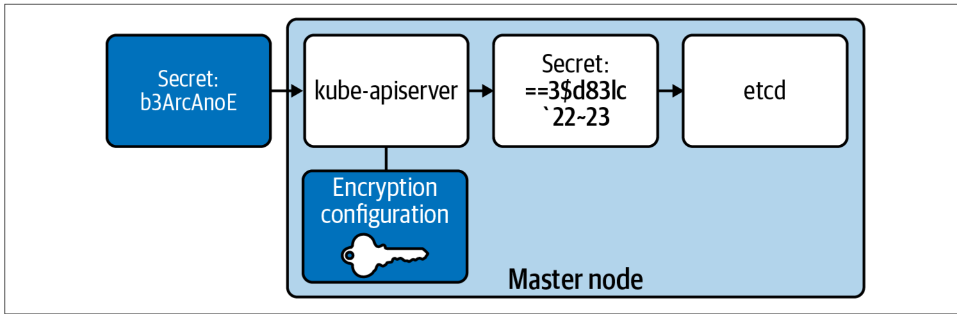


Figure 7-4. The relationship between an encryption key, on the API server, being used to encrypt secrets before storing them in etcd.

The key, held within an `EncryptionConfiguration`, is used to encrypt and decrypt Secret objects as they move through the API server. Should an attacker get access to etcd, they would see the encrypted data within, meaning the Secret data is not compromised. Keys can be created using a variety of providers, including secretbox, aescbc, and aesgcm.

Each provider has its own trade-offs, and we recommend working with your security team to select the appropriate option. Kubernetes issue #81127 is a good read on some considerations around these providers. If your enterprise needs to comply with standards such as the Federal Information Processing Standards (FIPS), these choices should be carefully considered. In our example we'll use secretbox, which acts as a fairly performant and secure encryption provider.

To set up static-key encryption, we must generate a 32-byte key. Our encryption and decryption model is symmetric, so a single key serves both purposes. How you generate a key can vary enterprise to enterprise. Using a Linux host, we can easily use `/dev/urandom` if we're satisfied with its entropy:

```
head -c 32 /dev/urandom | base64
```

Using this key data, an EncryptionConfiguration should be added to all nodes running a kube-apiserver. This static file should be added using configuration management such as ansible or KubeadmConfigSpec if using Cluster API. This ensures keys can be added, deleted, and rotated. The following example assumes the configuration is stored at */etc/kubernetes/pki/secrets/encryption-config.yaml*:

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - secretbox:
        keys:
        - name: secret-key-1
          secret: u7mcOcHKbFh9eVluB18hbFIsVfwpvgbXv650QacDYXA==
    # identity is a required (default) provider
    - identity: {}
```

The list of providers is ordered, meaning encryption will always occur using the first key and decryption will be attempted in order of keys listed. Identity is the default plain-text provider and should be last. If it's first, secrets will not be encrypted.

To respect the preceding configuration, every instance of the kube-apiserver must be updated to load the EncryptionConfiguration locally. In */etc/kubernetes/manifests/kube-apiserver.yaml*, an argument can be added as follows.

```
--encryption-provider-config=/etc/kubernetes/pki/secrets/encryption-config.yaml
```

Once the kube-apiserver(s) restart, this change will take effect and secrets will be encrypted before being sent to etcd. The kube-apiserver restart may be automatic. For example, when using static Pods to run the API server, a change to the manifest file will trigger a restart. Once you're past stages of experimentation, it's recommended you pre-provision hosts with this file and ensure the `encryption-provider` is enabled by default. The EncryptionConfiguration file can be added using configuration management tools such as Ansible or, with cluster-api, by setting that static file in a kubeadmConfigSpec. Note this cluster-api approach will put the EncryptionConfiguration in user data; make sure the user data is encrypted! Adding the `encryption-provider-config` flag to the API server can be done by adding the argument to the `apiServer` within a ClusterConfiguration, assuming you're using kubeadm. Otherwise, ensure the flag is present based on your mechanism for starting the server.

To validate the encryption, you can apply a new secret object to the API server. Assuming the secret is named `login2`, using the script from the previous section we can retrieve it as follows:

```
# ./etcctl-script get /registry/secrets/default/login2

/registry/secrets/default/login2
k8s:enc:secretbox:v1:secret-key-1:^DH
                                                  HN,lU/:L kdR<_h (fO$V
y.
  r/m
MjVĄGP<%B0kZHY}->q|&c?a\i#xoZsVXd+8_rCɣgcj[Mv<X5N):MQ'7t
```

```
'pLBxqپ)b7+r49ع`f
                  6(iciQϑΓ$'.ejbprλ=Cp+R-D%q!r/pbv1_.izyPlQ)1!7@X\0
                                                                  EiLr(dwlS
```

Here we can see the data is fully encrypted in etcd. Note there is metadata specifying which provider (`secretbox`) and key (`secret-key-1`) was used to do the encryption. This is important to Kubernetes as it supports many providers and keys at once. Any object created before the encryption key was set; let's assume `login1` can be queried and will still show up in plain text:

```
# ./etcctl-script get /registry/secrets/default/login1

/registry/secrets/default/login1
k8s
```

This demonstrates two important concepts. One, `login1` is *not* encrypted. While the encryption key is in place, only newly created or altered secret objects will be encrypted using this key. Secondly, when passing back through the kube-apiserver, no provider/key mapping is present and no decryption will be attempted. This latter concept is important because it is highly recommended you rotate encryption keys over a defined span. Let's say you rotate once every three months. When three months have elapsed, we'd alter the EncryptionConfiguation as follows:

```yaml
- secretbox:
        keys:
        - name: secret-key-2
          secret: xgI5XTIRQHN/C6mlS43MuAWTSzuwkGSvIDmEcw6DDl8=
        - name: secret-key-1
          secret: u7mcOcHKbFh9eVluB18hbFIsVfwpvgbXv650QacDYXA=
```

It is *crucial* that `secret-key-1` is not removed. While it will not be used for new encryption, it is used for existing secret objects, previously encrypted by it, for decryption! The removal of this key will prevent the API server from returning secret objects, such as `login2`, to clients. Since this key is first, it will be used for all new encryption. When secret objects are updated, they will be re-encrypted using this new key over time. Until then, the original key can remain in the list as a fallback decryption option. If you delete the key, you'll see the following responses from your client:

```
Error from server (InternalError): Internal error occurred: unable to transform
key "/registry/secrets/default/login1": no matching key was found for the
provided Secretbox transformer
```

Authors Recommendation: Understand New Vectors

With each step taken toward increasing your defense in depth, it is crucial you understand how attack vectors have shifted. The static-key encryption model is certainly more secure than no encryption. However, note that the encryption key lives on the

same host at the API server. Many Kubernetes deployments run etcd and the kube-apiserver on the same host, meaning with root access, the attacker could decrypt the data they query from etcd. Ideally, you should not rely solely on encryption at rest using a static key. If this is unacceptable, it may be time to consider using an external secret store or leveraging a KMS-plug-in. Both of these alternative approaches are covered in subsequent sections.

Envelope Encryption

Kubernetes 1.10 and later supports integrating with a KMS to achieve envelope encryption. Envelope encryption involves two keys: the key encryption key (KEK) and the data encryption key (DEK). KEKs are stored externally in a KMS and aren't at risk unless the KMS provider is compromised. KEKs are used to encrypt DEKs, which are responsible for encrypting Secret objects. Each Secret object gets its own unique DEK to encrypt and decrypt the data. Since DEKs are encrypted by a KEK, they can be stored with the data itself, preventing the kube-apiserver from needing to be aware of many keys. Architecturally, the flow of envelope encryption would look like the diagram shown in Figure 7-5.

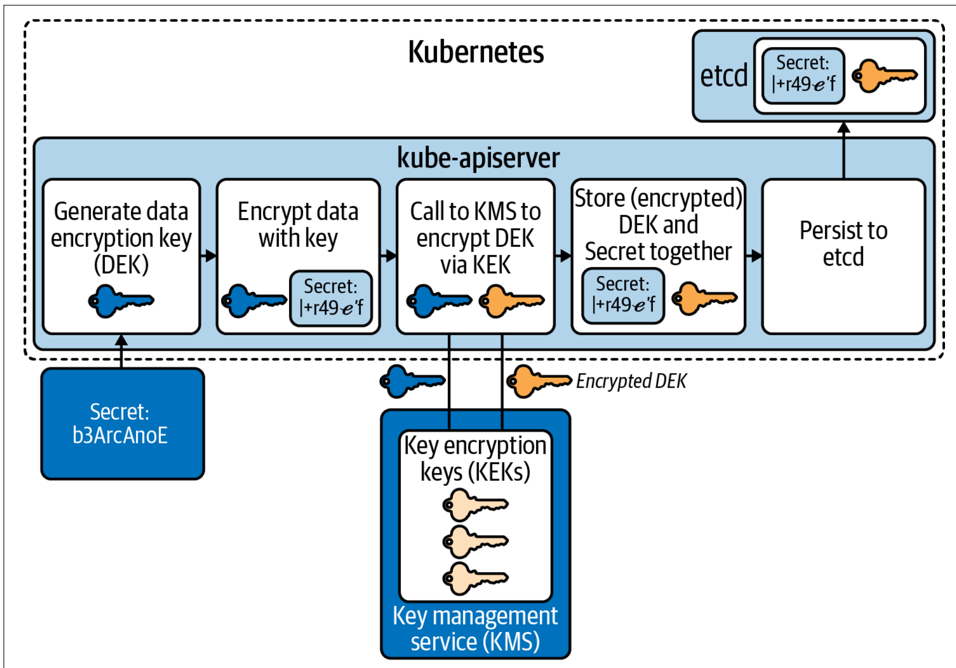


Figure 7-5. Flow to encrypt secrets using envelope encryption. The KMS layer lives outside the cluster.

There can be some variance in how this flow works, based on a KMS provider, but generally this demonstrates how envelope encryption functions. There are multiple benefits to this model:

- KMS is external to Kubernetes, increasing security via isolation.
- Centralization of KEKs enables easy rotation of keys.
- Separation of DEK and KEK means that secret data is never sent to or known by the KMS
- KMS is concerned only with decrypting DEKs.
- Encryption of DEKs means they are easy to store alongside their secret, making management of keys in relation to their secrets easy.

The provider plug-ins work by running a privileged container implementing a gRPC server that can communicate with a remote KMS. This container runs exclusively on master nodes where a kube-apiserver is present. Then, similar to setting up encryption in the previous section, an EncryptionConfiguration must be added to master nodes with settings to communicate with the KMS plug-in:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
- resources:
- secrets
providers:
- kms:
    name: myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
    timeout: 3s
# required, but not used for encryption
- identity: {}
```

Assuming the EncryptionConfiguration is saved on each master node at */etc/kubernetes/pki/secrets/encryption-config.yaml*, the kube-apiserver arguments must be updated to include the following:

```
--encryption-provider-config=/etc/kubernetes/pki/secrets/encryption-config.yaml
```

Changing the value should restart the kube-apiserver. If it doesn't, a restart is required for the change to take effect.

From a design perspective, this is a viable model. However, KMS plug-in implementations are scarce and the ones that do exist are immature. When we wrote this book, the following data points are true. There are no tagged releases for the aws-encryption-provider (AWS) or the k8s-cloudkms-plugin (Google). Azure's plug-in kubernetes-kms has notable limitations, such as no support for key rotation. So with the exception of running in a managed service, such as GKE where the KMS plug-in

is automatically available and supported by Google, usage may prove unstable. Lastly, the only cloud provider-agnostic KMS plug-in available was kubernetes-vault-kms-plugin, which was only partially implemented and has been archived (abandoned).

External Providers

Kubernetes is not what we'd consider an enterprise-grade secret store. While it does offer a Secret API that will be used for things like Service Accounts, for enterprise secret data it may fall short. There is nothing inherently wrong with using it to store application secrets, as long as the risks and options are understood, which is largely what this chapter has described thus far! However, many of our clients demand more than what the Secret API can offer, especially those working in sectors such as financial services. These users need capabilities such as integration with a hardware security module (HSM) and have advanced key rotation policies.

Our guidance is generally to start with what Kubernetes offers and see if the approaches to harden its security (i.e., encryption) are adequate. As described in the previous section, KMS encryption models that offer envelope encryption provide a pretty strong story around the safety of secret data in etcd. If we need to extend beyond this (and we often do), we then look to what secret management tooling preexists that the engineering team(s) have operational knowledge of. Running secret management systems in a production-ready capacity can be a challenging task, similar to running any stateful service where the data contained within needs to be not only highly available but protected from potential attackers.

Vault

Vault is an open source project by HashiCorp. It is by far the most popular project we run into with our clients when it comes to secret management solutions. Vault has found several ways to integrate in the cloud native space. Work has been done around providing first-class integration in frameworks such as Spring and in Kubernetes itself. One emerging pattern is to run Vault within Kubernetes and enable Vault to use the TokenReview API to authenticate requests against the Kubernetes API Server. Next, we'll explore two common Kubernetes integration points, including sidecar and initContainer injection along with a newer approach, CSI integration.

Cyberark

Cyberark is another popular option we see with clients. As a company, it's been around for a while, and often we find preexisting investments to exist and a desire to integrate Kubernetes with it. Cyberark offers a Credential Provider and Dynamic Access Provider (DAP). DAP provides multiple enterprise mechanisms Kubernetes administrators may want to integrate with. Similar to Vault, it supports the ability to use initContainers alongside your application to communicate with DAP.

Injection Integration

Once an external secret store is available to workloads in Kubernetes, there are several options for retrieval. This section covers these approaches, our recommendations, and trade-offs. We'll cover each design approach to consuming secrets and describe Vault's implementation.

This approach runs an initContainer and/or sidecar container to communicate with an external secret store. Typically, secrets are injected into the Pod's filesystem, making them available to all containers running in a Pod. We highly recommend this approach when possible. The major benefit is that it decouples the secret store entirely from the application. However, this does make the platform more complex, as facilitating secret injection is now an offering of the Kubernetes-based platform.

Vault's implementation of this model uses a MutatingWebhook pointed at a vault-agent-injector. As Pods are created, based on annotations, the vault-agent-injector adds an initContainer (used for retrieval of the initial secret) and a sidecar container to keep secrets updated, if needed. Figure 7-6 demonstrates this flow of interaction between the Pod and Vault.

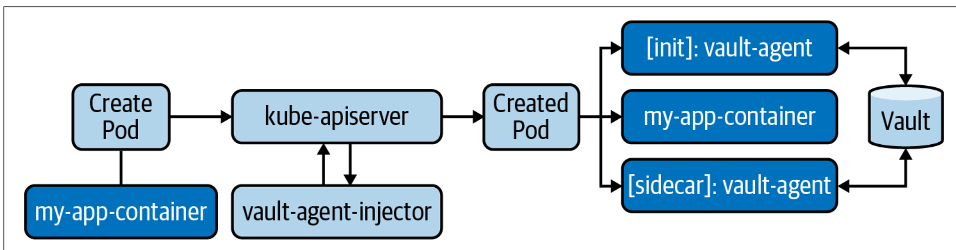


Figure 7-6. Sidecar injection architecture. Along with my-app-container, all Vault Pods are run as sidecars.

The configuration of the MutatingWebhook that will inject these vault-specific containers is as follows:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  labels:
    app.kubernetes.io/instance: vault
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: vault-agent-injector
  name: vault-agent-injector-cfg
webhooks:
- admissionReviewVersions:
  - v1beta1
  clientConfig:
    caBundle: REDACTED
    service:
```

```yaml
      name: vault-agent-injector-svc
      namespace: default
      path: /mutate
      port: 443
  failurePolicy: Ignore
  matchPolicy: Exact
  name: vault.hashicorp.com
  namespaceSelector: {}
  objectSelector: {}
  reinvocationPolicy: Never
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    - UPDATE
    resources:
    - pods
    scope: '*'
  sideEffects: Unknown
  timeoutSeconds: 30
```

The MutatingWebhook is invoked on every Pod CREATE or UPDATE event. While evaluation will occur on every Pod, not every Pod will be mutated, or injected with a vault-agent. The vault-agent-injector is looking for two annotations in every Pod spec:

`vault.hashicorp.com/agent-inject: "true"`
: Instructs the injector to include a vault-agent initContainer, which retrieves secrets and writes them to the Pod's filesystem, prior to other containers starting.

`vault.hashicorp.com/agent-inject-status: "update"`
: Instructs the injector to include a vault-agent sidecar, which runs alongside the workload. It will update the secret, should it change in Vault. The initContainer still runs in this mode. This parameter is optional and when it is not included, the sidecar is not added.

When the vault-agent-injector does a mutation based on `vault.hashicorp.com/agent-inject: "true"`, the following is added:

```yaml
initContainers:
- args:
  - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json
  - vault agent -config=/tmp/config.json
  command:
  - /bin/sh
  - -ec
  env:
  - name: VAULT_CONFIG
```

```yaml
    value: eyJhd
  image: vault:1.3.2
  imagePullPolicy: IfNotPresent
  name: vault-agent-init
  securityContext:
    runAsGroup: 1000
    runAsNonRoot: true
    runAsUser: 100
  volumeMounts:
  - mountPath: /vault/secrets
    name: vault-secrets
```

When the vault-agent-injector sees the annotation `vault.hashicorp.com/agent-inject-status: "update"`, the following is added:

```yaml
containers:
  #
  # ORIGINAL WORKLOAD CONTAINER REMOVED FOR BREVITY
  #
- name: vault-agent
  args:
  - echo ${VAULT_CONFIG?} | base64 -d > /tmp/config.json
  - vault agent -config=/tmp/config.json
  command:
  - /bin/sh
  - -ec
  env:
  - name: VAULT_CONFIG
    value: asdfasdfasd
  image: vault:1.3.2
  imagePullPolicy: IfNotPresent
  securityContext:
    runAsGroup: 1000
    runAsNonRoot: true
    runAsUser: 100
  volumeMounts:
  - mountPath: /vault/secrets
    name: vault-secrets
```

With the agents present, they will retrieve and download secrets based on the Pod annotations, such as the following annotation that requests a database secret from Vault:

```
vault.hashicorp.com/agent-inject-secret-db-creds: "serets/db/creds"
```

By default, the secret value will be persisted as if a Go map was printed out. Syntactically, it appears as follows. All secrets are put into */vault/secrets*:

```
key: map[k:v],
key: map[k:v]
```

To ensure that formatting of a secret is optimal for consumption, Vault supports adding templates into the annotation of Pods. This uses standard Go templating. For

example, to create a JDBC connection string, the following template can be applied to a secret named `creds`:

```
spec:
  template:
    metadata:
      annotations:
        vault.hashicorp.com/agent-inject: "true"
        vault.hashicorp.com/agent-inject-status: "update"
        vault.hashicorp.com/agent-inject-secret-db-creds: "secrets/db/creds"
        vault.hashicorp.com/agent-inject-template-db-creds: |
          {{- with secret "secrets/db/creds" -}}
          jdbc:oracle:thin:{{ .Data.data.username }}/{{ .Data.data.password }}
          {{- end }}
```

A primary area of complexity in this model is authentication and authorization of the requesting Pod. Vault provides several authentication methods. When running Vault within Kubernetes and especially in this sidecar injection model, you may wish to set up Vault to authenticate against Kubernetes so that Pods can provide their existing Service Account tokens as identity. Setting up this authentication mechanism appears as follows:

```
# from within a vault container

vault write auth/kubernetes/config \
    kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \ ❶
    kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    token_reviewer_jwt=\
    "$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" ❷
```

❶ This environment variable should be present in the Vault Pod by default.

❷ Location of this Pod's Service Account token, used to auth against the Kubernetes API server when performing TokenReview requests.

When requests for secrets enter Vault, the requester's Service Account can then be validated by Vault. Vault does this by communicating through the Kubernetes TokenReview API to validate the identity of the requester. Assuming the identity is validated, Vault must then determine whether the Service Account is authorized to access the secret. These authorization policies and bindings between Service Accounts and policies must be configured and maintained within Vault. In Vault, a policy is written as follows:

```
# from within a vault container
vault policy write team-a - <<EOF

path "secret/data/team-a/*" {
  capabilities = ["read"]
}
EOF
```

This has created a policy in Vault referred to as `team-a`, which provides read access to all the secrets within *secret/data/team-a/*:

```
vault policy list
default
team-a
root
```

The last step is to associate the requester's Service Account with the policy so Vault can authorize access:

```
vault write auth/kubernetes/role/database \
    bound_service_account_names=webapp \ ❶
    bound_service_account_namespaces=team-a \ ❷
    policies=team-a \ ❸
    ttl=20m ❹
```

❶ Name of the requester's Service Account.

❷ Namespace of the requester.

❸ Binding to associate this account to one or many policies.

❹ Duration the vault-specific authorization token should live for. Once expired, authn/authz is performed again.

The vault-specific process we have explored so far likely applies to any variety of external secret management stores. You'll be faced with some amount of overhead regarding integration of identity and authorization around secret access when dealing with systems beyond Kubernetes core.

CSI Integration

A newer approach to secret store integration is to leverage the secrets-store-csi-driver. At the time of this writing, this is a Kubernetes subproject within kubernetes-sigs. This approach enables integration with secret management systems at a lower level. Namely, it enables Pods to gain access to externally hosted secrets without running a sidecar or initContainer to inject secret data into the Pod. The result is secret interaction feeling more like a platform service and less like something applications need to integrate with. The secrets-store-csi-driver runs a driver Pod (as a DaemonSet) on every host, simliar to how you'd expect a CSI driver to work with a storage provider.

The driver then relies on a provider that is responsible for secret lookup in the external system. In the case of Vault, this would involve installing the `vault-provider` binary on every host. The binary location is expected to be where the driver's `provider-dir` mount is set. This binary may preexist on the host or, most commonly,

it is installed via a DaemonSet-like process. The overall architecture would appear close to what's shown in Figure 7-7.

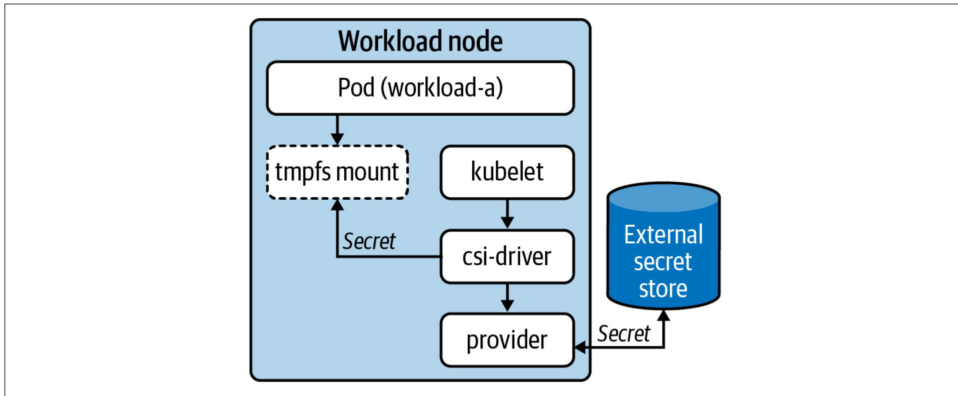


Figure 7-7. CSI driver interaction flow.

This is a fairly new approach that seems promising based on its UX and ability to abstract secret providers. However, it does pose additional challenges. For example, how is identity handled when the Pod itself is not requesting the secret? This is something the driver and/or provider must figure out since they're making requests on behalf of the Pod. For now, we can look at the primary API, which includes the SecretProviderClass. To interact with an external system such as Vault, the SecretProviderClass would look as follows:

```
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: apitoken
spec:
  provider: vault
  parameters:
    roleName: "teama"
    vaultAddress: "https://vault.secret-store:8000" ❶
    objects:  |
      array:
        - |
          objectPath: "/secret/team-a" ❷
          objectName: "apitoken" ❸
          objectVersion: ""
```

❶ This is the location of Vault and would have the Service name (`vault`) followed by the Namespace `secret-store`.

❷ This is the path in Vault the Key/Value object was written to.

❸ This is the actual object to lookup in `team-a`.

With the SecretProviderClass in place, a Pod can consume and reference this as follows:

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: busybox
spec:
  containers:
  - image:
    name: busybox
    volumeMounts:
    - name: secrets-api
      mountPath: "/etc/secrets/apitoken"
      readOnly: true
  volumes:
    - name: secrets-api
      csi:
        driver: secrets-store.csi.k8s.com
        readOnly: true
        volumeAttributes:
          secretProviderClass: "apitoken"
```

When this Pod starts, the driver and provider attempt to retrieve the secret data. The secret data will appear in a volume mount as any Kubernetes secret would, assuming authentication and authorization to the external provider is successful. From the driver Pod on the node, you can examine the logs to see the command sent to the provider:

```
level=info msg="provider command invoked: /etc/kubernetes/
secrets-store-csi-providers/vault/provider-vault --attributes [REDACTED]
--secrets [REDACTED] [--targetPath /var/lib/kubelet/pods/
643d7d88-fa58-4f3f-a7eb-341c0adb5a88/volumes/kubernetes.io~csi/
secrets-store-inline/mount --permission 420]"
```

In summary, secret-store-csi-drive is an approach worth keeping an eye on. Over time, if the project stabilizes and providers begin to mature, we could see the approach becoming common for those building application platforms on top of Kubernetes.

Secrets in the Declarative World

A common aspiration of application deployments, continuous integration, and continuous delivery is to move purely into a declarative model. This is the same model used in Kubernetes where you declare a desired state and over time controllers work to reconcile the desired state with current state. For application developers and DevOps teams, these aspirations commonly surface in a pattern called GitOps. A

central tenet of most GitOps approaches is to use one or many git repositories as the source of truth for workloads. When a commit is seen on some branch or tag, it can be picked up by build and deploy processes, often inside a cluster. This eventually aims to make the available workload capable of receiving traffic. Models such as GitOps are covered at greater length in Chapter 15.

When taking a purist-declarative approach, secret data creates a unique challenge. Sure you can commit your configurations alongside your code, but what about the credentials and keys used by your application? We have a feeling an API key showing up in a commit might make some people unhappy. There are some ways around this. One is, of course, to keep secret data outside of this declarative model and repent your sins toward the GitOps gods. Another is to consider "sealing" your secret data, in a way that accessing the data exposes nothing about the meaningful value, which is what we'll explore in the next section.

Sealing Secrets

How can we truly seal a secret? The concept is nothing new. Using asymmetric cryptography, we can ensure a way to encrypt secrets, commit them to places, and not worry about anyone exposing the data. In this model, we have an encryption key (typically public) and a decryption key (typically private). The idea is that any secret created by the encryption key cannot have its value compromised without the private key being compromised. Of course, we need to ensure many things to stay safe in this model, such as choose a cipher we can trust, ensure the private key is *always* safe, and establish both encryption key and secret data rotation policies. A model we'll explore in the coming sections is how this looks when a private key is generated in the cluster, and developers can be distributed their own encryption key that they can use on their secret data.

Sealed Secrets Controller

Bitnami-labs/sealed-secrets is a commonly used, open source project for achieving what has been described. However, should you choose alternative tooling or build something yourself, the key concepts are unlikely to change drastically.

The key component to this project is a sealed-secret-controller that runs inside the cluster. By default, it generates the keys needed to perform encryption and decryption. On the client side, developers use a command-line utility called kubeseal. Being that we're using asymmetric encryption, kubeseal needs to know only about the public key (for encryption). Once developers encrypt their data using it, they won't even be able to decrypt the values directly. To get started, we first deploy the controller to the cluster:

```
kubectl apply -f
https://github.com/bitnami-labs/sealed-secrets/releases/\
download/v0.9.8/controller.yaml
```

By default, the controller will create encryption and decryption keys for us. However, it is possible to bring your own certificates. The public (cert) and private (key) are stored in a Kubernetes Secret under *kube-system/sealed-secret-key*. The next step is allowing developers to retrieve the encryption key so they can get to work. This should *not* be done by accessing the Kubernetes Secret directly. Instead, the controller exposes an endpoint that can be used to retrieve the encryption key. How you access this service is up to you, but clients need to be able to call it using the following command, which has its flow detailed in Figure 7-8:

```
kubeseal --fetch-cert
```

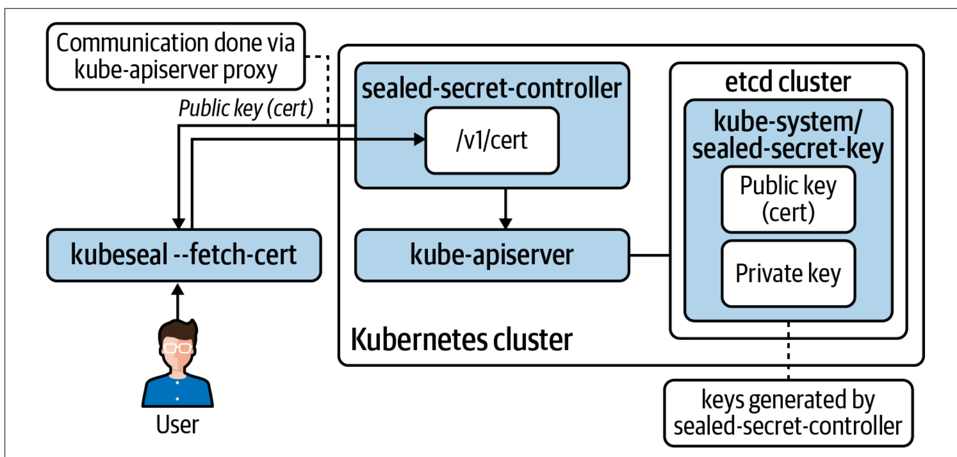


Figure 7-8. Sealed-secret-controller architecture.

Once the public key is loaded in kubeseal, you can generate SealedSecret CRDs that contain (encrypted) secret data. These CRDs are stored in etcd. The sealed-secret-controller makes the secrets available using standard Kubernetes Secrets. To ensure SealedSecret data is converted to a Secret correctly, you can specify templates in the SealedSecret object.

You can start with a Kubernetes Secret, like any other:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  dbuser: aGVwdGlvCg==
  dbkey: YmVhcmNhbm9lCg==
```

To "seal" the secret, you can run kubeseal against it and generate an encrypted output in JSON:

```
kubeseal mysecret.yaml

{
  "kind": "SealedSecret",
  "apiVersion": "bitnami.com/v1alpha1",
  "metadata": {
    "name": "mysecret",
    "namespace": "default",
    "creationTimestamp": null
  },
  "spec": {
    "template": {
      "metadata": {
        "name": "mysecret",
        "namespace": "default",
        "creationTimestamp": null
      },
      "type": "Opaque"
    },
    "encryptedData": {
      "dbkey": "gCHJL+3bTRLw6vL4Gf......",
      "dbuser": "AgCHJL+3bT......"
    }
  },
  "status": {

  }
}
```

The preceding SealedSecret object can be placed anywhere. As long as the sealing key, held by the sealed-secret-controller is not compromised, the data will be safe. Rotation is especially important in this model, which is covered in a subsequent section.

Once applied, the flow and storage look as described in Figure 7-9.

The Secret object made by the sealed-secret-controller is owned by its corresponding SealedSecret CRD:

```
ownerReferences:
- apiVersion: bitnami.com/v1alpha1
  controller: true
  kind: SealedSecret
  name: mysecret
  uid: 49ce4ab0-3b48-4c8c-8450-d3c90aceb9ee
```

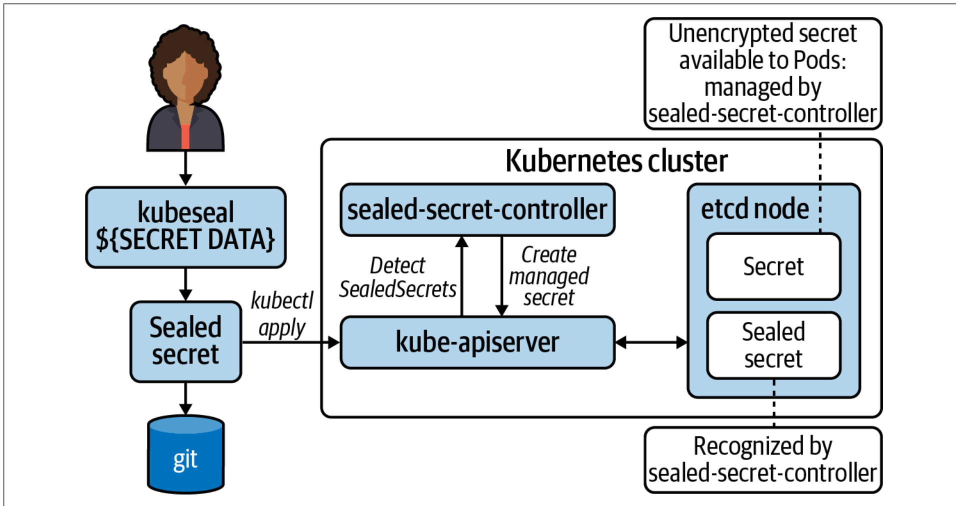


Figure 7-9. Sealed-secret-controller interaction around managing sealed and unsealed secrets.

This means that if the SealedSecret is deleted, its corresponding Secret object will be garbage collected.

Key Renewal

If the sealed-secret private key is leaked (perhaps due to RBAC misconfiguration), every secret should be considered compromised. It's especially important that the sealing key is renewed on an interval and that you understand the scope of "renewal." The default behavior is for this key to be renewed every 30 days. It does not replace the existing keys; instead, it is appended to the existing list of keys capable of unsealing the data. However, the new key is used for all new encryption activity. Most importantly, existing sealed secrets are not re-encrypted.

In the event of a leaked key, you should:

- Immediately rotate your encryption key.
- Rotate all existing secrets.
- Remember that just re-encrypting isn't good enough. For example, someone could easily go into git history, find the old encrypted asset, and use the compromised key on it. Generally speaking, you should have rotation and renewal strategies for passwords and keys, respectively.

SealedSecrets uses a trick where the Namespace is used during encryption. This provides an isolation mechanic where a SealedSecret truly belongs to the Namespace it was created in and cannot just be moved between them. Generally, this default behavior is the most secure and should just be left as is. However, it does support configurable access policies, which are covered in the sealed-secrets documentation.

Multicluster Models

Another key consideration for sealed-secret models is deployment topologies involving many clusters. Many of these topologies treat clusters ephemerally. In cases such as these, it may be harder to run sealed-secret-style controllers because—unless you are sharing private keys among them all—you now need to worry about having unique keys for each cluster. Additionally, the point of interaction a developer has to get the encryption key (as described in previous sections) goes from one cluster to many. While by no means an impossible problem to solve, it is worth considering.

> **Know the Scope!**
>
> When working with clients, we often run into a misunderstanding around what SealedSecrets solves for. Commonly, there is the perception that you can use a SealedSecret approach as an alternative to encrypting etcd or running an enterprise-grade secret store such as Vault. These concerns are *not* what sealed secrets aim to solve! This approach enables us to encrypt and safely store data in git repositories. However, the private key and unencrypted secrets *still* end up in Kubernetes. This means, in the absence of taking any steps beyond Kubernetes Secret API defaults, they will exist in an unencrypted state (at the application layer). In summary, be sure to know the scope of this and all other solutions talked about in this chapter!

Best Practices for Secrets

Application consumption of secrets is highly dependent on the language and frameworks at play. While variance is high, there are general best practices we recommend and encourage application developers to consider.

Always Audit Secret Interaction

A Kubernetes cluster should be configured with auditing enabled. Auditing allows you to specify the events that occur around specific resources. This will tell you when and by whom a resource was interacted with. For mutations, it will also detail what changed. Auditing secret events is critical in reacting to access issues. For details about auditing, see the cluster audit documentation.

Don't Leak Secrets

While leaking secrets is never desirable, in multitenant Kubernetes environments it's important to consider how secrets can be leaked. A common occurrence is to accidentally log a secret. For example, we have seen this a few times when platform engineers build operators (covered in Chapter 11). These operators often deal with secrets for the systems they are managing and potentially external systems they need to connect to. During the development phase, it can be common to log this secret data for the sake of debugging. Logs go to stdout/stderr and are, in many Kubernetes-based platforms, forwarded to a log analysis platform. This means the secret may pass in plain text through many environments and systems.

Kubernetes is primarily a declarative system. Developers write manifests that can easily contain secret data, especially when testing. Developers should work with caution to ensure that secrets used while testing don't get committed into source control repositories.

Prefer Volumes Over Environment Variables

The most common ways to access secrets provided by Kubernetes is to propagate the value into an environment variable or volumes. For most applications, volumes should be preferred. Environment variables can have a higher chance of being leaked through various means—for example, an echo command performed while testing or a framework automatically dumping environment variables on startup or during a crash. This doesn't mean these concerns are inherently solved for with volumes!

Security aside, the key benefit to app developers is that when secrets change, volumes are automatically updated; this will enable hot-reloading of secrets such as tokens. For a secret change to take place with environment variables, Pods must be restarted.

Make Secret Store Providers Unknown to Your Application

There are several approaches an application can take to retrieve and consume its required secrets. These can range from calling a secret store within business logic to expecting an environment variable to be set on startup. Following the philosophy of separation of concerns, we recommend implementing secret consumption in a way that whether Kubernetes, Vault, or other providers are managing the secret does not matter to the application. Achieving this makes your application portable and platform agnostic, and it reduces the complexity of your app's interaction. Complexity is reduced because for an application to retrieve secrets from a provider it needs to both understand how to talk to the provider and be able to authenticate for communication with the provider.

To achieve this provider-agnostic implementation, applications should prefer loading secrets from environment variables or volumes. As we said earlier, volumes are the most ideal. In this model, an application will assume the presence of secrets in one or many volumes. Since volumes can be updated dynamically (without Pod restart) the application can watch the filesystem if a hot-reload of secrets is desired. By consuming from the container's local filesystem, it does not matter whether the backing store is Kubernetes or otherwise.

Some application frameworks, such as Spring, include libraries to communicate directly to the API server and auto-inject secrets and configuration. While these utilities are convenient, consider the points just discussed to determine what approaches hold the most value to your application.

Summary

In this chapter we've explored the Kubernetes Secret API, ways to interact with secrets, means of storing secrets, how to seal secrets, and some best practices. With this knowledge, it's important we consider the amount of depth we're interested in protecting, and with that, determining how to prioritize solving for each layer.

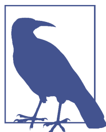
CHAPTER 8
Admission Control

We have written many times in this book about the flexible, modular design of Kubernetes being one of its great strengths. Sensible defaults can be replaced, augmented, or built upon to provide alternative or more fully featured experiences for platform consumers. Admission control is one area that particularly benefits from this flexible design goal. Admission control is concerned with validating and mutating requests to the Kubernetes API server *before* they are persisted in etcd. This ability to intercept objects with fine granularity and control opens up a number of interesting use cases. For example:

- Ensuring that new objects cannot be created in a Namespace that is currently being deleted (in terminating state)
- Enforcing that new Pods are not going to run as the root user
- Making sure that the total sum of memory used by all the Pods in a Namespace does not exceed a user-defined limit
- Ensuring that Ingress rules cannot be overwritten accidentally
- Adding a sidecar container to every Pod (e.g., Istio)

First we'll take a high-level look at the admission chain, which is the process all requests to the API server go through. Then we'll move on to cover the in-tree controllers. These are built-in admission controllers that can be enabled and disabled via flags to the API server and enable some of the preceding use cases. Other use cases require more custom implementation and are integrated via a flexible webhook model. We'll dedicate a lot of time to diving into the webhook model as it provides the most powerful and flexible options for integrating admission control into a cluster. Lastly, we'll finish by covering Gatekeeper, which is an opinionated open source

project that implements the webhook model and provides additional user-friendly functionality.



Further into this chapter we'll dive into some code written in the Go programming language. Kubernetes and many other cloud native tools are implemented in Go due to its rapid speed of development, strong concurrency primitives, and clean design. It's not necessary to know Go to understand most of this chapter (but we'd advise you to look into it if you're interested in Kubernetes tooling), and we will discuss the trade-off of needing development skills when weighing custom versus off-the-shelf tooling choices.

The Kubernetes Admission Chain

Before we look closer at the functionality and mechanics of individual controllers, let's first understand the flow of requests to and from the Kubernetes API server as shown in Figure 8-1.

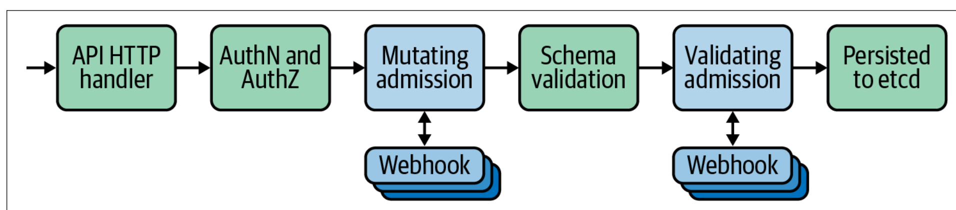


Figure 8-1. Admission chain.

Initially when requests arrive at the API server they are authenticated and authorized to ensure that the client is valid and able to perform the requested action (e.g., create a Pod in a specific Namespace) according to any configured RBAC rules.

In the next stage, requests pass through mutating admission controllers represented by the leftmost blue box in Figure 8-1. These can be built-in controllers or calls to external (out-of-tree) mutating webhooks (we'll discuss these later in the chapter). These controllers are able to modify the resource attributes before they pass onto future phases. As an example of why this might be useful, let's consider the Service Account controller (which is built in and enabled by default). When a Pod is submitted, the Service Account controller inspects the Pod's spec to ensure that it has the `serviceAccount` (SA) field set. If not, then it adds the field and sets it to the `default` SA for the Namespace. It also adds `ImagePullSecrets` and a Volume to allow the Pod to access its Service Account token.

Requests then undergo schema validation to ensure that the object being submmitted is valid according to the defined schema. Here it ensures things like mandatory fields are set. This ordering is important as it means we can set fields in mutating admission controllers before the object is validated.

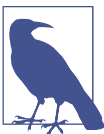
The final stage before the object is persisted to etcd is for it to pass through validating admission controllers, represented by the rightmost blue box in Figure 8-1. These can be built-in controllers or calls to external (out-of-tree) validating webhooks (we'll briefly cover these later in the chapter). These validating controllers differ from mutating controllers in the sense that they are only able to admit or reject the request, *not* modify the payload. They differ from the prior *schema validation* step in that they are concerned with validating against operational logic, not a standardized schema.

An example validating admission controller is the `NamespaceLifecycle` controller. It has several jobs related to Namespaces, but the one we'll take a look at is its responsibility to reject requests for new objects to be created in a Namespace that is currently being deleted. We can see the behavior in this code snippet:

```
// ensure that we're not trying to create objects in terminating Namespaces
if a.GetOperation() == admission.Create {
  if namespace.Status.Phase != v1.NamespaceTerminating {
    return nil ❶
  }

  err := admission.NewForbidden(a, fmt.Errorf("unable to create new content in
  namespace %s because it is being terminated", a.GetNamespace()))
  if apierr, ok := err.(*errors.StatusError); ok {
    apierr.ErrStatus.Details.Causes = append(apierr.ErrStatus.Details.Causes,
    metav1.StatusCause{
      Type:    v1.NamespaceTerminatingCause,
      Message: fmt.Sprintf("namespace %s is being terminated", a.GetNamespace()),
      Field:   "metadata.namespace",
    })
  }
  return err ❷
}
```

❶ If the operation is a Create but the Namespace is currently *not* terminating, return no error. The request would pass this controller.

❷ Else, return an API error stating that the Namespace is being terminated. If an error is returned, the request is denied.



For a request to pass and the object to be persisted into etcd it must be admitted by *all* validating admission controllers. For it to be denied, only *one* controller needs to reject it.

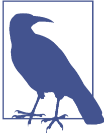
In-Tree Admission Controllers

When Kubernetes was first released there was only a minimal number of interfaces for users to *plug in* or extend external functionality, such as the Container Network Interface (CNI). Other integrations with cloud providers, storage providers, and the implementation of admission controllers were all baked into the core Kubernetes code base and often described as being *in-tree*. Over time the project has sought to increase the number of pluggable interfaces, and we have seen the creation of the Container Storage Interface (CSI) and the movement toward external cloud providers.

Admission controllers are one area where many core features are still in-tree. Kubernetes ships with many different admission controllers that can be enabled or disabled by configuring API server flags. This model has proved problematic for those users of cloud-managed Kubernetes platforms who historically did not have access to configure those flags. PodSecurityPolicy (PSP) is an example of a controller that enables advanced and robust security capabilities across the cluster but is *not* enabled by default, therefore excluding users from benefitting from it.

However, admission control is slowly following the trend of shifting code out of the API server and moving toward increased pluggability. The start of this process came with the addition of mutating and validating webhooks. These are two flexible admission controllers that allow us to specify that the API server should forward requests (that match specific criteria) and delegate admission decisions to external webhooks. We will discuss these in greater detail in the next section.

Another step in this process is the announced deprecation of the current PodSecurityPolicy built-in controller. Although there are multiple approaches to replace it, we think the implementation of PSPs will be delegated to an external admission controller, as the community continues to move code out of tree. In fact, we believe that more of the built-in admission controllers will be eventually moved out of tree. These would be replaced either by recommendations to utilize third-party tooling or standardized components that live in the Kubernetes upstream organization but not the core code base, thereby allowing users a sane default choice with the ability to replace if necessary.



A subset of the built-in admission controllers is enabled by default. These are intended as a set of *sane defaults* that should work well for most clusters. We won't replicate the list here, but you should take care to ensure the controllers you need are enabled. Also note that the UX for this feature can be a little confusing. To enable additional (nondefault) controllers you must use the `--enable-admission-plugins` flag to the API server, and to *disable* default controllers you must specify the `--disable-admission-plugins` list parameter.

There is a lot of good information on the in-tree controllers available in the official Kubernetes documentation, so we're not going to cover much more on them here. The real power of admission controllers is enabled by the two special validating and mutating webhooks, which is where we're headed next!

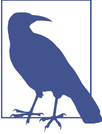
Webhooks

All admission controllers sit in the *critical path* for requests going to the Kubernetes API server. They have varying scopes, so not all requests may be intercepted, but you should definitely be aware of this when enabling and/or injecting them. This is especially relevant when discussing webhook admission controllers for two reasons. First, they have added latency as they reside out of tree and must be called via HTTPS. Second, they have a broad potential scope of functionality, maybe even calling out to third-party systems. Great care should be taken to make admission controllers perform as efficiently as possible, returning at the earliest opportunity.

Webhooks are a special type of admission controller. We can configure the Kubernetes API server to send an API request to external webhook endpoints and receive a decision (whether the original request should be allowed, denied, or altered/mutated) response. This is incredibly powerful for a number of reasons:

- The receiving web server can be written in any language that can expose an HTTPS listener. We can take advantage of web frameworks, libraries, and expertise that may be available to us to implement any logic we need to make admission decisions.
- They can be run in or out of cluster. We can take advantage of the discovery and operator primitives that are available to us if we want to run them in-cluster, or we can implement reusable functionality in a serverless function, for example.

- We are able to make callouts to systems and datastores external to Kubernetes to make policy decisions. For instance, we could query a centralized security system to check if specific images were approved for use in Kubernetes manifests.



The API server will call webhooks over TLS, so webhooks must present certificates trusted by the Kubernetes API. This is often achieved by deploying Cert Manager into the cluster and automatically generating certificates. If running out of cluster, you will need to provision certificates that are trusted by the Kubernetes API server, either from a public root CA or some internal CA that Kubernetes is aware of.

For the webhook model to work, there must be a defined schema for the request and response messages exchanged between the API server and the webhook server. In Kubernetes this is defined as an AdmissionReview object and is a JSON payload that contains information about the request, including:

- API version, group, and kind
- Metadata such as the name and Namespace, and a unique ID to correlate it with the response decision
- The operation attempted (e.g., CREATE)
- Information about the user initiating the request including their group membership
- Whether this is a *dry run* request (this is important as we'll see later when we discuss design considerations)
- The actual resource

All of this information can be used by the receiving webhook to calculate an admission decision. Once decided, the server needs to respond with an AdmissionReview message of its own (this time with a `response` field). It will contain:

- The unique ID from the request (for correlation)
- Whether the request should be allowed to proceed
- An optional customized error status and message

Validating webhooks are not able to modify the requests sent to them and can admit or reject only the original object. This restriction makes them fairly limited; however, they are a good fit when ensuring that objects applied to the cluster conform to security standards (specific user IDs, no host mounts, etc.) or contain all required metadata (internal team labels, annotations, etc.).

In the case of a *mutating* webhook the response structure can also include a patch set (if desired). This is a base64-encoded string containing a valid JSONPatch structure encapsulating the changes that should be made to the request before it is admitted to the API server. If you want a more detailed explanation of all the fields and structure for AdmissionReview objects, then the official documentation does a great job here.

A simple example of a mutating controller might be one that adds a set of labels containing team- or workload-specific metadata to Pods or Deployments. Another more complex but common use of mutating controllers you may come across is the injection of a sidecar proxy in many service mesh implementations. The way this works is that the service mesh (Istio in this case) runs an admission controller that mutates Pod specs to add a sidecar container that will participate in the data plane of the mesh. This injection occurs by default but can be overridden by annotations at the Namespace or Pod level to provide additional control.

This model is an effective way of enriching Deployments with additional functionality but hiding that complexity to improve the end-user experience. However, as with many decisions this can be a double-edged sword. One downside of mutating controllers is that visibility is removed from the end user, with objects being applied to the cluster that are not consistent with those that they originally created, potentially causing confusion if the user is unaware that mutating controllers are in operation on the cluster.

Configuring Webhook Admission Controllers

Cluster administrators can use the MutatingWebhookConfiguration and ValidatingWebhookConfiguration kinds to specify the configuration of dynamic webhooks. Following is an annotated example briefly describing the relevant sections. We'll dig into some of the more advanced considerations for some of these fields in the following section:

```
apiVersion: admissionregistration.k8s.io/v1
kind: MutatingWebhookConfiguration
metadata:
  name: "test-mutating-hook"
webhooks:
- name: "test-mutating-hook"
  rules: ❶
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"] ❷
    resources:   ["pods"] ❸
    scope:       "Namespaced" ❹
  clientConfig: ❺
    service:
      namespace: test-ns
      name: test-service
```

```
      path: /test-path
      port: 8443
    caBundle: "Ci0tLS0tQk...tLS0K" ❻
  admissionReviewVersions: ["v1", "v1beta1"] ❼
  sideEffects: "None" ❽
  timeoutSeconds: "5" ❾
  reinvocationPolicy: "IfNeeded" ❿
  failurePolicy: "Fail" ⓫
```

❶ Matching rules. What API/kind/version/operations this webhook should be sent.

❷ The operations that should trigger a call to the webhook.

❸ Which kind to target.

❹ Whether Namespace-scoped or cluster-scoped resources should be targeted.

❺ Describes how the API server should connect to the webhook. In this case it's in cluster at *test-service.test-ns.svc*.

❻ A PEM encoded CA bundle that will be used to validate the webhook's server certificate.

❼ Declare the `admissionReviewVersions` that the webhook supports.

❽ Describes whether the webhook has external side effects (calls/dependencies to external systems).

❾ How long to wait until triggering the `failurePolicy`.

❿ Whether this webhook can be re-invoked (this may happen after other webhooks have been called).

⓫ Whether the webhook should fail *open* or *closed*. This has security implications.

As you can see in the preceding configuration, we can be very granular about selecting which requests we want to intercept with our admission webhooks. For example, if we wanted to target only requests that are creating Secrets, we could use the following rule:

```
# <...snip...>
rules: ❶
- apiGroups:   [""]
  apiVersions: ["v1"]
  operations:  ["CREATE"] ❷
  resources:   ["secrets"] ❸
```

```yaml
  scope:       "Namespaced" ❹
# <...snip...>
```

We can additionally combine this with Namespace or object selectors, which enable further granularity. These allow us to specify any number of Namespaces to target and/or objects with specific labels; for instance, in the following snippet we are choosing only Secrets that are in Namespaces that have label of `webhook: enabled`:

```yaml
# <...snip...>
namespaceSelector:
  matchExpressions:
  - key: webhook
    operator: In
    values: ["enabled"]
# <...snip...>
```

Webhook Design Considerations

There are several factors to be mindful of when writing and implementing admission webhooks. We'll talk more in detail about how these impact some real-world scenarios in the next section, but at a high level you should be aware of the following concerns:

Failure modes

If a webhook is unreachable or sends an unknown response back to the API server, it is treated as failing. Administrators must choose whether to fail *open* or *closed* in this situation by setting the `failurePolicy` field to `Ignore` (allow the request) or `Fail` (reject the request).

For security-related (or critical functionality) webhooks, `Fail` is the safest option. For noncritical hooks `Ignore` may be safe (potentially in conjunction with a reconciling controller as a backup). Combine these recommendations with those discussed under the performance item in this list.

Ordering

The first thing to note with regards to API server request flow is that mutating webhooks will all be called (potentially more than once) *before* validating webhooks are called. This is important because it enables validating webhooks (which may reject a request based on security requirements) always to see the *final* version of a resource before it is applied.

Mutating webhooks are not guaranteed to be called in a specific order and may be called multiple times if subsequent hooks modify a request. This can be modified by specifying the `reinvocationPolicy`, but ideally webhooks should be designed for idempotency to ensure ordering does not affect their functionality.

Performance

Webhooks are called as part of the critical path of requests flowing to the API server. If a webhook is critical (security-related) and fails closed (if a timeout occurs, the request is denied), then it should be designed with high availability in mind. As one of our esteemed former colleagues often comments, admission control can become *bottleneck-as-a-service* if users are not careful in its application.

If a webhook is resource-intensive and/or has external dependencies, consideration should be taken for how often the hook will be called, and the performance impact of adding the functionality into the critical path. In these situations a controller that reconciles objects once in-cluster may be preferable. When writing webhook configurations you should try to scope them down as tightly as possible to ensure they are not called unnecessarily or on irrelevant resources.

Side effects

Some webhooks may be responsible for modifying external resources (e.g., some resource in a cloud provider) based on a request to the Kubernetes API. These webhooks should be aware of and respect the `dryRun` option and skip external state modification when it is enabled. Webhooks are responsible for declaring that they either have no side effects or respect this option by setting the `sideEffects` field. More information on the valid options for this field and the behavior of each option is detailed in the official documentation.

Writing a Mutating Webhook

In this section we'll take a look at two approaches for writing a mutating admission webhook. First we'll talk briefly about implementing one with a plain HTTPS handler that is language agnostic. Then we'll take a deeper dive into a real use case while covering the controller-runtime upstream project, which is designed to help teams develop Kubernetes controller components.

Both of the solutions in this section require expertise in either Go (for controller-runtime) or another programming language. There are cases where this requirement is an impediment to creating and implementing admission controllers. If your teams don't have the experience or need to write bespoke webhooks, the final section in this chapter offers a solution for configurable admission policies that don't require programming knowledge.

Plain HTTPS Handler

One of the advantages of the webhook model for admission controllers is that we're able to implement them from scratch in any language. The examples we're using here are written in Go, but any language capable of TLS-enabled HTTP handling and JSON parsing is acceptable.

This way of writing a webhook provides the most flexibility to integrate with the current stacks in use but comes at the cost of many high-level abstractions (although languages with mature Kubernetes client libraries can alleviate this).

As described in the introduction to this section, admission control webhooks receive and return requests from and to the API server. The schema of these messages is well known, so it's possible to receive the request and modify the object (via patches) manually.

For a concrete example of this approach let's take an in-depth look at the AWS IAM Roles for Service Accounts mutating webhook. This webhook is used to inject a Projected Volume into Pods with a Service Account token that can be used for authentication to AWS services. (See Chapter 10 for more detail on the security aspect of this use case.)

```
// <...snip...>
type patchOperation struct { ❶
  Op    string      `json:"op"`
  Path  string      `json:"path"`
  Value interface{} `json:"value,omitempty"`
}

volume := corev1.Volume{ ❷
  Name: m.volName,
  VolumeSource: corev1.VolumeSource{
    Projected: &corev1.ProjectedVolumeSource{
      Sources: []corev1.VolumeProjection{
        {
          ServiceAccountToken: &corev1.ServiceAccountTokenProjection{
            Audience:          audience,
            ExpirationSeconds: &m.Expiration,
            Path:              m.tokenName,
          },
        },
      },
    },
  },
}

patch := []patchOperation{ ❸
  {
    Op:    "add",
    Path:  "/spec/volumes/0",
```

```
      Value: volume,
    },
  }

  if pod.Spec.Volumes == nil { ❹
    patch = []patchOperation{
      {
        Op:   "add",
        Path: "/spec/volumes",
        Value: []corev1.Volume{
          volume,
        },
      },
    }
  }

  patchBytes, err := json.Marshal(patch) ❺
  // <...snip...>
```

❶ Define a `patchOperation` struct that will be marshaled to JSON for the response back to the Kubernetes API server.

❷ Construct the `Volume` struct with the relevant ServiceAccountToken content.

❸ Create an instance of the `patchOperation` with the Volume content previously constructed.

❹ If there are currently no Volumes, create that key and add the Volume content previously constructed.

❺ Create the JSON object containing the patch contents.

Note that the actual implementation for this admission webhook includes some additional functionality that also adds to the patch set (e.g., adding environment variables), but we're going to ignore that for the purposes of this example. After the patch set is complete we need to return an AdmissionResponse object that contains our patch set (the `Patch` field in the following snippet):

```
  return &v1beta1.AdmissionResponse{
    Allowed: true,
    Patch:   patchBytes,
    PatchType: func() *v1beta1.PatchType {
      pt := v1beta1.PatchTypeJSONPatch
      return &pt
    }(),
  }
```

We can see in this example that there is a lot of manual work to be done generating patch sets and constructing the appropriate response for the API server. This is even present when utilizing some of the Kubernetes libraries available in the Go language. However, there is a large amount of supporting code we've omitted that's required to handle errors, graceful shutdown, HTTP header handling, and so on.

While this approach affords us maximum flexibility, it requires more domain knowledge and is more complex to implement and maintain. This trade-off will be all-too-familiar for most readers. and care needs to be taken when evaluating your specific use cases and internal expertise.

In the next section we'll see an approach that removes a lot of the boilerplate and bespoke work in favor of implementing an upstream helper framework, controller-runtime.

Controller Runtime

In this section we're going to dive into the upstream project controller-runtime and see what abstractions it provides on top of the native Kubernetes client libraries to make writing admission controllers more streamlined. To provide more color we'll use an open source controller we built to satisfy a community requirement as a way of illustrating some of the advantages of controller-runtime and cover some of the techniques and pitfalls previously discussed. While we have simplified the functionality and code of the controller somewhat for brevity, the core underlying ideas remain.

Kubebuilder

The upstream repository contains examples that can help you get started implementing webhooks for built-in types (e.g., Pod, Deployment, etc.). If you want to implement webhooks for custom resources (CRDs), then the Kubebuilder project is probably more suitable for a holistic solution. Kubebuilder utilizes `controller-runtime` and provides additional generation utilities and helpers. We'll dive more into CRDs and Kubebuilder later in this book.

If you *are* using Kubebuilder, the project provides a convenient marker system that allows you to generate the relevant manifests to deploy your webhook to the Kubernetes cluster. For example:

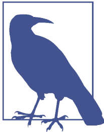
```
/* +kubebuilder:webhook:path=/infoblox-ipam,mutating=true,failurePolicy=fail,
groups="infrastructure.cluster.x-k8s.io",resources=vspheremachines,
verbs=create,versions=v1alpha3,
name=mutating.infoblox.ipam.vspheremachines.infrastructure.cluster.x-k8s.io */
```

The controller we'll be walking through is a webhook designed to perform the following actions:

1. Watch for Cluster API VSphereMachine objects.
2. Based on a configurable field, allocate an IP address in an external IPAM system (in this case, Infoblox).
3. Insert the allocated IP into the static IP field in the VSphereMachine.
4. Allow the mutated requested through to the Kubernetes API server to be actioned (by the Cluster API controller) and persisted into etcd.

This use case is a good candidate for a custom-built (using controller-runtime) mutating webhook for a couple of reasons:

- We need to mutate the request to add an IP address *before* the request hits the API server (otherwise it would error).
- We're calling out to an external system (Infoblox) and can therefore leverage its Go library for interactions.
- Small amount of boilerplate allows newer community and/or client developers to understand and extend the functionality.



Although out-of-scope for *this* chapter, we did accompany this webhook with a controller that runs in the cluster. This is important when your webhooks interact with and modify or depend on *external* state (Infoblox in this case) because you should be constantly reconciling that state, rather than relying on the state just seen at admission time. This is something to consider when building a mutating admission webhook and may increase the complexity of your solutions if the extra component is required.

Controller-runtime webhooks must implement a `Handle` method whose signature is:

```
func (w *Webhook) Handle(
  ctx context.Context,
  req admission.Request) admission.Response
```

The `admission.Request` object is an abstraction over the raw JSON that webhooks receive and provides easy access to the raw applied object, the operation being executed (e.g., `CREATE`), and many other useful pieces of metadata:

```
vm := &v1alpha3.VSphereMachine{} ❶
err := w.decoder.DecodeRaw(req.Object, vm) ❷
if err != nil {
  return admission.Errored(http.StatusBadRequest, err) ❸
}
```

❶ Create a new VSphereMachine object.

❷ Use the built-in decoder to decode the raw object from the request into the Go VSphereMachine object.

❸ Use the convenience method `Errored` to construct and return an error response if the decoding step returns an error.

The `vm` object from the request can be modified or validated in any way before the response is returned. In the following example we are checking to see if the `infoblox` annotation (denoting that our webhook should take action) is present on the VSphereMachine object. This is an important step to perform early on in the webhook because we can short-circuit out of any further logic if no action should be taken. If the annotation is not present, we utilize the convenience `Allowed` method to return the unmodified object through to the API server as quickly as possible. As we discussed earlier in "Webhook Design Considerations" on page 227, webhooks are on the critical path for API requests, and any actions we perform inside them should be as fast as possible:

```
if _, ok := vm.Annotations["infoblox"]; !ok {
  return admission.Allowed("Nothing to do")
}
```

Assuming we *should* handle this request and the preceding logic does not trigger, we retrieve an IP address from Infoblox (not shown) and write it directly into the `vm` object:

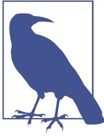
```
vm.Spec.VirtualMachineCloneSpec.Network.Devices[0].IPAddrs[0] = ipFromInfoblox ❶
marshaledVM, err := json.Marshal(vm) ❷
if err != nil { ❸
  return admission.Errored(http.StatusInternalServerError, err)
}
return admission.PatchResponseFromRaw(req.Object.Raw, marshaledVM) ❹
```

❶ Set the IP field on the `vm` object, thereby *mutating* it.

❷ Marshal the `vm` object to JSON ready to send back to the API server.

❸ If the marshaling fails we'll use the convenience error method we saw earlier.

❹ Another convenience method, `PatchReponseFromRaw`, sends back the response. We'll discuss this in more detail later.



There are use cases where you may want and/or need to intercept `DELETE` requests to the API server. An example of this might be to clean up some external state that might be tied to resources in the cluster. While this *can* be done in a webhook, you should consider whether you're failing open or closed and the risks of having misaligned state in the former case. Ideally, deletion logic should be implemented with a finalizer and a custom controller running in the cluster to guarantee cleanup.

In the preceding snippet we see another of controller-runtime's convenience methods, `PatchResponseFromRaw`. This method will automatically calculate the JSONPatch diffs required between the original raw object and the one we have been modifying before sending the correctly serialized response. Compared to the more manual approach covered in the previous section, this is a great way to remove some boilerplate and make our controller code leaner.

In the case of a simple validating hook, we can also leverage convenience functions like `admission.Allowed()` and `admission.Denied()` that can be used after processing the required logic.

If we're manipulating external state as part of an admission controller we need to be aware of and check for the `req.DryRun` condition. If this is set, the user is only executing a dry run, no-op request and we should ensure that our controller *does not* mutate external state in this case.

Controller-runtime provides a very strong foundation with which to build admission controllers, allowing us to focus on the logic we want to implement with minimal boilerplate. However, it does require programming expertise, and the admission logic is obfuscated inside the controller code, potentially leading to more confusing or unexpected results for end users.

In the next section of this chapter, we'll take a look at an emerging model that centralizes policy logic and introduces a standard language to author decision rules. The tools appearing in this area strive to combine the flexibility of a custom controller with greater usability features for less technical operators and/or end users.

Centralized Policy Systems

So far we've looked at a number of different methods for implementing and configuring admission controllers. Each has its own specific trade-offs that have to be considered when choosing to adopt them. In this final section, we're going to cover an emerging model of centralizing policy logic into one place and using standardized languages to express the allow/deny rules. This model has two main advantages:

- Programming knowledge is not required to create admission controllers, as we can express rules in a specific policy language (as opposed to a general-purpose programming language). This also means that changes to logic do not require rebuilding and redeploying the controller each time.
- Policies and rules are stored in a single location (in most cases, the cluster itself) for viewing, editing, and auditing.

This model is being built out and implemented by several open source tools and usually comprises two components:

- A policy/query language that can express conditions on whether an object should be admitted or rejected.
- A controller that sits in the cluster serving as an admission controller. The controller's job is to evaluate the policies/rules against objects coming into the API server and make admit or reject decisions.

For the rest of this chapter we're going to focus on the most popular implementation of this centralized policy model called Gatekeeper, although other tools such as Kyverno are also gaining traction. Gatekeeper is built on a lower-level tool called Open Policy Agent (OPA). OPA is an open source policy engine that applies policies written in the Rego language to ingested JSON documents and returns a result.

Rego Language

Rego is the declarative query language used by Open Policy Agent. It was created by the creators of OPA and is used as the policy language in the OPA engine. It is not designed to be a general-purpose programming language and is specialized for querying and performing logic operations over data structures. This approach results in a fairly lean syntax, but it can be difficult to read and write initially. We won't cover Rego syntax in detail in this book, but there is an online training portal where users can learn Rego and test their knowledge for free.

A calling application can utilize OPA by receiving the result and deciding how to proceed (making a policy decision). We know from this chapter that Kubernetes has a standard schema for sending requests and receiving admission decision responses, so it immediately seems like this is a promising fit. However, OPA itself is platform/context agnostic and is simply a policy engine that operates on JSON. We need a controller that will act as an interface between the OPA engine and Kubernetes. Gatekeeper is the tool that fulfills that interface role and provides some additional Kubernetes-native functionality around templates and extensibility to make it easier for platform operators to author and apply policies. Gatekeeper is deployed to the

cluster as an admission controller that allows users to write rules in Rego to make admission policy decisions about Kubernetes resources being applied to the cluster.

Gatekeeper enables a model whereby cluster operators can create and expose preset policy templates as `ConstraintTemplate` CRDs. These templates create new CRDs for the specific constraints that can accept custom input parameters (much like a function). This approach is powerful because end users can then create an instance of the constraint with their own values, which will be used by Gatekeeper as part of admission control to the cluster.

For some of the reasons detailed in the latter part of this section, you should be aware that Gatekeeper currently fails *open* by default. This can have serious security implications, and you should carefully understand the trade-offs (detailed in this chapter and in much of the official documentation) involved in each approach before implementing these solutions in production.

One common rule we've implemented in the field is ensuring that teams cannot overwrite existing Ingress resources. This is a requirement in most Kubernetes clusters, and some Ingress controllers (e.g., Contour) provide mechanisms to protect against this out of the box. However, if this is not the case with your tooling, you can use Gatekeeper to enforce this rule. This scenario is one of several maintained as a library of common policies in the official Gatekeeper documentation.

In this situation it's necessary to make a policy decision based on data that *exists externally to the object* that's being applied to the cluster. We need to query Kubernetes directly to know what Ingress resources already exist and to be able to inspect metadata around them to compare against the one being applied.

Let's take an even more complex example that builds on these ideas, and we'll walk through the implementation of each resource. In this case, we're going to annotate a Namespace with a regex pattern and ensure any Ingress applied in that Namespace conforms to the regex. We mentioned earlier that we need information about the cluster to be available to Gatekeeper to make policy decisions. This is achieved by defining a sync config to specify which resources in Kubernetes should be synchronized to Gatekeeper's cache (in order to provide that queryable data source):

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
  name: config
  namespace: "gatekeeper-system"
spec:
  sync: ❶
    syncOnly:
      - group: "extensions"
```

```
        version: "v1beta1"
        kind: "Ingress"
      - group: "networking.k8s.io"
        version: "v1beta1"
        kind: "Ingress"
      - group: ""
        version: "v1"
        kind: "Namespace"
```

❶ The sync section specifies all the Kubernetes resources that Gatekeeper should cache to assist with policy decisions.

The cache exists to remove the need for Gatekeeper to keep querying the Kubernetes API server for the required resources. However, there is potential for Gatekeeper to make decisions based on *stale* data. To mitigate this, there is an audit capability that intermittently runs policies against existing resources and records violations in the status field of each constraint. These should be monitored to ensure that violations that pass through (maybe as a result of a stale cache read) are not left unchecked.

Once the config is applied, then an administrator can create the ConstraintTemplate. This resource defines the main content of the policy and any input parameters that are available for administrators or other operators to provide/override:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: limitnamespaceingress
spec:
  crd:
    spec:
      names:
        kind: LimitNamespaceIngress
        listKind: LimitNamespaceIngressList
        plural: limitnamespaceingresss
        singular: limitnamespaceingress
      validation:
        # Schema for the `parameters` field in the constraint
        openAPIV3Schema:
          properties: ❶
            annotation:
              type: string
  targets: ❷
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package limitnamespaceingress

        violation[{"msg": msg}] {
```

```
          cluster := data.inventory.cluster.v1
          namespace := cluster.Namespace[input.review.object.metadata.namespace]
          regex := namespace.metadata.annotations[input.parameters.annotation]
          hosts := input.review.object.spec.rules[_].host
          not re_match(regex, hosts)
          msg := sprintf("Only ingresses matching %v in namespace %v allowed",
            [regex ,input.review.object.metadata.namespace])
        }
```

❶ The `properties` section defines the input parameters that will be available to inject into the Rego policy for each instantiation of the rule.

❷ The `targets` section contains the Rego code for our policy rules. We won't dig into Rego syntax here, but notice that the input parameter is being referenced via `input.parameters.<parameter_name>` (in this case, `annotation`).

The `annotation` in the custom input parameters allows the user to specify the specific annotation name that Gatekeeper should pull the regex pattern from. Rego won't trigger a violation if any statement returns `False`. In this case, we're checking that the hosts *do* match the regex, so to ensure that doesn't trigger a violation we need to invert the `re_match()` with `not` to ensure that a positive match doesn't trigger a violation and instead *allows* the request through admission control.

Lastly, we create an instance of the preceding policy to configure Gatekeeper to apply it against specific resources as part of admission control. The `LimitNamespaceIngress` object specifies that the rule should apply to Ingress objects for both `apiGroups` and designates `allowed-ingress-pattern` as the annotation that should be inspected for the regex pattern (this was the customizable input parameter):

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: LimitNamespaceIngress
metadata:
  name: limit-namespace-ingress
spec:
  match:
    kinds:
      - apiGroups: ["extensions", "networking.k8s.io"]
        kinds: ["Ingress"]
  parameters:
    annotation: allowed-ingress-pattern
```

Finally, the Namespace object itself is applied with the custom annotation and pattern. Here we are specifying the regex `\w\.my-namespace\.com` in the `allowed-ingress-pattern` field:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    # Note regex special character escaping
    allowed-ingress-pattern: \w\.my-namespace\.com
  name: ingress-test
```

The setup steps are all now complete. We can begin adding Ingress objects, and the rules we have configured will evaluate against them and either allow or deny the persistence/creation of the Ingress:

```yaml
# FAILS because the host doesn't match the pattern above
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-1
  namespace: ingress-test
spec:
  rules:
  - host: foo.other-namespace.com
    http:
      paths:
      - backend:
          serviceName: service1
          servicePort: 80
---
# SUCCEEDS as the pattern matches
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-2
  namespace: ingress-test
spec:
  rules:
  - host: foo.my-namespace.com
    http:
      paths:
      - backend:
          serviceName: service2
          servicePort: 80
```

The second Ingress will succeed as the `spec.rules.host` matches the regex pattern specified in the `allowed-ingress-pattern` annotation on the `ingress-test` Namespace. However, the first Ingress does not match and results in an error:

```
Error from server ([denied by limit-namespace-ingress] Only ingresses with
host matching \w\.my-namespace\.com are allowed in namespace ingress-test):
error when creating "ingress.yaml": admission webhook "validation.gatekeeper.sh"
denied the request: [denied by limit-namespace-ingress] Only ingresses with host
matching \w\.my-namespace\.com are allowed in namespace ingress-test
```

Gatekeeper has a number of strengths:

- The extensible `ConstraintTemplate` model allows admins to define common policies and share/reuse them as libraries across the organization.
- While it does require Rego knowledge, there is no additional programming language experience required, lowering the barrier to entry for policy design and creation.
- The underlying technology (OPA) is fairly mature and well-supported in the community. Gatekeeper is a newer layer but has experienced strong early support.
- Consolidating all policy enforcement into one admission controller allows us to access a centralized audit log, which is often important in regulated environments.

The main weakness of Gatekeeper is that it is currently unable to *mutate* requests. And while it does support external data sources (through a variety of methods), they can be cumbersome to implement. These issues will inevitably be addressed in the future, but should you have strong requirements in those areas, it's likely you will need to implement one of the custom admission control solutions described in previous sections.

One last consideration when utilizing Gatekeeper (and any general-purpose admission controller) is that the scope of requests captured by these tools is likely to be very broad. This is required for them to be useful because rules covering many different objects can be written and the controller itself needs to contain a superset of permissions to be able to capture them. However, this has several ramifications:

- As mentioned previously, these tools are in the critical path. If there is a bug or other issue with the controller and/or your configuration, there is the potential for widespread outage.
- As a continuation of the previous point, because the controllers intercept requests *to the control plane*, it's possible that you as administrators may also be temporarily *locked out* from performing remediation steps. This is pertinent especially in the case of resources that are important and/or integral to the *operation* of the cluster (e.g., networking resources and so on).
- The broad scope necessarily requires that a broad RBAC policy is attached to the admission controller/policy server. If there is a vulnerability in this software, then the potential for destructive actions is significant.

You should avoid configuring admission webhooks to intercept resources targeting the `kube-system` Namespace. Objects in this Namespace are often integral to the operation of the cluster and accidental mutations or rejections of these objects could cause serious issues in the cluster.

Summary

In this chapter we covered the many ways that we can control which objects are admitted to a Kubernetes cluster. Much like many of the concerns that we cover in this book, each way has its own specific trade-offs and decisions to make with regards to your individual requirements. Admission control is an area where even more careful examination and deeper knowledge is essential, given its heavy application in the area of cluster and workload security.

Built-in controllers expose a solid set of functionality but are unlikely to be all that you need. We expect more and more of these actions to move to external (out-of-tree) controllers that leverage the mutating and validating webhook capabilities. In the near term you may find that building your own webhooks is required (either from scratch or using a framework) for more complex functionality. However, as we see broad admission policy tools like Gatekeeper become more mature, we think this is where a lot of value can be added.

CHAPTER 9
Observability

The ability to observe any software system is critical. If you cannot examine the condition of your running applications, you cannot effectively manage them. And that is what we are addressing with observability: the various mechanisms and systems we use to understand the condition of running software that we are responsible for. We should acknowledge that we're not adhering to the control theory definition of observability in this context. We chose to use this term simply because it has become popular and we want people to readily understand what we're getting at.

The components of observability can be broken into three categories:

Logging
: Aggregating and storing the logged event messages written by programs

Metrics
: Collecting time series data, making it available in dashboards, and alerting upon it

Tracing
: Capturing data for requests that traverse multiple distinct workloads in the cluster

In this chapter, we will cover how to implement effective observability in Kubernetes-based platforms so that you can safely manage a platform and the workloads it hosts in production. First, we will explore logging and examine the systems for aggregating logs and forwarding them to your company's logging backend. Next, we'll cover how to collect metrics, how to visualize that data, and how to alert upon it. Lastly, we'll cover tracing requests through distributed systems so as to better understand what's happening when applications are composed of distinct workloads. Let's jump into logging and cover the commonly successful models there.

Logging Mechanics

This section covers logging concerns in a Kubernetes-based platform. We're primarily dealing with the mechanisms for capturing, processing, and forwarding logs from your platform components and tenant workloads to a storage backend.

Once upon a time, the software we ran in production usually wrote logs to a file on disk. Aggregation of logs—if performed at all—was a simpler exercise because there were fewer distinct workloads and fewer instances of those workloads compared with today's systems. In a containerized world, our applications commonly log to standard out and standard error the way an interactive CLI would. Indeed, this became accepted as best practice for modern service-oriented software even before containers became prevalent. In cloud native software ecosystems, there are more distinct workloads and instances of each, but they're also ephemeral and often without a disk mounted to persist the logs—hence the shift away from writing logs to disk. This introduced challenges in the collection, aggregation, and storage of logs.

Often a single workload will have multiple replicas, and there may be multiple distinct components to examine. Without centralized log aggregation, analyzing (viewing and parsing) logs in this scenario becomes very tedious, if not practically impossible. Consider having to analyze logs for a workload that has *dozens* of replicas. In these cases, it is essential to have a central collection point that allows you to search the log entries across replicas.

In covering logging mechanics, we'll first look at strategies for capturing and routing the logs from the containerized workloads in your platform. This includes the logs for the Kubernetes control plane and platform utilities as well as the platform tenants. We'll also cover the Kubernetes API Server audit logs as well as Kubernetes Events in this section. Lastly, we'll address the notion of alerting upon conditions found in logged data and alternative strategies for that. We will not cover the storage of logs because most enterprises have a log backend that we will integrate with—it is generally not a concern of the Kubernetes-based platform itself.

Container Log Processing

Let's look at three ways log processing could be done for the containerized workloads in a Kubernetes-based platform:

Application forwarding
: Send logs to the backend directly from the application.

Sidecar processing
: Use a sidecar to manage the logs for an application.

Node agent forwarding
: Run a Pod on each Node that forwards logs for all containers on that Node to the backend.

Application forwarding

In this case, the application needs to be integrated with the backend storage of logs. The developers have to build this functionality into their application and maintain that functionality. If the log backend changes, an update to the application will likely be required. Since log processing is virtually universal, it makes much more sense to offload this from the application. Application forwarding is not a good option in most situations and is rarely seen in production environments. It makes sense only if you have a heritage application that is being migrated to a Kubernetes-based platform that already integrates with a log backend.

Sidecar processing

In this model, the application runs in one container and writes logs to one or more files in the shared storage for the Pod. Another container in the same Pod, a sidecar, reads those logs and processes them. The sidecar does one of two things with the logs:

1. Forwards them directly to the log storage backend
2. Writes the logs to standard error and standard out

Forwarding directly to the backend is the primary use case for sidecar log processing. This approach is uncommon and is usually a makeshift workaround where the platform offers no log aggregation system.

In situations where the sidecar writes the logs to standard out and standard error, it does so to leverage Node agent forwarding (which is covered in the next section). This is also an uncommon method and is only useful if you are running an application that just isn't able to write logs to standard out and standard error.

Node agent forwarding

With node agent forwarding, a log processing workload runs on each node in the cluster, reads the logfiles for each container written by the container runtime, and forwards the logs to the backend storage.

This is the model we generally recommend and is, by far, the most common implementation. It is a useful pattern because:

- There is a single point of integration between the log forwarder and the backend, as opposed to different sidecars and/or applications having to maintain that integration.

- The configuration of standardized filtering, attaching metadata, and forwarding to multiple backends is centralized.
- Log rotation is taken care of by the kubelet or container runtime. If the application is writing logfiles inside the container, the application itself or the sidecar (if there is one) needs to handle log rotation.

The prevailing tools used for this node agent log forwarding are Fluentd and Fluent Bit. As the names suggest, they are related projects. Fluentd was the original, is written primarily in Ruby, and has a rich ecosystem of plug-ins around it. Fluent Bit came from a demand for a more lightweight solution for environments like embedded Linux. It is written in C and has a much smaller memory footprint than Fluentd, but it does not have as many plug-ins available.

The general guidance we give platform engineers when choosing a log aggregation and forwarding tool is to use Fluent Bit unless there are plug-ins for Fluentd that have compelling features. If you find a need to leverage Fluentd plug-ins, consider running it as a cluster-wide aggregator in conjunction with Fluent Bit as the node agent. In this model, you use Fluent Bit as the node agent, which is deployed as a DaemonSet. Fluent Bit forwards logs to Fluentd running in the cluster as a Deployment or StatefulSet. Fluentd performs further tagging and routes the logs to one or more backends where developers access them. Figure 9-1 illustrates this pattern.

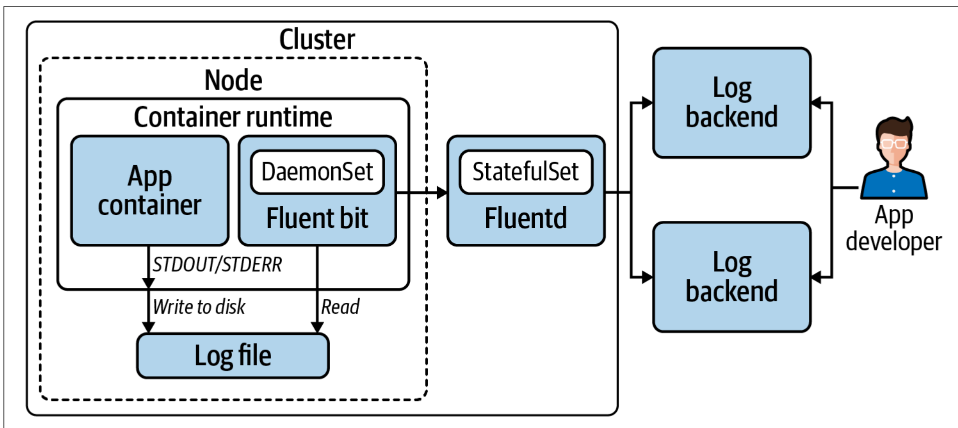


Figure 9-1. Aggregation of logs from containerized apps to backend.

While we strongly prefer the node agent forwarding method, it's worth calling out the potential problems with centralizing log aggregation. You are introducing a central point of failure for each node, or for the entire cluster if you use a cluster-wide aggregator in your stack. If your node agent gets bogged down by one workload that is logging excessively, that could affect the collection of logs for all workloads on that node. If you have a Fluentd cluster-wide aggregator running as a Deployment, it will be

using the ephemeral storage layer in its Pod as a buffer. If it gets killed before it can flush the logs from its buffer, you will lose logs. For this reason, consider running it as a StatefulSet so those logs aren't lost if the Pod goes down.

Kubernetes Audit Logs

This section covers collecting audit logs from the Kubernetes API. These logs offer a way to find out who did what in the cluster. You will want to have these turned on in production so that you can perform a root cause analysis in the event that something goes wrong. You may also have compliance requirements that necessitate them.

Audit logs are enabled and configured with flags on the API server. The API server allows you to capture a log of every stage of every request sent to it, including the request and response bodies. In reality, it's unlikely you will want *every* request logged. There are a lot of calls to the API server so there will be a very large number of log entries to store. You can use rules in an audit policy to qualify which requests and stages you wish your API server to write logs for. Without any audit policy, the API server won't actually write any logs. Tell the API server where your audit policy is on the control plane node's filesystem with the `--audit-policy-file` flag. Example 9-1 shows several rules that illustrate how the policy rules work so that you can limit the volume of log information without excluding important data.

Example 9-1. Example audit policy

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: None ❶
  users: ["system:kube-proxy"]
  verbs: ["watch"]
  resources:
  - group: "" # core
    resources: ["endpoints", "services", "services/status"]
- level: Metadata ❷
  resources:
  - group: ""
    resources: ["secrets", "configmaps"]
  - group: authentication.k8s.io
    resources: ["tokenreviews"]
  omitStages:
  - "RequestReceived"
- level: Request ❸
  verbs: ["get", "list", "watch"]
  resources:
  - group: ""
  - group: "apps"
  - group: "batch"
  omitStages:
```

```yaml
  - "RequestReceived"
- level: RequestResponse ❹
  resources:
  - group: ""
  - group: "apps"
  - group: "batch"
  omitStages:
  - "RequestReceived"
# Default level for all other requests.
- level: Metadata ❺
  omitStages:
  - "RequestReceived"
```

❶ The `None` auditing level means the API server will not log events that match this rule. So when the user `system:kube-proxy` requests a watch on the listed resources, the event is not logged.

❷ The `Metadata` level means that only request metadata is logged. When any request for the listed resources is received by the API server, it will log which user made what type of requests for what resource, but not the body of the request or response. The `RequestReceived` stage will not be logged. This means it will not write a separate log entry when the request is received. It will write a log entry when it starts the response for a long-running watch. It will write a log entry after it completes the response to the client. And it will log any panic that occurs. But will omit a log entry when the request is first received.

❸ The `Request` level will instruct the API server to log the request metadata and request body, but *not* the response body. So when any client sends a get, list, or watch request, the potentially verbose response body—that contains the object/s—is not logged.

❹ The `RequestResponse` level logs the most information: the request metadata, the request body, and the response body. This rule lists the same API groups as the previous. So, in effect, this rule says that if a request is *not* a get, list, or watch for a resource in one of these groups, additionally log the response body. In effect this becomes the default log level for the listed groups.

❺ Any other resources that aren't matched in previous rules will have this default applied, which says to skip the additional log message when a request is received and log only request metadata and excluded request and response bodies.

As with other logs in your system, you will want to forward the audit logs to some backend. You can use either the application forwarding or node agent forwarding strategies we covered earlier in this chapter. Much of the same principles and patterns apply.

For the application forwarding approach, you can configure the API server to send logs directly to a webhook backend. In this case you tell the API server with a flag where the config file is that contains the address and credentials to connect. This config file uses the kubeconfig format. You will need to spend some time tuning the configuration options for buffering and batching to ensure all logs arrive at the backend. For example, if you set a buffer size for the number of events to buffer before batching that is too low and it overflows, events will be dropped.

For node agent forwarding, you can have the API server write logfiles to the filesystem on the control plane node. You can provide flags to the API server to configure the filepath, maximum retention period, maximum number of files, and maximum logfile size. In this case you can aggregate and forward logs with tools like Fluent Bit and Fluentd. This is likely a good pattern to follow if you are already using these tools to manage logs with the node agent forwarding discussed earlier.

Kubernetes Events

In Kubernetes, Events are a native resource. They are a way for platform components to expose information about what has happened to different objects through the Kubernetes API. In effect, they are a kind of platform log. Unlike other logs, they are not generally stored in a logging backend. They are stored in etcd and, by default, are retained for one hour. They are most commonly used by platform operators and users when they want to gather information about actions taken against objects. Example 9-2 shows the Events provided when describing a newly created Pod.

Example 9-2. Events given with Pod description

```
$ kubectl describe pod nginx-6db489d4b7-q8ppw
Name:         nginx-6db489d4b7-q8ppw
Namespace:    default
...
Events:
  Type    Reason     Age        From
  ----    ------     ----       ----
    Message
    -------
  Normal  Scheduled  <unknown>  default-scheduler
    Successfully assigned default/nginx-6db489d4b7-q8ppw
  Normal  Pulling    34s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Pulling image "nginx"
  Normal  Pulled     30s        kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Successfully pulled image "nginx"
```

```
  Normal  Created    30s       kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Created container nginx
  Normal  Started    30s       kubelet, ip-10-0-0-229.us-east-2.compute.internal
    Started container nginx
```

You can also retrieve the same Events directly as shown in Example 9-3. In this case it includes the Events for the ReplicaSet and Deployment resources in addition to the Pod Events we saw when describing that resource.

Example 9-3. Events for a namespace retrieved directly

```
$ kubectl get events -n default
LAST SEEN   TYPE     REASON              OBJECT
    MESSAGE
2m5s        Normal   Scheduled           pod/nginx-6db489d4b7-q8ppw
    Successfully assigned default/nginx-6db489d4b7-q8ppw
2m5s        Normal   Pulling             pod/nginx-6db489d4b7-q8ppw
    Pulling image "nginx"
2m1s        Normal   Pulled              pod/nginx-6db489d4b7-q8ppw
    Successfully pulled image "nginx"
2m1s        Normal   Created             pod/nginx-6db489d4b7-q8ppw
    Created container nginx
2m1s        Normal   Started             pod/nginx-6db489d4b7-q8ppw
    Started container nginx
2m6s        Normal   SuccessfulCreate    replicaset/nginx-6db489d4b7
    Created pod: nginx-6db489d4b7-q8ppw
2m6s        Normal   ScalingReplicaSet   deployment/nginx
    Scaled up replica set nginx-6db489d4b7 to 1
```

Being that Kubernetes Events are available through the Kubernetes API, it is entirely possible to build automation to watch for, and react to, specific Events. However, in reality, we don't see this commonly done. One additional way you can leverage them is through an Event exporter that exposes them as metrics. See "Prometheus" on page 251 for more on Prometheus exporters.

Alerting on Logs

Application logs expose important information about the behavior of your software. They are especially valuable when unexpected failures occur that require investigation. This may lead you to find patterns of events that precipitate problems. If you find yourself wanting to set up alerts on Events exposed in logs, first consider using metrics instead. If you expose metrics that represent that behavior, you can implement alerting rules against them. Log messages are less reliable to alert on as they are more subject to change. A slight change to the text of a log message may inadvertently break the alerting that uses it.

Security Implications

Don't forget to give some thought to the access users have to the various logs aggregated in your backend. You may not want your production API server audit logs accessible to everyone. You may have sensitive systems with information that only privileged users should have access to. This may impact the tagging of logs or may necessitate using multiple backends, impacting your forwarding configurations.

Now that we've covered the various mechanics involved in managing the logs from your platform and its tenants, let's move on to metrics and alerting.

Metrics

Metrics and alerting services are vital to the usability of the platform. Metrics allow us to plot measured data on a timeline and recognize divergences that indicate undesirable or unexpected behavior. They help us understand what's happening with our applications, inform us whether they are behaving as we expect, and give us insights into how we can remedy problems or improve how we manage our workloads. And, critically, metrics give us useful measurements to alert on. Notifications of failures, or preferably warnings of impending failures, give us the opportunity to avert and/or minimize downtime and errors.

In this section, we're going to cover how to provide metrics and alerting as a platform service using Prometheus. There is considerable detail to explore here, and it will be helpful to reference a particular software stack as we do so. This is not to say you cannot or should not use other solutions. There are many circumstances where Prometheus may *not* be the right solution. However, Prometheus does provide an excellent model for addressing the subject. Regardless of the exact tools you use, the Prometheus model provides a clear implementation reference that will inform how you approach this topic.

First, we will take a general look at what Prometheus is, how it collects metrics, and what functions it provides. Then we will address various general subtopics, including long-term storage and the use case for pushing metrics. Next, we'll cover custom metrics generation and collection as well as organization and federation of metrics collection across your infrastructure. Also, we will dive into alerting and using metrics for actionable showback and chargeback data. Finally, we will break down each of the various components in the Prometheus stack and illustrate how they fit together.

Prometheus

Prometheus is an open source metrics tool that has become the prevalent open source solution for Kubernetes-based platforms. The control plane components expose Prometheus metrics, and virtually every production cluster uses Prometheus exporters to get metrics from things like the underlying nodes. Due to this, many enterprise

systems such as Datadog, New Relic, and VMware Tanzu Observability support consuming Prometheus metrics.

Prometheus metrics are simply a standard format for time series data that can actually be used by any system. Prometheus uses a scraping model whereby it collects metrics from targets. So applications and infrastructure do not typically *send* metrics anywhere, they expose them at an endpoint from which Prometheus can scrape them. This model removes the app's responsibility for knowing anything about the metrics system beyond the format in which to present the data.

This scraping model for collecting metrics, its ability to process large amounts of data, the use of labels in its data model, and the Prometheus Query Language (PromQL) make it a great metrics tool for dynamic, cloud native environments. New workloads can be readily introduced and monitored. Expose Prometheus metrics from an application or system, add a scrape configuration to a Prometheus server, and use PromQL to turn the raw data into meaningful insights and alerts. These are some of the core reasons Prometheus became such a popular choice in the Kubernetes ecosystem.

Prometheus provides several critical metrics functions:

- Collects metrics from targets using its scraping model
- Stores the metrics in a time series database
- Sends alerts, usually to Alertmanager, which is discussed later in this chapter, based on alerting rules
- Exposes an HTTP API for other components to access the metrics stored by Prometheus
- Provides a dashboard that is useful for executing ad hoc metric queries and getting various status info

Most teams use Prometheus for metrics collection paired with Grafana for visualization when they start out. However, maintaining organized use of the system in a production environment can become challenging for smaller teams. You will have to solve for long-term storage of your metrics, scaling Prometheus as the volume of metrics grows, as well as organizing the federation of your metrics systems. None of these are trivial problems to solve and manage over time. So if the maintenance of the metrics stack becomes cumbersome as the system scales, you can migrate to one of those commercial systems without changing the type of metrics you use.

Long-Term Storage

It's important to note that Prometheus is not designed for long-term storage of metrics. Instead, it provides support for writing to remote endpoints, and there are a number of solutions that can be used for this kind of integration. The questions you have to answer when providing a metrics solution as a part of your application platform are around data retention. Do you offer long-term storage only in production? If so, what retention period at the Prometheus layer do you offer in nonprod environments? How will you expose the metrics in long-term storage to users? Projects such as Thanos and Cortex offer tool stacks to help solve these problems. Just keep in mind how your platform tenants will be able to leverage these systems and let them know what retention policies they can expect.

Pushing Metrics

Not every workload is a good fit for the scraping model. In these cases the Prometheus Pushgateway may be used. For example, a batch workload that shuts down when it has finished its work may not give the Prometheus server the chance to collect all metrics before it disappears. For this situation, the batch workload can push its metrics to the Pushgateway which, in turn, exposes those metrics for the Prometheus server to retrieve. So if your platform will support workloads that call for this support, you will likely need to deploy the Pushgateway as a part of your metrics stack and publish information for tenants to leverage it. They will need to know where it is in the cluster and how to use its REST-like HTTP API. Figure 9-2 illustrates an ephemeral workload leveraging a Prometheus client library that supports pushing metrics to a Pushgateway. Those metrics are then scraped by a Prometheus server.

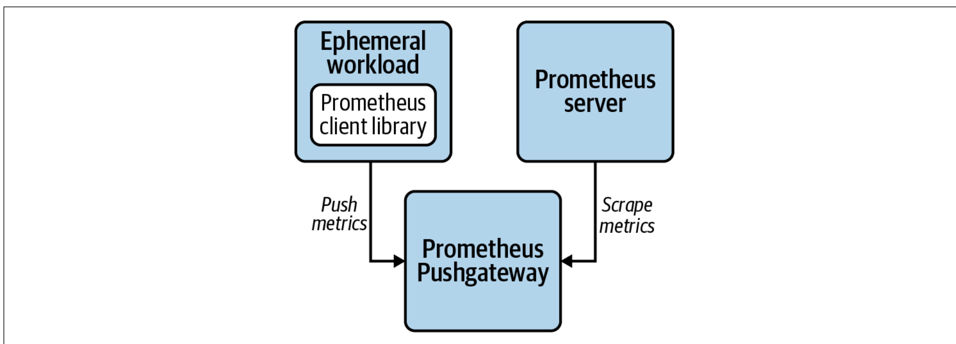


Figure 9-2. Pushgateway for ephemeral workload.

Custom Metrics

Prometheus metrics can be exposed natively by an application. Many applications that are developed expressly to run on Kubernetes-based platforms do just this. There

are several officially supported client libraries as well as a number of community-supported libraries. Using these, your application developers will likely find it trivial to expose custom Prometheus metrics for scraping. This is covered in depth in Chapter 14.

Alternatively, *exporters* can be used when Prometheus metrics are not natively supported by an app or system. Exporters collect data points from an application or system, then expose them as Prometheus metrics. A common example of this is the Node Exporter. It collects hardware and operating system metrics, then exposes those metrics for a Prometheus server to scrape. There are community-supported exporters for a wide range of popular tools, some of which you may find useful.

Once an application that exposes custom metrics is deployed, the next matter is adding the application to the scrape configuration of a Prometheus server. This is usually done with a ServiceMonitor custom resource used by the Prometheus Operator. The Prometheus Operator is covered further in "Metrics Components" on page 260, but for now it is enough to know that you can use a custom Kubernetes resource to instruct the operator to auto-discover Services based on their Namespace and labels.

In short, instrument the software you develop in-house where possible. Develop or leverage exporters where native instrumentation is not feasible. And collect the exposed metrics using convenient auto-discovery mechanisms to provide visibility into your systems.

While the use of labels in the Prometheus data model is powerful, with power comes responsibility. You can shoot yourself in the foot with them. If you overuse labels, the resource consumption of your Prometheus servers can become untenable. Familiarize yourself with the impact of high cardinality of metrics and check out the instrumentation guide in the Prometheus docs.

Organization and Federation

Processing metrics can be particularly compute-intensive, so subdividing this computational load can help manage resource consumption for Prometheus servers. For example, use one Prometheus server to collect metrics for the platform and use other Prometheus servers to collect custom metrics from applications or node metrics. This is particularly applicable in larger clusters where there are far more scrape targets and much larger volumes of metrics to process.

However, doing this will fragment the locations in which you can see this data. One way to solve this is through federation. Federation, in general, refers to consolidating data and control into a centralized system. Prometheus federation involves collecting important metrics from various Prometheus servers into a central Prometheus server.

This is accomplished using the same scraping model used to collect metrics from workloads. One of the targets that a Prometheus server can scrape metrics from is another Prometheus server.

This can be done within a single Kubernetes cluster, among several Kubernetes clusters, or both. This provides a very flexible model in that you can organize and consolidate your metrics systems in ways that suit the patterns you use to manage your Kubernetes clusters. This includes federating in layers or tiers. Figure 9-3 shows an example of a global Prometheus server scraping metrics from Prometheus servers in different datacenters which, in turn, scrape metrics from targets within their cluster.

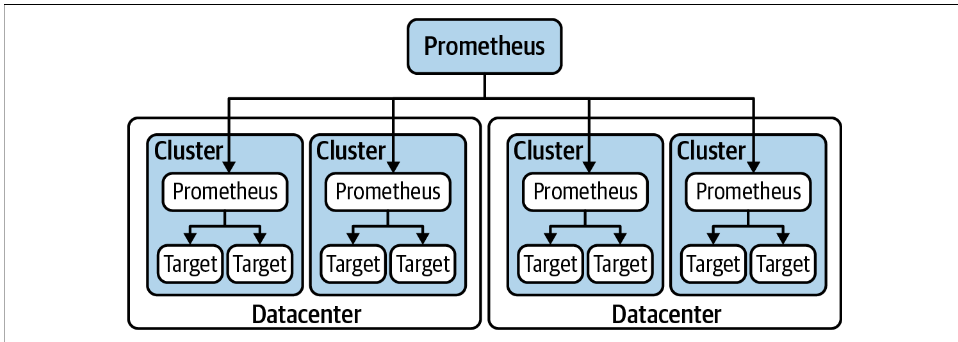


Figure 9-3. Prometheus federation.

While Prometheus federation is powerful and flexible, it can be complex and burdensome to manage. A relatively recent development that offers a compelling way to collect metrics from all your Prometheus servers is Thanos, an open source project that builds federation-like capabilities on top of Prometheus. It is supported by the Prometheus Operator and can be layered onto existing Prometheus installations. Cortex is another promising project in this capacity. Both Thanos and Cortex are incubating projects in the CNCF.

Carefully plan out the organization and federation of your Prometheus servers to support scaling and expanding your operations as platform adoption grows. Give careful consideration to the consumption model for tenants. Avoid making them use a multitude of different dashboards to access metrics for their workloads.

Alerts

Prometheus uses alerting rules to generate alerts from metrics. When an alert is triggered, the alert will usually be sent to a configured Alertmanager instance. Deploying Alertmanager and configuring Prometheus to send alerts to it is somewhat trivial when using the Prometheus Operator. Alertmanager will process the alert and integrate with messaging systems to make your engineers aware of issues. Figure 9-4 illustrates the use of distinct Prometheus servers for the platform control plane and

tenant applications. They both use a common Alertmanager to process alerts and notify receivers.

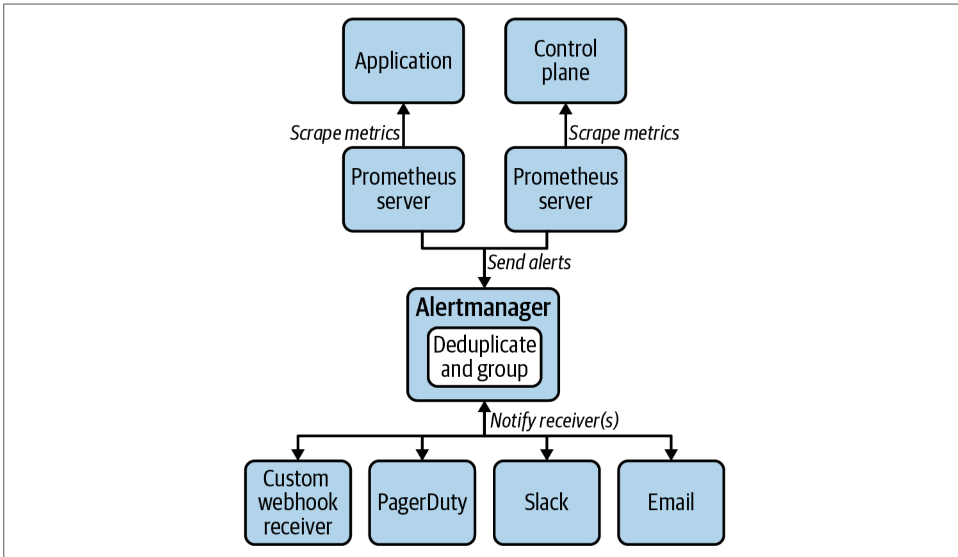


Figure 9-4. Alerting components.

In general, be careful not to *over* alert. Excessive critical alerts will burn out your on-call engineers and the noise of false positives can drown out actual critical events. So take the time to tune your alerts to be useful. Add useful descriptions to the annotations on the alerts so that when engineers are alerted to a problem, they have some useful context to understand the situation. Consider including links to runbooks or other docs that can aid the resolution of the alerted incident.

In addition to alerts for the platform, consider how to expose alerting for your tenants so that they may set up alerts on the metrics for their application. This involves giving them ways to add alerting rules to Prometheus, which is covered more in "Metrics Components" on page 260. It also includes setting up the notification mechanisms through Alertmanager so that application teams are alerted according to the rules they set.

Dead man's switch

One alert in particular is worth addressing as it is universally applicable and particularly critical. What happens if your metrics and alerting systems go down? How will you get an alert of *that* event? In this case you need to set up an alert that fires periodically under normal operating conditions and that, if those alerts stop, fires a critical alert to let on-call know your metrics and/or alerting systems are down. PagerDuty has an integration they call Dead Man's *Snitch* that provides this feature. Alternatively,

you could set up a custom solution with webhook alerts to a system you install. Regardless of the implementation details, ensure you are notified urgently if your alerting system goes offline.

Showback and Chargeback

Showback is a term commonly used to describe resource usage by an organizational unit or its workloads. *Chargeback* is the association of costs with that resource usage. These are perfect examples of meaningful, actionable expressions of metrics data.

Kubernetes offers the opportunity to dynamically manage compute infrastructure used by app development teams. If this readily available capacity is not managed well, you may find cluster sprawl and poor utilization of resources. It is greatly advantageous to the business to streamline the process of deploying infrastructure and workloads for efficiency. However, this streamlining can also lead to waste. For this reason, many organizations make their teams and lines of business accountable for the usage with showback and chargeback.

In order to make it possible to collect relevant metrics, workloads need to be labeled with something useful like a "team" or "owner" name or identifier. We recommend establishing a standardized system for this in your organization and use admission control to enforce the use of such a label on all Pods deployed by platform tenants. There are occasionally other useful methods of identifying workloads, such as by Namespace, but labels are the most flexible.

There are two general approaches to implementing showback:

Requests
: Requests are based on the resources a team reserves with the resource requests that are defined for each container in a Pod.

Consumption
: Consumption is based on what a team actually consumes through resource requests *or* actual usage, whichever is higher.

Showback by requests

The requests-based method leverages the aggregate resource requests defined by a workload. For example, if a Deployment with 10 replicas requests 1 CPU core per replica, it is considered to have used 10 cores per unit of time that it was running. In this model, if a workload bursts above its requests and uses 1.5 cores per replica on average, it would have gotten those resources for free; those additional 5 cores consumed above its resource requests are *not* attributed to the workload. This approach is advantageous in that it is based on what the scheduler can assign to nodes in the cluster. The scheduler considers resource requests as reserved capacity on a node. If a node has spare resources that aren't being used, and a workload bursts to use that

otherwise unused capacity, that workload got those resources for free. The solid line in Figure 9-5 indicates the CPU resources attributed to a workload using this method. Consumption that bursts above requests is *not* attributed.

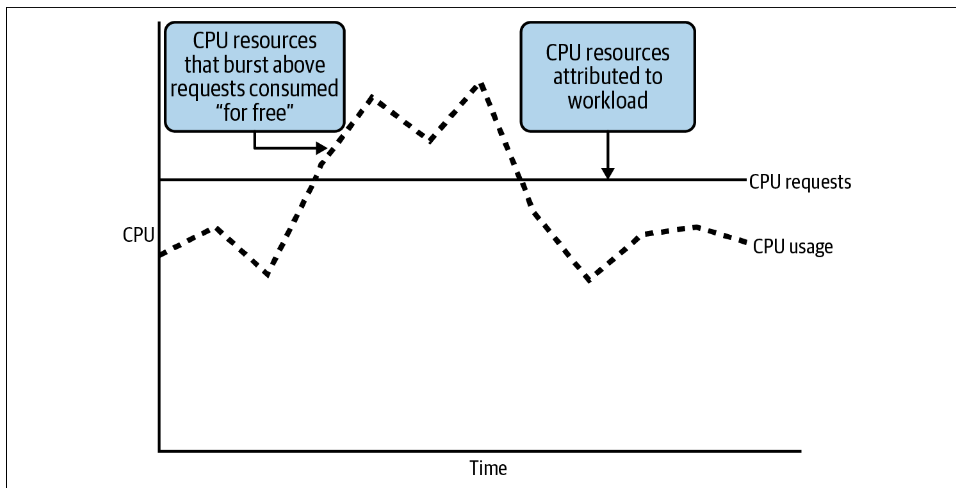


Figure 9-5. Showback based on CPU requests.

Showback by consumption

In a consumption-based model, a workload would be assigned the usage of its resource requests *or* its actual usage, whichever is higher. With this approach, if a workload commonly and consistently used more than its requested resources, it would be shown to have used those resources it actually consumed. This approach would remove the possible incentive to game the system by setting resource requests low. This could be more likely to lead to resource contention on overcommitted nodes. The solid line in Figure 9-6 indicates the CPU resources attributed to a workload using this consumption-based method. In this case, consumption that bursts above requests is attributed.

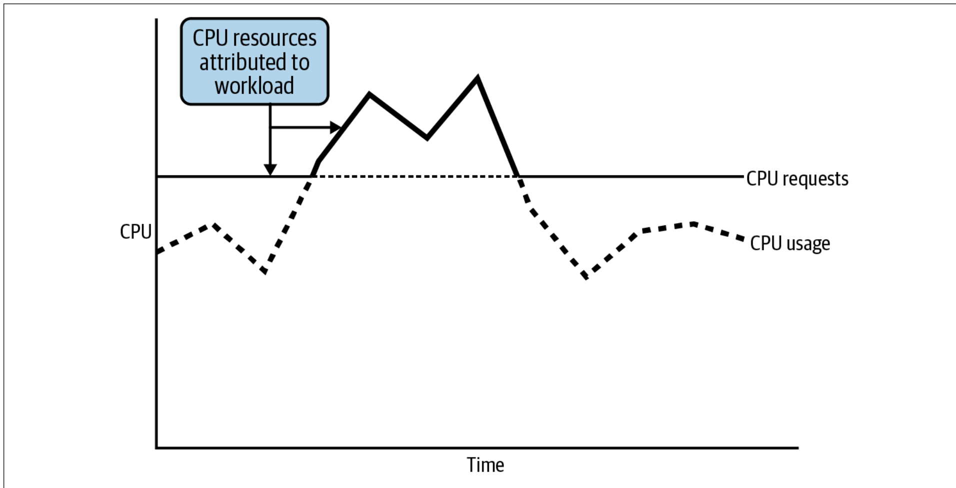


Figure 9-6. Showback based on CPU consumption when bursting above requests.

In "Metrics Components" on page 260, we will cover kube-state-metrics, a platform service that exposes metrics related to Kubernetes resources. If you use kube-state-metrics as a part of your metrics stack, you will have the following metrics available for resource requests:

- CPU: `kube_pod_container_resource_requests`
- Memory: `kube_pod_container_resource_requests_memory_bytes`

Resource usage can be obtained with the following metrics:

- CPU: `container_cpu_usage_seconds_total`
- Memory: `container_memory_usage_bytes`

Lastly for showback, you should decide whether to use CPU or memory for determining showback for a workload. For this, calculate the percentage of total cluster resource consumed by a workload for both CPU and memory. The higher value should apply because if a cluster runs out of CPU *or* memory it cannot host more workloads. For example, if a workload uses 1% of cluster CPU and 3% of cluster memory, it is effectively using 3% of the cluster since a cluster without any more memory cannot host any more workloads. This will also help inform whether you should employ different node profiles to match the workloads they host, which is discussed in "Infrastructure" on page 35.

Chargeback

Once we solve showback, chargeback becomes possible because we have metrics to apply costs to. Costs for machines will usually be pretty straightforward if using a public cloud provider. It may be a little more complicated if you are buying your own hardware, but somehow or another you need to come up with two cost values:

- Cost per unit of time for CPU
- Cost per unit of time for memory

Apply these costs to the showback value determined and you have a model for internally charging your platform tenants.

Network and storage

So far we've looked at showback and chargeback for the compute infrastructure used by workloads. This covers a majority of use cases we have seen in the field. However, there are workloads that consume considerable networking bandwidth and disk storage. This infrastructure can contribute significantly to the true cost of running some applications and should be considered in those cases. The model will be largely the same: collect the relevant metrics and then decide whether to charge according to resources reserved, consumed, or a combination of both. How you collect those metrics will depend on the systems used for this infrastructure.

At this point we have covered how Prometheus works and the general topics you should grasp before diving into the details of the deployed components. Next is a tour of those components that are commonly used in a Prometheus metrics stack.

Metrics Components

In this section we will examine the components in a very commonly used approach to deploying and managing a metrics stack. We'll also cover some of the management tools at your disposal and how all the pieces fit together. Figure 9-7 illustrates a common configuration of components in a Prometheus metrics stack. It does not include the Prometheus Operator, which is a utility for deployment and management of this stack, rather than a part of the stack itself. The diagram does include some autoscaling components to illustrate the role of Prometheus Adapter, even though autoscaling is not covered here. See Chapter 13 for details on that topic.

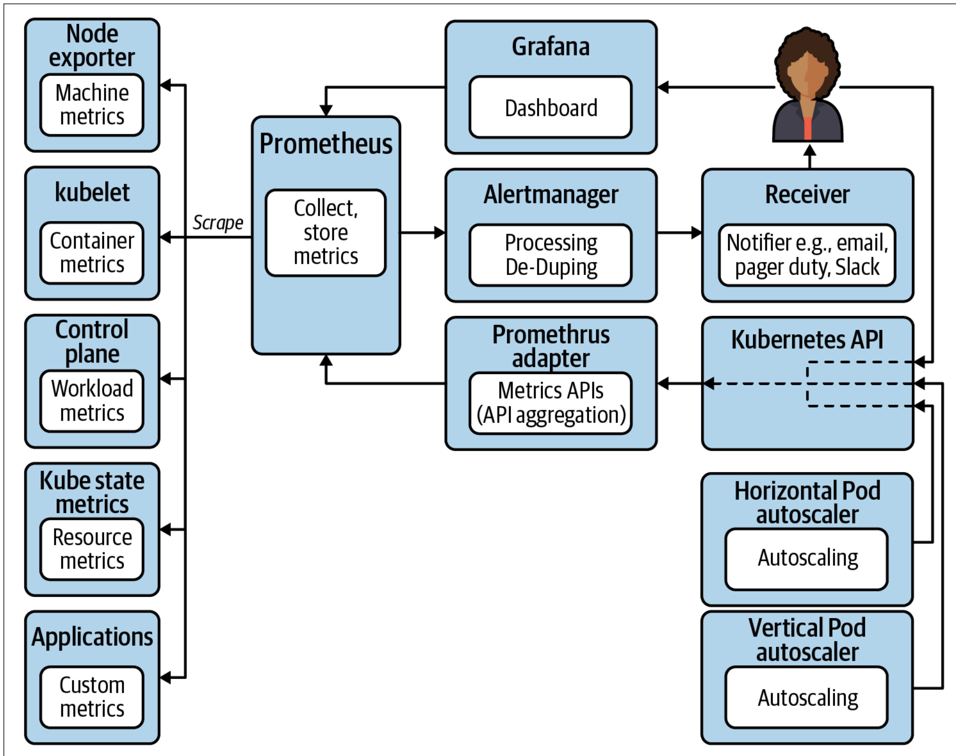


Figure 9-7. Common components in a Prometheus metrics stack.

Prometheus Operator

The Prometheus Operator is a Kubernetes operator that helps deploy and manage the various components of a Kubernetes metrics system for the platform itself as well as the tenant workloads. For more information about Kubernetes operators in general, see "The Operator Pattern" on page 317. The Prometheus Operator uses several custom resources that represent Prometheus servers: Alertmanager deployments, scrape configurations that inform Prometheus of the targets to scrape metrics from, and rules for recording metrics and alerting on them. This greatly reduces the toil in deploying and configuring Prometheus servers in your platform.

These custom resources are very useful to platform engineers but can also provide a very important interface for your platform tenants. If they require a dedicated Prometheus server, they can achieve that by submitting a Prometheus resource to their Namespace. If they need to add alerting rules to an existing Prometheus server, they can do so with a PrometheusRule resource.

The related kube-prometheus project is a great place to start with using the Prometheus Operator. It provides a collection of manifests for a complete metrics stack. It

includes Grafana dashboard configurations for useful visualization out of the box, which is very handy. But treat it as a place to start and understand the system so you can mold it to fit your requirements so that, once in production, you have confidence that you have comprehensive metrics and alerting for your systems.

The rest of this section covers the components you will get with a kube-prometheus deployment so you can clearly understand and customize these components for your needs.

Prometheus servers

With the Prometheus Operator in your clusters, you can create Prometheus custom resources that will prompt the operator to create a new StatefulSet for a Prometheus server. Example 9-4 is an example manifest for a Prometheus resource.

Example 9-4. Sample Prometheus manifest

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: platform
  namespace: platform-monitoring
  labels:
    monitor: platform
    owner: platform-engineering
spec:
  alerting:  ❶
    alertmanagers:
    - name: alertmanager-main
      namespace: platform-monitoring
      port: web
  image: quay.io/prometheus/prometheus:v2.20.0  ❷
  nodeSelector:
    kubernetes.io/os: linux
  replicas: 2
  resources:
    requests:
      memory: 400Mi
  ruleSelector:  ❸
    matchLabels:
      monitor: platform
      role: alert-rules
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
  serviceAccountName: platform-prometheus
  version: v2.20.0
  serviceMonitorSelector:  ❹
```

```
    matchLabels:
      monitor: platform
```

❶ Informs the configuration of Prometheus for where to send alerts.

❷ The container image to use for Prometheus.

❸ Informs the Prometheus Operator which PrometheusRules apply to this Prometheus server. Any PrometheusRule created with the labels shown here will be applied to this Prometheus server.

❹ This does the same for ServiceMonitors as the `ruleSelector` does for PrometheusRules. Any ServiceMonitor resources that have this label will be used to inform the scrape config for this Prometheus server.

The Prometheus custom resource allows platform operators to readily deploy Prometheus servers to collect metrics. As mentioned in "Organization and Federation" on page 254, it may be useful to divide metrics collection and processing load among multiple deployments of Prometheus within any given cluster. This model is enabled by the ability to spin up Prometheus servers using a custom Kubernetes resource.

In some use cases, the ability to spin up Prometheus servers with the Prometheus Operator is also helpful to expose to platform tenants. A team's applications may emit a large volume of metrics that will overwhelm existing Prometheus servers. And you may want to include a team's metrics collection and processing in its resource budget, so having a dedicated Prometheus server in their Namespace may be a useful model. Not every team will have an appetite for this approach where they deploy and manage their own Prometheus resources. Many may require further abstraction of the details, but it is an option to consider. If using this model, don't discount the added complexity this will introduce for dashboards and alerting for the metrics gathered, as well as for federation and long-term storage.

Deploying Prometheus servers is one thing, but ongoing management of configuration for them is another. For this, the Prometheus Operator has other custom resources, the most common being the ServiceMonitor. When you create a ServiceMonitor resource, the Prometheus Operator responds by updating the scrape configuration of the relevant Prometheus server. Example 9-5 shows a ServiceMonitor that will create a scrape configuration for Prometheus to collect metrics from the Kubernetes API server.

Example 9-5. Example manifest for a ServiceMonitor resource

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
```

```yaml
  labels:
    k8s-app: apiserver
    monitor: platform ➊
  name: kube-apiserver
  namespace: platform-monitoring
spec:
  endpoints: ➋
  - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    interval: 30s
    port: https
    scheme: https
    tlsConfig:
      caFile: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      serverName: kubernetes
  jobLabel: component ➌
  namespaceSelector: ➍
    matchNames:
    - default
  selector: ➎
    matchLabels:
      component: apiserver
      provider: kubernetes
```

➊ This is the label that is referred to in Example 9-1 of a `Prometheus` manifest by the `serviceMonitorSelector`.

➋ The `endpoints` provide configuration about the port to use and how to connect to the instances that Prometheus will scrape metrics from. This example instructs Prometheus to connect using HTTPS and provides a Certificate Authority and server name to verify the connection endpoint.

➌ In Prometheus terms, a "job" is a collection of instances of a service. For example, an individual apiserver is an "instance." All the apiservers in the cluster collectively comprise a "job." This field indicates which label contains the name that should be used for the job in Prometheus. In this case the job will be `apiserver`.

➍ The `namespaceSelector` instructs Prometheus in which Namespaces to look for Services to scrape metrics for this target.

➎ The `selector` enables service discovery by way of labels on a Kubernetes Service. In other words, any Service (in the default Namespace) that contains the specified labels will be used to find the targets to scrape metrics from.

Scrape configurations in a Prometheus server may also be managed with PodMonitor resources for monitoring groups of Pods (as opposed to Services with the ServiceMonitor), as well as Probe resources for monitoring Ingresses or static targets.

The PrometheusRule resource instructs the operator to generate a rule file for Prometheus that contains rules for recording metrics and alerting upon metrics. Example 9-6 shows an example of a PrometheusRule manifest that contains a recording rule and an alerting rule. These rules will be put in a ConfigMap and mounted into the Prometheus server's Pod/s.

Example 9-6. Example manifest for a PrometheusRule resource

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  labels:
    monitor: platform
    role: alert-rules ❶
  name: sample-rules
  namespace: platform-monitoring
spec:
  groups:
  - name: kube-apiserver.rules
    rules:
    - expr: |  ❷
        sum by (code,resource) (rate(
            apiserver_request_total{job="apiserver",verb=~"LIST|GET"}[5m]
        ))
      labels:
        verb: read
      record: code_resource:apiserver_request_total:rate5m
  - name: kubernetes-apps
    rules:
    - alert: KubePodNotReady  ❸
      annotations:
        description: Pod {{ $labels.namespace }}/{{ $labels.pod }} has been in a
          non-ready state for longer than 15 minutes.
        summary: Pod has been in a non-ready state for more than 15 minutes.
      expr: |
        sum by (namespace, pod) (
          max by(namespace, pod) (
            kube_pod_status_phase{job="kube-state-metrics", phase=~"Pending|Unknown"}
          ) * on(namespace, pod) group_left(owner_kind) topk by(namespace, pod) (
            1, max by(namespace, pod, owner_kind) (kube_pod_owner{owner_kind!="Job"})
          )
        ) > 0
      for: 15m
      labels:
        severity: warning
```

❶ This is the label that is referred to in Example 9-1 of a `Prometheus` manifest by the `ruleSelector`.

❷ This is an example of a recording rule for the total LIST and GET requests to all Kubernetes API server instances over a 5-minute period. It uses an expression on the `apiserver_request_total` metric exposed by the API server and stores a new metric called `code_resource:apiserver_request_total:rate5m`.

❸ Here is an alerting rule that will prompt Prometheus to send a warning alert if any Pod gets stuck in a not ready state for more than 15 minutes.

Using the Prometheus Operator and these custom resources to manage Prometheus servers and their configurations has proven to be a very useful pattern and has become very prevalent in the field. If you are using Prometheus as your primary metrics tool, we highly recommend it.

Alertmanager

The next major component is Alertmanager. This is a separate, distinct workload that processes alerts and routes them to receivers that constitute the communication medium to the on-call engineers. Prometheus has alerting rules that prompt Prometheus to fire off alerts in response to measurable conditions. Those alerts get sent to Alertmanager, where they are grouped and deduplicated so that humans don't receive a flood of alerts when outages occur that affect multiple replicas or components. Then notifications are sent via the configured receivers. Receivers are the supported notification systems, such as email, Slack, or PagerDuty. If you want to implement an unsupported or custom notification system, Alertmanager has a webhook receiver that allows you to provide a URL to which Alertmanager will send a POST request with a JSON payload.

When using the Prometheus Operator, a new Alertmanager can be deployed with a manifest, as shown in Example 9-7.

Example 9-7. Example manifest for an Alertmanager resource

```
apiVersion: monitoring.coreos.com/v1
kind: Alertmanager
metadata:
  labels:
    alertmanager: main
  name: main
  namespace: platform-monitoring
spec:
  image: quay.io/prometheus/alertmanager:v0.21.0
  nodeSelector:
    kubernetes.io/os: linux
  replicas: 2  ❶
  securityContext:
    fsGroup: 2000
```

```
    runAsNonRoot: true
    runAsUser: 1000
  serviceAccountName: alertmanager-main
  version: v0.21.0
```

❶ Multiple replicas may be requested to deploy Alertmanager in a highly available configuration.

While this custom resource gives you very convenient methods to deploy Alertmanager instances, it is very rarely necessary to deploy multiple Alertmanagers in a cluster, especially since it can be deployed in a highly available configuration. You could consider a centralized Alertmanager for multiple clusters, but having one per cluster is wise since it reduces external dependencies for any given cluster. Leveraging a common Alertmanager for a cluster provides the opportunity for tenants to leverage a single PrometheusRule resource to configure new alerting rules for their app. In this model, each Prometheus server is configured to send alerts to the cluster's Alertmanager.

Grafana

For platform operators to be able to reason about what is happening in a complex Kubernetes-based platform, it is crucial to build charts and dashboards from the data stored in Prometheus. Grafana is an open source visualization layer that has become the default solution for viewing Prometheus metrics. The kube-prometheus project provides a variety of dashboards to use as a basis and starting point, not to mention many others available in the community. And, of course, you have the freedom to build your own charts to display the time series data from any system you manage as a part of your platform.

Visualizing metrics is also critical for application teams. This is relevant to how you deploy your Prometheus servers. If you are leveraging multiple Prometheus instances in your cluster, how will you expose the metrics gathered to the tenants of the platform? On one hand, adding a Grafana dashboard to each Prometheus server may be a useful pattern. This could provide a convenient separation of concerns. However, on the other hand, if users find they have to routinely log in to multiple distinct dashboards, that may be cumbersome. In this case you have two options:

- Employ federation to collect metrics from different servers into a single server and then add a dashboard to that for a single place to access metrics for a set of systems. This is the approach used with projects such as Thanos.
- Add multiple data sources to a single Grafana dashboard. In this case a single dashboard exposes metrics from several Prometheus servers.

The choice boils down to whether you prefer the complexity to be in federating Prometheus instances or in managing more complex Grafana configurations. There

is the additional resource consumption to consider with the federation server option, but if that is palatable, it is mostly a matter of preference.

If you are using a single Prometheus server for a cluster and your platform operators and tenants are going to the same place to get metrics, you will need to consider permissions around viewing and editing dashboards. You will likely need to configure the organizations, teams, and users appropriately for your use case.

Node exporter

Node exporter is the node agent that usually runs as a Kubernetes DaemonSet and collects machine and operating system metrics. It gives you the host-level CPU, memory, disk I/O, disk space, network stats, and file descriptor information, to name just a few of the metrics it collects by default. As mentioned earlier, this is one of the most common examples of an exporter. Linux systems don't export Prometheus metrics natively. The node exporter knows how to collect those metrics from the kernel and then exposes them for scraping by Prometheus. It is useful any time you want to monitor the system and hardware of Unix-like machines with Prometheus.

kube-state-metrics

kube-state-metrics provides metrics related to a range of Kubernetes resources. It is essentially an exporter for information about resources collected from the Kubernetes API. For example, kube-state-metrics exposes Pod start times, status, labels, priority class, resource requests, and limits; all the information that you would normally use `kubectl get` or `kubectl describe` to collect. These metrics can be useful to detect critical cluster conditions, such as Pods stuck in crash loops or Namespaces nearing their resource quotas.

Prometheus adapter

Prometheus adapter is included here because it is a part of the kube-prometheus stack. However, it is not an exporter, nor is it involved in the core functionality of Prometheus. Instead, Prometheus adapter is a *client* of Prometheus. It retrieves metrics from the Prometheus API and makes them available through the Kubernetes metrics APIs. This enables autoscaling functionality for workloads. Refer to Chapter 13 for more information on autoscaling.

As you can see, there are many components to a production-grade metrics and alerting system. We've looked at how to accomplish this with Prometheus and the patterns demonstrated with the kube-prometheus stack including the Prometheus Operator to alleviate toil in managing these concerns. Now that we've covered logging and metrics, let's take a look at tracing.

Distributed Tracing

Tracing in general refers to a specialized kind of event capturing that follows an execution path. While tracing can be applicable to a single piece of software, in this section we're dealing with distributed tracing that spans multiple workloads and traces requests in microservice architectures. Organizations that have embraced distributed systems benefit greatly from this technology. In this section, we'll discuss how to make distributed tracing available as a platform service for your application teams.

An important distinction between distributed tracing compared to logging and metrics is that the tracing technology between the applications and platform must be compatible. As long as an app logs to stdout and stderr, the platform services to aggregate logs don't care how logs are written inside the app. And common metrics like CPU and memory consumption can be gathered from workloads without specialized instrumentation. However, if an application is instrumented with a client library that is incompatible with the tracing system offered by the platform, tracing will not work at all. For this reason, close collaboration between platform and application development teams is critical in this area.

In covering the topic of distributed tracing, we will first look at the OpenTracing and OpenTelemetry specifications along with some of the terminology that is used when discussing tracing. We'll then cover the components that are common with the popular projects used for tracing. After that, we'll touch on the application instrumentation necessary to enable tracing as well as the implications of using a service mesh.

OpenTracing and OpenTelemetry

OpenTracing is an open source specification for distributed tracing that helps the ecosystem converge on standards for implementations. The spec revolves around three concepts that are important in understanding tracing:

Trace
: When an end user of a distributed application makes a request, that request traverses distinct services that process the request and participate in satisfying the client request. The trace represents that entire transaction and is the entity that we are interested in analyzing. A trace consists of multiple spans.

Span
: Each distinct service that processes a request represents a span. The operations that occur within the workload boundaries constitute a single span that are part of a trace.

Tag
: Tags are metadata that are attached to spans to contextualize them within a trace and provide searchable indexes.

When traces are visualized, they will generally include each individual trace and readily indicate which components in a system are impacting performance the most. They will also help track down where errors are occurring and how they are affecting other components of an application.

Recently, the OpenTracing project merged with OpenCensus to form OpenTelemetry. At the time of this writing, OpenTelemetry support is still experimental in Jaeger, which is a fair barometer of adoption, but it is reasonable to expect OpenTelemetry to become the de facto standard.

Tracing Components

To offer distributed tracing as a platform service, there needs to be a few platform components in place. The patterns we'll discuss here are applicable to open source projects such as Zipkin and Jaeger, but the same models will often apply for other projects and commercially supported products that implement the OpenTracing standards.

Agent

Each component in a distributed application will output a span for each request processed. The agent acts as a server to which the application can send the span information. In a Kubernetes-based platform it will usually be a node agent that runs on each machine in the cluster and receives all spans for workloads on that node. The agent will forward batched spans to a central collector.

Collector

The collector processes the spans and stores them in the backend database. It is responsible for validating, indexing, and performing any transformations before persisting the spans to storage.

Storage

The supported databases vary from project to project, but the usual suspects are Cassandra and Elasticsearch. Even when sampling, distributed tracing systems collect very large amounts of data. The databases used need to be able to process and quickly search these large volumes of data to produce useful analysis.

API

As you might expect, the next component is an API that allows clients to access the stored data. It exposes the traces and their spans to other workloads or visualization layers.

User interface

This is where the rubber hits the road for your platform tenants. This visualization layer queries the API and reveals the data to app developers. It is where engineers can view the data collected in useful charts to analyze their systems and distributed applications.

Figure 9-8 illustrates these tracing components, their relationships to one another, and the common deployment methods.

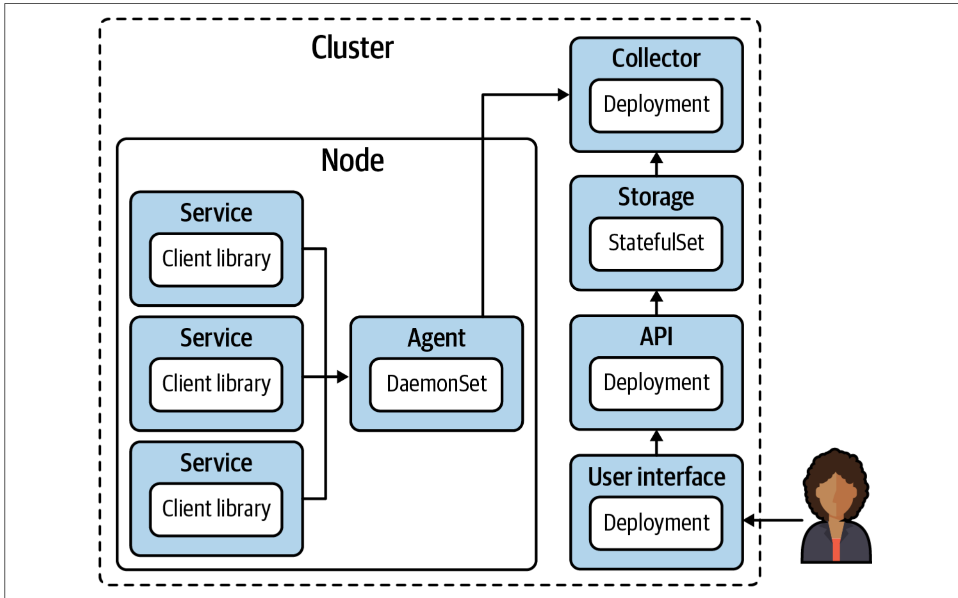


Figure 9-8. Components of a tracing platform service.

Application Instrumentation

In order for these spans to get collected and correlated together into traces, the application has to be instrumented to deliver this information. For this reason, it is crucial to get buy-in from your application development teams. The best tracing platform service in the world is useless if the applications are not delivering the raw data needed. Chapter 14 goes into this topic in more depth.

Service Meshes

If using a service mesh, you will likely want your mesh data included in your traces. Service meshes implement proxies for requests going to and from your workloads, and getting timing trace spans on those proxies helps you understand how they affect performance. Note that your application will still need to be instrumented, even if using a service mesh. The request headers need to be propagated from one service request to the next through a trace. Service meshes are covered in detail in Chapter 6.

Summary

Observability is a core concern in platform engineering. It's fair to say that no application platform could be considered production-ready without observability solved. As a baseline, make sure you are able to reliably collect logs from your containerized workloads and forward them to your logging backend along with the audit logs from the Kubernetes API server. Also consider metrics and alerting a minimum requirement. Collect the metrics exposed by the Kubernetes control plane, surface them in a dashboard, and alert on them. Work with your application development teams to instrument their applications to expose metrics where applicable and collect those, too. Finally, if your teams have embraced microservice architectures, work with your app dev teams to instrument their apps for tracing and install the platform components to leverage that information as well. With these systems in place you will have the visibility to troubleshoot and refine your operations to improve performance and stability.

CHAPTER 10

Identity

Establishing the identity of both users and application workloads is a key concern when designing and implementing a Kubernetes platform. No one wants to be in the news for having their systems breached. So it's vital that we ensure that only the appropriately privileged entities (human or application) can access particular systems or take certain actions. For this we need to ensure that there are both Authentication and Authorization systems implemented. As a refresher:

- *Authentication* is the process of establishing the identity of an application or user.
- *Authorization* is the process of determining what actions an application or user are able to do, after they have been authenticated.

This chapter is solely focused on authentication. That's not to say that authorization is not important, and we will touch on it briefly where appropriate. For more information you should definitely research Role Based Access Control (RBAC) in Kubernetes (there are many great resources available) and ensure that you have a solid strategy for implementing it for your own applications, so that you understand the permissions that are required by any external applications that you might deploy.

Establishing identity for the purposes of authentication is a key requirement of almost every distributed system. A simple example of this that everyone has used is a username and password. Together, the information identifies you as a user of the system. In this context then, *identity* needs to have a couple of properties:

- It needs to be verifiable. If a user enters their username and password, we need to be able to go to a database or source of truth and compare the values to make sure they're correct. In the case of a TLS certificate that might be presented, we need to be able to verify that certificate against a trusted issuing Certificate Authority (CA).

- It needs to be unique. If an identity provided to us is not unique, we cannot specifically identify the bearer. However, we need to maintain uniqueness only *within our desired scope*—for example, a username or email address.

Establishing identity is also a crucial precursor to handling authorization concerns. Before we can determine what scope of resource access should be granted, we need to uniquely identify the entity that is authenticating to the system.

Kubernetes clusters commonly serve multiple tenants where many users and teams are deploying and operating multiple applications in a single cluster. Solving for tenancy in Kubernetes presents challenges (many covered in this book), of which one is identity. Given the matrix of privileges and resources that must be considered, we must solve for many deployment and configuration scenarios. Development teams should have access to their applications. Operations teams should have access to all applications and might need access to platform services. Application-to-application communication should be limited among applications. Then the list goes on. What about shared services? Security teams? Deployment tooling?

These are all common concerns and add significant complexity to cluster configuration and maintenance. Remember, we have to keep these privileges updated somehow as well. These things are easy to get wrong. But the good news is that Kubernetes has capabilities that allow us to integrate with external systems and to model identity and access controls in a secure fashion.

In this chapter we'll begin by discussing *user* identity and the different methods for authenticating users to Kubernetes. We'll then move on to options and patterns for establishing *application* identity within a Kubernetes cluster. We'll see how to authenticate applications to the Kubernetes API server (for writing tools that interact directly with Kubernetes, such as operators). We'll also cover how to establish unique application identities to enable those applications to authenticate to each other within the cluster, in addition to authenticating to *external* services like AWS.

User Identity

In this section we'll cover the methods and patterns for implementing a robust system of *user* identity across your Kubernetes cluster(s). In this context we're defining a user as a human who will be interacting with the cluster directly (either through the Kubectl CLI or the API). The properties of identity (described in the previous section) are common to user and application identity, but some of the methods will differ. For example, we always want our identities to be verifiable and unique; however, these properties will be achieved in different ways for a user utilizing OpenID Connect (OIDC) versus an application using service account tokens.

Authentication Methods

There are a number of different authentication methods available to Kubernetes operators, and each have their own strengths and weaknesses. In keeping with the core theme of this book, it's essential to understand your specific use cases, evaluate what's going to work for you, integrate with your systems, provide the user experience (UX), and deliver the security posture that your organization requires.

In this section we'll cover each method of establishing *user* identity and their trade-offs while describing some commonly used patterns we've implemented in the field. Some of the methods described here are platform-specific and tied to functionality available in certain cloud vendors, while others are platform-agnostic. How well a system integrates into your existing technology landscape is definitely going to be a factor in determining whether to adopt it. The trade-off is between extra functionality available in new tooling versus the ease-of-maintenance of integrations with the incumbent stack.

In addition to providing identity, some of the methods described may also provide encryption. For example, the flow described for the public key infrastructure (PKI) method provides certificates that could be used in Mutual Transport Layer Security (mTLS) communication. However, encryption is not the focus of this chapter and is an incidental benefit from those methods of identity grants.

Shared secrets

A shared secret is a unique piece (or set) of information that is held by the calling entity and the server. For example, when an application needs to connect to a MySQL database, it can use a username and password combination to authenticate. This method necessitates that both parties have access to that combination in some form. You must create an entry in MySQL with that information, and then distribute the secret to any calling application that may need it. Figure 10-1 shows this pattern, with the backend application storing valid credentials that need to be presented by the frontend to gain access.

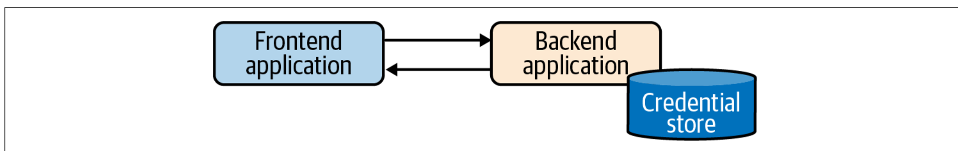
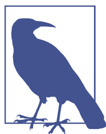


Figure 10-1. Shared secrets flow.

Kubernetes provides two options that allow you to utilize a shared secret model to authenticate to the API server. In the first method you can give the API server a list of comma-separated values (CSV) mapping usernames (and optionally, groups) to static tokens. When you want to authenticate to the API server, you can provide the token

as a Bearer token within the HTTP Authorization header. Kubernetes will treat the request as coming from the mapped user and act accordingly.

The other method is to supply the API server with a CSV of username (and optionally, groups) and password mappings. With this method configured, users can supply the credentials base64-encoded in the HTTP Basic Authorization header.



Kubernetes has no resource or object called User or Group. These are just predefined names for the purposes of identification within RBAC RoleBindings. The user can be mapped from a static file to token or password (as described previously), can be pulled from the CN of an x509 cert, or can be read as a field from an OAuth request, etc. The method of determining the user and group is entirely dependent on the authentication method in use, and Kubernetes has no way to define or manage them internally. In our opinion this pattern is a strength of the API because it allows us to plug in a variety of different implementations and delegate those concerns to systems specifically designed to handle them.

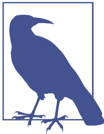
Both of these methods have serious weaknesses and are not recommended. Some of these weaknesses are due to the Kubernetes-specific implementation, and some are inherent to the shared secret model, which we'll discuss shortly. In Kubernetes, the main issues are:

- Static token and/or password files must be stored (in plain text) somewhere accessible to the API server. This is less of a risk than it initially seems, because if someone was able to compromise an API server and gain access to that node you would have greater things to worry about than an unencrypted password file. However, Kubenetes installations are mostly automated, and all assets required for setup should be stored in a repository. This repository must be secured, audited, and updated. This opens another potential area for carelessness or bad practices to leak credentials.
- Both the static tokens and the username/password combinations have no expiration date. If any credentials are compromised, the breach must be identified quickly and remediated by removing the relevant credentials and restarting the API server.
- Any modifications to these credential files require that the API server is restarted. In practice (and in isolation) this is fairly trivial. However, many organizations are rightly moving away from manual intervention into their running software and servers. Changing configurations is now mostly a rebuild and redeploy process versus simply SSH'ing into the machines (cattle over pets). Therefore,

modifying API server configurations and restarting the processes is likely a more involved action.

Outside of the Kubernetes-specific disadvantages just described, the shared secrets model suffers from another drawback. If I am an untrusted entity, how can I authenticate to a secret store *in the first place* to receive an appropriate identity? We'll look more at this *secure introduction* problem and how to solve it in "Application/Workload Identity" on page 288.

Public key infrastructure



This section assumes you are already familiar with PKI concepts.

The PKI model uses certificates and keys to uniquely identify and authenticate users to Kubernetes. Kubernetes makes extensive use of PKI to secure communications between all of the core components of the system. It's possible to configure the Certificate Authorities (CA) and certificates in multiple ways, but we will demonstrate it using kubeadm, the method most commonly seen in the field (and the de facto installation method for upstream Kubernetes).

After installing a Kubernetes cluster, you typically get a kubeconfig file with the `kubernetes-admin` user details. This file is essentially the root key to the cluster. Usually, this kubeconfig file is called *admin.conf* and is similar to this:

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <.. SNIP ...>
    server: https://127.0.0.1:32770
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
  user:
    client-certificate-data: <.. SNIP ...>
    client-key-data: <.. SNIP ...>
```

To determine the user that this will authenticate us to the cluster with, we need to first base64 decode the `client-certificate-data` field and then display the contents using something like `openssl`:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2587742639643938140 (0x23e98238661bcd5c)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Jul 25 19:48:42 2020 GMT
            Not After : Jul 25 19:48:44 2021 GMT
        Subject: O=system:masters, CN=kubernetes-admin
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <.. SNIP ...>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
    Signature Algorithm: sha256WithRSAEncryption
         <.. SNIP ...>
```

We see from the certificate that it was issued by the Kubernetes CA, and identifies the User as `kubernetes-admin` (the subject `CN` field) in the `system:masters` group. When using x509 certificates, any Organizations (O=) present are treated by Kubernetes as groups that the user should be considered part of. We will discuss some advanced methods around user and group configuration and permissions later in this chapter.

In the preceding example we saw the default configuration for the `kubernetes-admin` user, a reserved default name that enables cluster-wide administrative privileges. It would also be useful to see how to configure the provisioning of certificates to identify other regular system users who can then be given appropriate permissions using the RBAC system. Provisioning and maintaining a large set of certificate artifacts is an arduous task, but one that Kubernetes can help us with by using some built-in resources.

In order for the CSR flow described next to function correctly, the controller-manager needs to be configured with the `--cluster-signing-cert-file` and `--cluster-signing-key-file` parameters as shown here:

```yaml
spec:
  containers:
  - command:
    - kube-controller-manager
```

```
- --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
- --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
# Additional flags removed for brevity
image: k8s.gcr.io/kube-controller-manager:v1.17.3
```

Any entity with appropriate RBAC permissions can submit a Certificate Signing Request object to the Kubernetes API. If a user should be able to *self submit*, this means we need to provide a mechanism for the user to submit those requests. One way of doing this is to explicitly configure permissions to allow the `system:anonymous` User and / or `system:unauthenticated` group to submit and retrieve CSRs.

Without this, any unauthenticated user would by definition be unable to initiate the process that would allow them to become authenticated. We should definitely be wary of this approach, though, as we never want to give unauthenticated users any access to the Kubernetes API server. A common way of providing self-service for CSRs is therefore to provide a thin abstraction or portal on top of Kubernetes that will run with the appropriate permissions. Users can log in to the portal using some other credentials (usually SSO) and initiate this CSR flow (as shown in Figure 10-2).

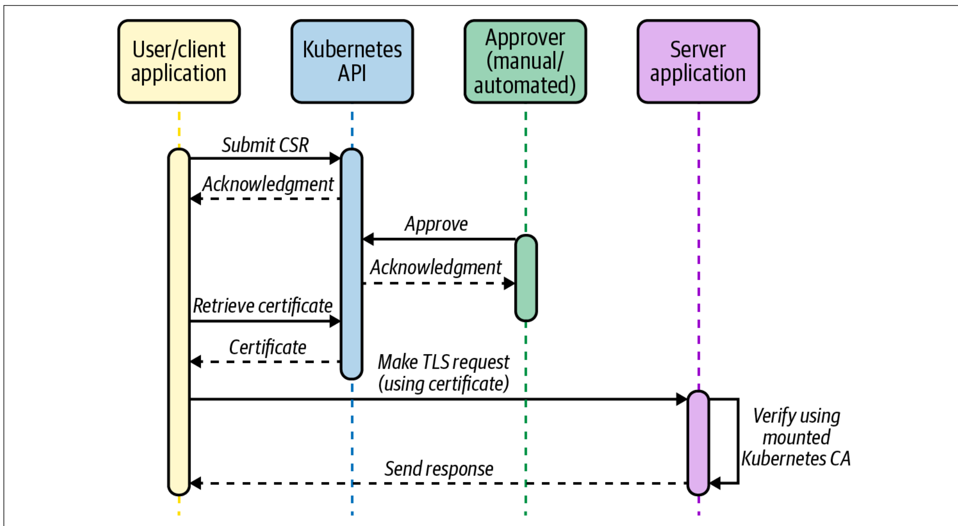


Figure 10-2. CSR flow.

In this flow the user could generate a private key locally and then submit this through the portal. Or, the portal could generate private keys for each user and return them with the approved certificate to the user. Generation can be done using `openssl` or any number of other tools/libraries. The CSR should contain the metadata the user wants encoded into their x509 certificate, including their user name and any additional groups they should be part of. The following example creates a certificate request that identifies the user as *john*:

```
$ openssl req -new -key john.key -out john.csr -subj "/CN=john"
$ openssl req -in john.csr -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: CN=john
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    <.. SNIP ...>
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
         <.. SNIP ...>
```

After generating the CSR we can submit it to the cluster via our portal in a CertificateSigningRequest resource. Following is an example of the request as a YAML object, but our portal would likely programatically apply this via the Kubernetes API rather than constructing the YAML manually:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: john
spec:
  request: $(cat john.csr | base64 | tr -d '\n')
  usages:
  - client auth
EOF
```

This creates a CSR object in Kubernetes with a `pending` state, awaiting approval. This CSR object contains the (base64-encoded) signing request and the username of the requestor. If using a Service Account token to authenticate to the Kubernetes API (as a Pod would in an automated flow), then the username will be the Service Account name. In the following example, I was authenticated to the Kubernetes API as the `kubernetes-admin` user and it appears in the Requestor field. If using a portal we'd see the Service Account assigned to that portal component.

```
$ kubectl get csr
NAME     AGE   REQUESTOR          CONDITION
my-app   17h   kubernetes-admin   Pending
```

While the request is pending, the user has not been granted any certificate. The next stage involves a cluster administrator (or a user with appropriate permissions) approving the CSR. This may also be automated if the user's identity can be programatically determined. Approval will issue a certificate back to the user that can be used to assert identity on that Kubernetes cluster. For this reason, it's important to

perform verification that the submitter of the request *is* who they claim to be. This could be achieved by adding some additional identifying metadata to the CSR and having an automated process validate the information against the claimed identity, or by having an out-of-band process to verify the user's identity.

Once the CSR has been approved, the certificate (in the `status` field of the CSR) can be retrieved and used (in conjunction with their private key) for TLS communications with the Kubernetes API. In our portal implementation, the CSR would be pulled by the portal system and made available for the requesting user once they log back in and recheck the portal:

```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-app
# Additional fields removed for brevity
status:
  certificate: <.. SNIP ...>
  conditions:
  - lastUpdateTime: "2020-03-04T15:45:30Z"
    message: This CSR was approved by kubectl certificate approve.
    reason: KubectlApprove
    type: Approved
```

When decoding the certificate we can see that it contains the relevant identity information (*john*) in the CN field:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            66:82:3f:cc:10:3f:aa:b1:df:5b:c5:42:cf:cb:5b:44:e1:45:49:7f
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Mar  4 15:41:00 2020 GMT
            Not After : Mar  4 15:41:00 2021 GMT
        Subject: CN=john
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    <.. SNIP ...>
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                EE:8E:E5:CC:98:41:78:4A:AE:32:75:52:1C:DC:DD:D0:9B:95:E0:81
```

```
        Signature Algorithm: sha256WithRSAEncryption
            <.. SNIP ...>
```

Finally, we can craft a kubeconfig containing our private key and the approved certificate that will allow us to communicate with the Kubernetes API server as the *john* user. The certificate we get back from the preceding CSR process goes into the `client-certificate-data` field shown here in the kubeconfig:

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: <.. SNIP ...>
    server: https://127.0.0.1:32770
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
  user:
    client-certificate-data: <.. SNIP ...>
    client-key-data: <.. SNIP ...>
```

We have seen implementations of this model in the field whereby an automated system provisions certificates based on some verifiable SSO credentials or other authentication method. When automated, these systems can be successful, but we do not recommend them. Relying on x509 certificates as a primary authentication method for users of Kubernetes introduces a number of issues:

- Certificates provisioned through the Kubernetes CSR flow cannot be revoked prior to their expiry. There is currently no support for certificate revocation lists of Online Certificate Status Protocol (OSCP) stapling in Kubernetes.
- Additional PKI needs to be provisioned, supported, and maintained, in addition to creating and maintaining a component responsible for provisioning certificates based on external authentication.
- x509 certificates have expiry timestamps, and these should be kept relatively short to reduce the risk should a pair (key/cert) be compromised. This short life span means that there is a high churn in certificates, and these must be distributed out to users regularly to ensure that consistent access to the cluster is maintained.
- There needs to be a way to verify the identity of anyone requesting a certificate. In an automated system, you can engineer ways of doing this via externally

verifiable metadata. In the absence of such metadata, out-of-band verification is often too time-consuming to be practical, especially given the short life span of certificates as noted previously.

- Certificates are are localized to one cluster. In the field we see many (10s–100s) Kubernetes clusters across projects and groups. Requiring unique credentials for each cluster multiplies the complexity of storing and maintaining the relevant credentials. This leads to a degraded user experience.

Remember that even when not using certificates as the primary authentication method, you should still keep the *admin.conf* kubeconfig somewhere secure. If for whatever reason other authentication methods become unavailable, this can act as an admin break-glass solution to access the cluster.

OpenID Connect (OIDC)

In our opinion, the best choice when setting up user authentication and identity with Kubernetes is to integrate with an existing Single Sign-On system or provider. Almost every organization already has a solution such as Okta, Auth0, Google, or even internal LDAP/AD that provides a single place for users to authenticate and gain access to internal systems. For something like authentication (where security is a strong factor), outsourcing the complexity is a solid choice unless you have very specialized requirements.

These systems have many advantages. They are built on well-understood and widely supported standards. They consolidate all management of user accounts and access to a single well-secured system, making maintenance and removal of accounts/access straightforward. Additionally, when using the common OIDC framework, they also allow users to access downstream applications without exposing credentials to those systems. Another advantage is that many Kubernetes clusters across multiple environments can leverage a single identity provider, reducing variance between cluster configurations.

Kubernetes supports OIDC directly as an authentication mechanism (as shown in Figure 10-3). If your organization is using an identity provider that natively exposes the relevant OIDC endpoints, then configuring Kubernetes to take advantage of this is straightforward.

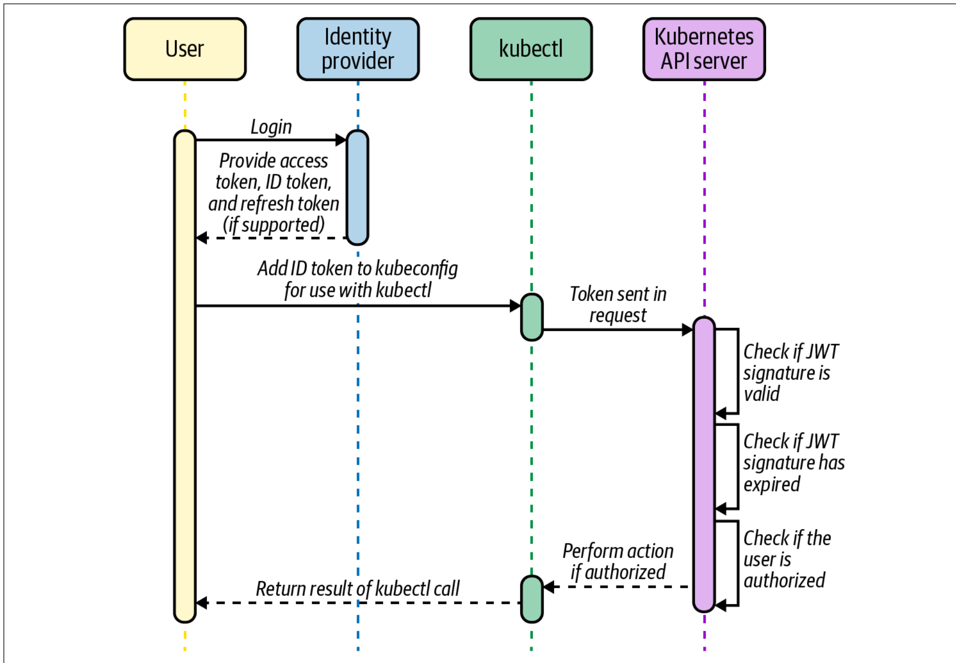


Figure 10-3. OIDC flow. Reproduced from the official Kubernetes documentation.

However, there are several scenarios where some extra tooling may be required or desired to provide additional functionality or improve user experience. Firstly, if your organization has multiple identity providers it is necessary to utilize an OIDC aggregator. Kubernetes only supports defining a single identity provider in its configuration options, and an OIDC aggregator is capable of acting as a single intermediary to multiple other providers (either OIDC or other methods). We have used Dex (a sandbox project within the Cloud Native Computing Foundation) with success many times before, although other popular options like Keycloak and UAA offer similiar functionality.

Remember that authentication is in the critical path to cluster access. Dex, Keycloak, and UAA are all configurable to variable degrees and you should optimize for availability and stability when implementing these solutions. These tools are additional maintenance burdens and must be configured, updated, and secured. In the field we always try to emphasize the need to understand and own any additional complexity that is introduced to your environment and clusters.

While configuring the API server to utilize OIDC is straightforward, attention must be given to providing a seamless user experience for the users of the cluster. OIDC

solutions will return a token identifying us (given a successful login); however, we require a properly formatted kubeconfig in order to access and perform operations on the cluster. When we hit this use case in the field early on, our colleagues developed a simple web UI called Gangway to automate the process of logging in through an OIDC provider and generating a conformant kubeconfig from the returned token (complete with relevant endpoints and certificates).

Despite OIDC being our preferred method of authentication, it does not suit all cases, and secondary methods may be required. OIDC (as defined in the specification) requires users to log in directly through the web interface of the identity provider. This is for obvious reasons, to ensure the user is actually providing the credentials only to the trusted provider and not to the consuming application. This requirement can cause issues in the case where robot users require access to the system. This is common for automated tooling like CI/CD systems and others who are unable to respond to the web-based credential challenge.

In these cases, we have seen a couple of different models/solutions:

- In cases where robot users are tied to centrally managed accounts, it's possible to implement a kubectl authentication plug-in that would log in to the external system and receive a token in response. Kubernetes can be configured to verify this token via the webhook token authenticator method. This method will likely require some custom coding to create the token generator/webhook server.
- For other cases, we have seen users fall back to using a certificate-based auth for robot accounts that don't need to be centrally managed. This of course means you need to manage certificate issuance and rotation, but it doesn't require any custom components.
- Another manual but effective alternative solution is to create a Service Account for the tool and utilize the token generated for API access. If the tool is running *in* cluster it can use the credential directly mounted into the Pod. If the tool is *outside* the cluster we can manually copy and paste the token into a secure location accessible by the tool and utilize that when making kubectl or API calls. Service Accounts are covered in more detail in "Service Account Tokens (SAT)" on page 293.

Implementing Least Privilege Permissions for Users

Now that we've seen the different ways it's possible to implement identity and authentication, let's turn to the related topic of authorization. It's out of scope for this book to go deep into how you should be configuring RBAC across your clusters. This will likely vary significantly between applications, environments, and teams. However, we do want to describe a pattern we've implemented successfully in the field around the principle of least privilege when designing administrative access roles.

Whether you have chosen to go with a cluster-per-team approach, or a multitenant cluster approach, you will likely have *super-admin* users on the operations team who are responsible for configuring, upgrading, and maintaining the environment. While individual teams should have restricted permissions based on the access they require, these admins will have full reign over the entire cluster and therefore greater potential to accidentally perform destructive actions.

In an ideal world, all cluster access and operations would be performed by an automated process—GitOps or something similar, perhaps. However, practically speaking, we regularly see users individually accessing clusters and found the following pattern to be an effective way to limit potential issues. It is tempting to bind an administrator role to a particular operator's username/identity directly, only for them to delete something important while mistakenly having loaded the wrong kubeconfig, for example. It should never happen until it does!

Kubernetes supports the concept of *impersonation*, and with this we can create an experience that behaves closely to *sudo* on Linux systems by restricting the default permissions of the user and requiring them to elevate permissions to perform sensitive commands. Practically speaking, we want to enable these users to view everything by default but deliberately elevate their privileges to be able to write. This model significantly reduces the chances of the preceding scenario occurring.

Let's work through how you might implement the privilege elevation pattern just described. We'll assume that our operations team's user identities are all part of an `ops-team` group in Kubernetes. As mentioned earlier, Kubernetes has no defined concept of a group per se, so we mean that those users all have additional attributes in their Kubernetes identity (x509 cert, OIDC claim, etc.) that identify them as being part of the group.

We create the ClusterRoleBinding that allows users in the `ops-team` group access to the `view` built-in ClusterRole, which is what gives us our default read-only access:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-view
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: ops-team
```

Now we create a ClusterRoleBinding to allow our `cluster-admin` user to have `cluster-admin` ClusterRole permissions on the cluster. Remember, we're not binding

this ClusterRole directly to our `ops-team` group. No user can *directly* identify as the `cluster-admin` user; this will be a user that is *impersonated* and their permissions *assumed* by another authenticated user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-crb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: cluster-admin
```

Finally, we create a ClusterRole called `cluster-admin-impersonator` that allows the impersonation of the `cluster-admin` user, and a ClusterRoleBinding that binds that capability to everyone in the `ops-team` group:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-admin-impersonator
rules:
- apiGroups: [""]
  resources: ["users"]
  verbs: ["impersonate"]
  resourceNames: ["cluster-admin"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-impersonate
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin-impersonator
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: ops-team
```

Now let's use a kubeconfig for a user (john) in the `ops-team` group to see how the elevation of privileges works in practice:

```
$ kubectl get configmaps
No resources found.

$ kubectl create configmap my-config --from-literal=test=test
Error from server (Forbidden): configmaps is forbidden: User "john"
```

```
cannot create resource "configmaps" in API group "" in the namespace "default"

$ kubectl --as=cluster-admin create configmap my-config --from-literal=test=test
configmap/my-config created
```

We used the preceding setup for admin users, although implementing something similiar for every user (having a *team-a* group, a *team-a* view role, and a *team-a* admin user) is a solid pattern that removes a lot of the potential for costly mistakes. Additionally, one of the great things about the impersonation approach just described is that all of this is played out in the Kubernetes audit logs, so we can see the original user log in, impersonate the cluster-admin, and then take action.

Application/Workload Identity

In the previous section, we saw the main methods and patterns for establishing the identity of human users of Kubernetes, and how they can authenticate to the cluster. In this section, we're going to take at look how we can establish identity for our workloads that run in the cluster. There are three main use cases we'll be examining:

- Workloads identifying themselves to other workloads within the cluster, to potentially establish a mutual authentication between them for additional security.
- Workloads identifying themselves to obtain appropriate access to the Kubernetes API itself. This is a common use case for custom controllers that need to watch and act on Kubernetes resources.
- Workloads identifying themselves and authenticating to external services. This could be anything outside of the cluster but will be primarily cloud vendor services running on AWS, GCP, etc.

In "Network Identity" on page 289, we'll look at two of the most popular Container Networking Interface (CNI) tools (Calico and Cilium) and see how they can assign identity and restrict access, primarily for the first use case we just described.

Secondly, we'll move on to service account tokens (SAT) and projected service account tokens (PSAT). These are flexible and important Kubernetes primitives that enable workload-to-workload identity (the first use case) in addition to being the primary mechanism for workloads identifying to the Kubernetes API itself (the second use case).

Next we'll cover options where an application's identity is provided by the platform itself. The most common use case we see in the field is workloads that need access to AWS services, and we'll look at the three main methods that are possible today.

Lastly, we'll extend the concept of platform-mediated identity to consider tooling that aims to provide a consistent model of identity across multiple platforms and environ-

ments. The flexibility of this approach can be used to cover all of the use cases we mentioned, and we'll show how this can be a very powerful capability.

Before implementing any of the patterns described in this section you should definitely evaluate your requirements as they relate to establishing workload-to-workload identity. Often, establishing this capability is an advanced-level activity, and the majority of organizations may not need to solve this topic, at least initially.

Shared Secrets

Most of the discussion around shared secrets for user identity also applies to application identity; however, there are some additional nuances and guidance based on field experience.

Once we have secrets in place that are known by the client and the server, how do we safely rotate them upon expiry? Ideally we want these secrets to have a fixed life span to mitigate the potential damage caused if one were to be compromised. Additionally, because they are *shared*, they need to be redistributed to both the client application and the server. Hashicorp's Vault is a prominent example of an enterprise secret store and features integrations with many tools that get close to the goal of solving this resyncing problem. However, Vault also suffers from the secure introduction problem we first encountered in "User Identity" on page 274.

This is the problem we have when trying to ensure that a shared secret is securely distributed to both the client and the serving entity *before* we have any model of identity and authentication established (the chicken and the egg). Any attempt to initially seed a secret between two entities could be compromised, breaking our guarantee of identity and unique authentication.

Despite the flaws already discussed, shared secrets have one strong advantange in that it is a model that is well supported and understood by almost all users and applications. This makes it a strong choice for cross-platform operability. We will see how to solve the secure introduction problem for Vault and Kubernetes with more advanced methods of authentication later in this chapter. Once Vault is securely configured with those methods, it is a fine choice (and one we have implemented many times) as many of the issues with shared secrets are mitigated.

Network Identity

Network primitives like IP addresses, VPNs, firewalls, etc., have historically been used as a form of identity for controlling which applications have access to what services. However, in a cloud native ecosystem these methods are breaking down and paradigms are changing. In our experience it is important to educate teams across the organization (especially networking and security) on these changes and how practices can (and should) adapt to embrace and accommodate them. Too often this is met

with resistance around concerns over security and/or control. In reality it's possible to achieve almost any posture if required, and time should be taken to understand the *actual requirements* of the teams, rather than getting stuck in implementation details.

In container-based environments, workloads share networking stacks and underlying machines. Workloads are increasingly ephemeral and move between nodes often. This results in a constant churn of IP addresses and network changes.

In a multicloud and API-driven world, the network is no longer a primary boundary. Calls commonly occur to external services across multiple providers, each of which may need a way to prove identity of our calling applications.

Existing traditional (platform level) network primitives (host IP addresses, firewalls, etc.) are no longer suitable for establishing workload identity and, if used at all, should be used only as an additional layer of defense in depth. This is not to say that network primitives *in general* are bad but that they must have additional workload context to be effective. In this section we'll look at how CNI options provide degrees of identity for Kubernetes clusters and how best to leverage them. CNI providers are able to contextualize requests and provide identity by combining network primitives and metadata retrieved from the Kubernetes API. We'll take a brief look at some of the most popular CNI implementations and see what capabilities can they can provide.

Calico

Calico provides network policy enforcement at layers 3 (Network) and 4 (Transport) of the OSI Model, enabling users to restrict communication between Pods based on their Namespace, labels, and other metadata. This enforcement is all enabled by modifying the network configuration `iptables/ipvs`) to allow/disallow IP addresses.

Calico also supports making policy decisions based on Service Accounts using a component called Dikastes when used in combination with Envoy proxy (either standalone Envoy or deployed as part of a service mesh like Istio). This approach enables enforcement at layer 7 (Application), based on attributes of the application protocol (headers, etc.) and relevant cryptographic identities (certificates, etc.).

By default, Istio (Envoy) will only perform mTLS and ensure that workloads present certificates signed by the Istio CA (Citadel). Dikastes runs as a sidecar alongside Envoy as a plug-in, as we can see in the architecture diagram in Figure 10-3. Envoy verifies the CA before consulting Dikastes for a decision on whether to admit or reject the request. Dikastes makes this decision based on user-defined Calico NetworkPolicy or GlobalNetworkPolicy objects:

```yaml
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: summary
```

```
spec:
  selector: app == 'summary'
  ingress:
    - action: Allow
      source:
        serviceAccounts:
          names: ["customer"]
        NamespaceSelector: app == 'bank'
  egress:
    - action: Allow
```

The preceding rule is specifying that the policy be applied to any Pods with the label `app: summary` and restricts access to Pods calling from the `customer` Service Account (in Namespaces with the label `app: bank`). This works because the Calico control plane (the Felix node agent) computes rules by reconciling Pods that are running under a specific Service Account with their IP addresses and subsequently syncing this information to Dikastes via a Unix socket.

This out-of-band verification is important as it mitigates a potential attack vector in an Istio environment. Istio stores each Service Account's PKI assets in a Secret in the cluster. Without this additional verification, an attacker who was able to steal that Secret would be able to masquerade as the asserted Service Account (by presenting those PKI assets), even though it may not be running as that account.

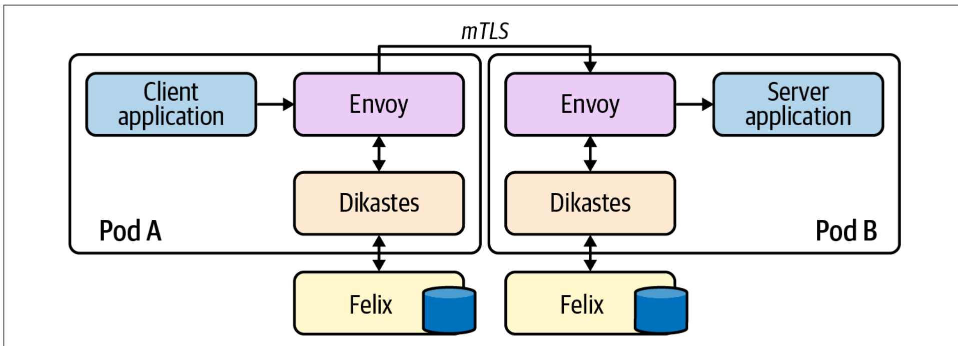


Figure 10-4. Architecture diagram using Dikastes with Envoy.

If your team is leveraging Calico already, then Dikastes can provide an extra layer of defense in depth and should definitely be considered. However, it requires Istio or some other mesh solution (e.g., standalone Envoy) to be available and running in the environment to validate the identity presented by the workload. These claims are not independently cryptographically verifiable, relying on the mesh to be present with every connected Service. This in itself adds a nontrivial level of complexity, and the trade-offs should be carefully evaluated. One strength of this approach is that Calico and Istio are both cross-platform, so this setup could be used to establish identity for

applications running both on and off Kubernetes within an environment (whereas some options we'll see are Kubernetes-only).

Cilium

Like Calico, Cilium also provides network policy enforcement at layers 3 and 4, enabling users to restrict communication between Pods based on their Namespace and other metadata (labels, for example). Cilium also supports (without additional tooling) the ability to apply policy at layer 7 and restrict access to Services via Service Accounts.

Unlike Calico, enforcement in Cilium is not based on IP address (and updating node networking configurations). Instead, Cilium calculates identities for each unique Pod/endpoint (based on a number of selectors) and encodes these identities into each packet. It then enforces whether packets should be allowed based on these identities using eBPF kernel hooks at various points in the datapath.

Let's briefly explore how Cilium calculates identities for an endpoint (Pod). The output of listing Cilium endpoints for an application is shown in the following code. We have omitted the list of labels in the snippet but have added an additional label to the last Pod in the list (`deathstar-657477f57d-zzz65`) that is not present on the other four Pods. As a result of this, we can see that the last Pod is therefore assigned a *different* identity to the previous four. Aside from that single differing label, all the Pods in the Deployment share a Namespace, Service Account, and several other arbitrary Kubernetes labels.

```
$ kubectl exec -it -n kube-system cilium-oid9h -- cilium endpoint list
NAMESPACE     NAME                           ENDPOINT ID   IDENTITY ID
default       deathstar-657477f57d-jpzgb     1474          1597
default       deathstar-657477f57d-knxrl     2151          1597
default       deathstar-657477f57d-xw2tr     16            1597
default       deathstar-657477f57d-xz2kk     2237          1597
default       deathstar-657477f57d-zzz65     1             57962
```

If we removed the divergent label, the `deathstar-657477f57d-zzz65` Pod would be reassigned the same identity as its four peers. This level of granularity gives us a lot of power and flexibility when assigning identities to individual Pods.

Cilium implements the Kubernetes-native NetworkPolicy API, and like Calico also exposes more fully featured capabilities in the form of CiliumNetworkPolicy and CiliumClusterwideNetworkPolicy objects:

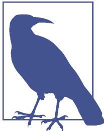
```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "k8s-svc-account"
spec:
  endpointSelector:
    matchLabels:
```

```yaml
      io.cilium.k8s.policy.serviceaccount: leia
  ingress:
  - fromEndpoints:
    - matchLabels:
        io.cilium.k8s.policy.serviceaccount: luke
    toPorts:
    - ports:
      - port: '80'
        protocol: TCP
      rules:
        http:
        - method: GET
          path: "/public$"
```

In this example, we are using special `io.cilium.k8s.policy.*` label selectors to target specific Service Accounts in the cluster. Cilium then uses its registry of identities (that we saw previously) to restrict/allow access as necessary. In the policy shown, we are restricting access to the path `/public` on port 80 for Pods with the `leia` Service Account. We are allowing access only from Pods with the `luke` Service Account.

Like Calico, Cilium is cross-platform so can be used across Kubernetes and non-Kubernetes environments. Cilium *is* required to be present with every connected Service for identities to be verifiable, so the overall complexity of your networking setup can increase with this approach. However, Cilium doesn't require a service mesh component to operate.

Service Account Tokens (SAT)



Service Accounts are primitives in Kubernetes that provide identity for groups of Pods. Every Pod runs under a Service Account. If a Service Account is not pre-created by an administrator and assigned to a Pod, they are assigned a default Service Account for the Namespace they reside in.

Service Account tokens are JSON Web Tokens (JWT) that are created as Kubernetes Secrets. Each Service Account (including the default Service Account) has a corresponding Secret that contains the JWT. Unless otherwise specified, these tokens are mounted into each Pod running under that Service Account and can be used to make requests to the Kubernetes API (and as this section shows, other services).

Kubernetes Service Accounts provide a way of assigning identity to a set of workloads. Role-Based Access Control (RBAC) rules then can be applied within the cluster to limit the scope of access for a specific Service Account. Service Accounts are the way that Kubernetes itself usually authenticates in-cluster access to the API:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: default
secrets:
- name: default-token-mf9v2
```

When a Service Account is created, an associated Secret is also created containing a unique JWT identifying the account:

```yaml
apiVersion: v1
data:
  ca.crt: <.. SNIP ...>
  namespace: ZGVmYXVsdA==
  token: <.. SNIP ...>
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: default
    kubernetes.io/service-account.uid: 59aee446-b36e-420f-99eb-a68895084c98
  name: default-token-mf9v2
  namespace: default
type: kubernetes.io/service-account-token
```

By default, Pods will automatically get the `default` Service Account token for their Namespace mounted if they do not specify a specific Service Account to use. This can (and should) be disabled to ensure that all Service Account tokens are explicitly mounted to Pods and their access scopes are well understood and defined (rather than falling back and assuming a default).

To specify a Service Account for a Pod, use the `serviceAccountName` field in the Pod spec:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: my-pod-sa
# Additional fields removed for brevity
```

This will cause the Service Account's Secret (containing the token) to be mounted into the Pod at `/var/run/secrets/kubernetes.io/serviceaccount/`. The application can retrieve the token and use it in a request to other applications/services in the cluster.

The destination application can verify the provided token by calling the Kubernetes `TokenReview` API:

```
curl -X "POST" "https://<kubernetes API IP>:<kubernetes API Port>\
/apis/authentication.k8s.io/v1/tokenreviews" \
```

```
     -H 'Authorization: Bearer <token>' \ ❶
     -H 'Content-Type: application/json; charset=utf-8' \
     -d $'{
  "kind": "TokenReview",
  "apiVersion": "authentication.k8s.io/v1",
  "spec": {
    "token": "<token to verify>" ❷
  }
}'
```

❶ This token is the Secret mounted into the destination application's Pod, allowing it to communicate with the API server.

❷ This token is the one the calling application has presented as proof of identity.

The Kubernetes API will respond with metadata about the token to be verified, in addition to whether or not it has been authenticated:

```
{
  "kind": "TokenReview",
    "apiVersion": "authentication.k8s.io/v1",
    "metadata": {
        "creationTimestamp": null
    },
    "spec": {
        "token": "<token to verify>"
    },
    "status": {
        "authenticated": true,
        "user": {
            "username": "system:serviceaccount:default:default",
            "uid": "4afdf4d0-46d2-11e9-8716-005056bf4b40",
            "groups": [
                "system:serviceaccounts",
                "system:serviceaccounts:default",
                "system:authenticated"
            ]
        }
    }
}
```

The preceding flow is shown in Figure 10-5.

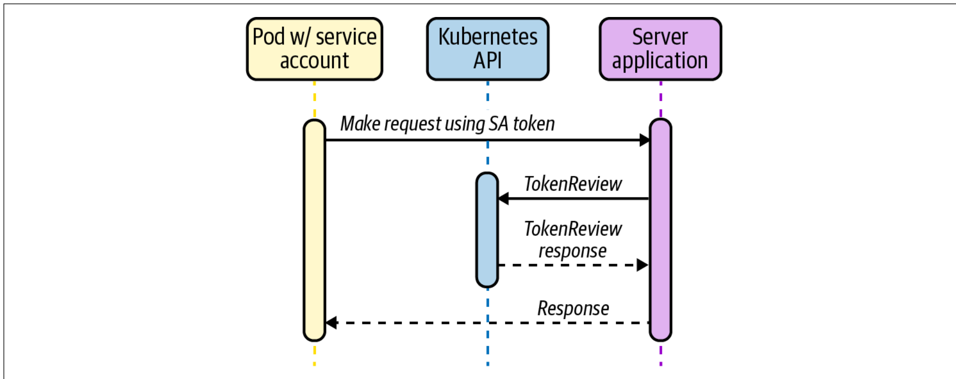
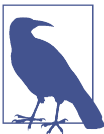


Figure 10-5. Service Account tokens.

Service Account tokens have been part of Kubernetes since very early on and provide a tight integration with the platform in a consumable format (JWT). We as operators also have a fairly tight control on their validity as tokens are invalidated if the Service Account or Secret is deleted. However, they have some features that make their use as identifiers suboptimal. Most importantly, the tokens are scoped to a specific Service Account, so are unable to validate anything with a more granular scope, for example a Pod or a single container. We also need to add functionality to our applications if we want to use and verify tokens as a form of client identity. This involves calling the TokenReview API with some custom component.

Tokens are also scoped to a single cluster, so we're not able to use Service Account tokens issued by one cluster as identity documents for services calling from other clusters without exposing each cluster's TokenReview API and encoding some additional metadata about the cluster where the request originated. All of this adds significant complexity to the setup, so we'd recommend not going down this path as a method of cross-cluster service identity/authentication.



To ensure that permissions can be granted to applications in an appropriately granular way, unique Service Accounts should be created for each workload that requires access to the Kubernetes API server. Additionally, if a workload *does not* require access to the Kubernetes API server, disable the mounting of a Service Account token by specifying the `automountServiceAccountToken: false` field on the `ServiceAccount` object.

For example, this can be set on the default Service Account for a Namespace to disable the auto-mounting of the credential token. This field can also be set on the `Pod` object, but note that the `Pod` field takes precedence if it's set in both places.

Projected Service Account Tokens (PSAT)

Beginning with Kubernetes v1.12 there is an additional method of identity available that builds on the ideas in service account tokens but seeks to address some of the weaknesses (such as lack of TTL, wide scoping, and persistence).

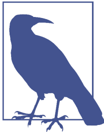
In order for the PSAT flow to function correctly, the Kubernetes API server needs to be configured with the parameter keys shown here (all are configurable, though):

```
spec:
  containers:
  - command:
    - kube-apiserver
    - --service-account-signing-key-file=/etc/kubernetes/pki/sa.key
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-account-issuer=api
    - --service-account-api-audiences=api
    # Additional flags removed for brevity
    image: k8s.gcr.io/kube-apiserver:v1.17.3
```

The flow for establishing and verifying identity is similar to the SAT method. However, instead of having our Pod/application read the automounted Service Account token, you instead mount a projected Service Account token as a Volume. This also injects a token into the Pod, but you can specify a TTL and custom audience for the token:

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  labels:
    app: test
spec:
  serviceAccountName: test
  containers:
  - name: test
    image: ubuntu:bionic
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: app-token
  volumes:
  - name: app-token
    projected:
      sources:
      - serviceAccountToken:
          audience: api ❶
          expirationSeconds: 600
          path: app-token
```

❶ The `audience` field is important because it prevents destination applications using the token from the calling application and attempting to masquerade as the calling application. The audience should always be scoped correctly depending on the destination application. In this case, we are scoping to communicate with the API server itself.



When using PSAT, a designated Service Account must be created and used. Kubernetes does not mount PSATs for Namespace default Service Accounts.

The calling application can read the projected token and use that in requests within the cluster. Destination applications can verify the token by calling the `TokenReview` API and passing the received token. With the PSAT method, the review will also verify that the TTL has not expired and will return additional metadata about the presenting application, including specific Pod information. This provides a tighter scope than regular SATs (which only assert a Service Account).

```
// Additional fields removed for brevity
"extra": {
  "authentication.kubernetes.io/pod-name": ["test"],
  "authentication.kubernetes.io/pod-uid":
    ["8b9bc1be-c71f-4551-aeb9-2759887cbde0"]
}
```

As shown in Figure 10-6, there is no real difference between the SAT and PSAT flows themselves (aside from the server verifying the `audience` field), only in the validity and granularity of the identity asserted by the token. The `audience` field is important as it identifies the intended recipient of the token. In keeping with the JWT official specification, the API will reject a token whose audience does not match the audience specified in the API server configuration.

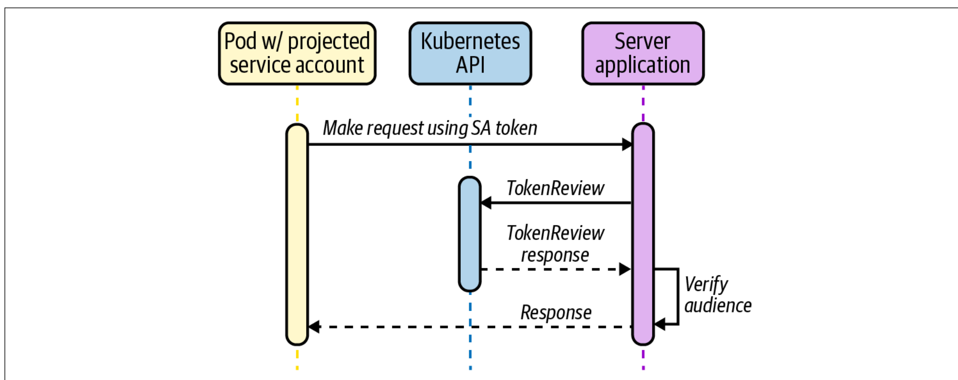


Figure 10-6. Projected Service Account tokens.

Projected Service Account tokens are a relatively recent but incredibly strong addition to Kubernetes' feature set. On their own they provide tight integration with the platform itself, they provide configurable TTLs, and they have a tight scope (individual Pods). They can also be used as building blocks to construct even more robust patterns (as we'll see in later sections).

Platform Mediated Node Identity

In cases where all workloads are running on a homogeneous platform (for example, AWS), it is possible for the platform itself to determine and assign identities to workloads because of the contextual metadata they possess about the workload.

Identity is not asserted by the workload itself but is determined based on its properties by an out-of-band provider. The provider returns the workload a credential to prove identity that may be used to communicate with other services on the platform. It then becomes trivial for the other services to verify that credential because they too are on the same underlying platform.

On AWS, an EC2 instance may request credentials to connect to a different service like an S3 bucket. The AWS platform inspects the metadata of the instance and can provide role-specific credentials back to the instance with which to make the connection, as shown in Figure 10-7.

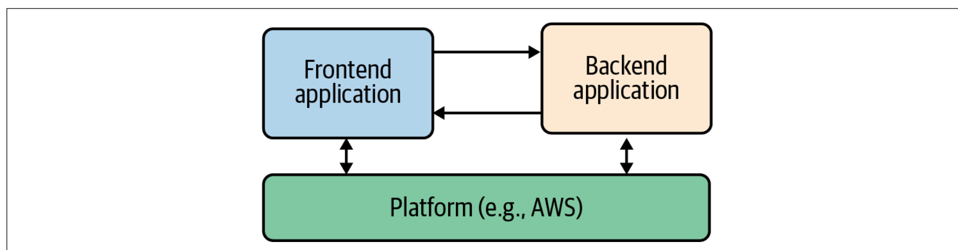
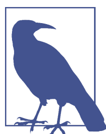


Figure 10-7. Platform mediated identity.



Remember that the platform still has to perform *authorization* on the request to ensure that the identity being used has the appropriate permissions. This method is only being used to *authenticate* the request.

Many cloud vendors expose functionality described in this section. We're choosing to focus on tooling that applies to and integrates with Amazon Web Services (AWS) because it is the vendor we most commonly see in the field.

AWS platform authentication methods/tooling

AWS provides a strong identity solution at the node level via the EC2 metadata API. This is an example of a platform-mediated system, whereby the platform (AWS) is able to determine the identity of a calling entity based on a number of intrinsic properties without the entity asserting any credentials/identity claim itself. The platform can then deliver secure credentials to the instance (in the form of a role, for example) that allows it to access any services defined by the relevant policies. As a whole this is referred to as Identity and Access Management (IAM).

This model underpins how AWS (and many other vendors) provide secure access to their own cloud services. However, with the rise of containers and other multitenant application models, this per-node identity/authentication system breaks down and requires additional tooling and alternative approaches.

In this section we'll look at the three main tooling options we encounter in the field. We'll cover kube2iam and kiam, two separate tools that share the same approximate implementation model (and therefore have similar advantages and disadvantages). We'll also describe why we don't recommend those tools today and why you should consider a more integrated solution such as the final option we cover, IAM Roles for Service Accounts (IRSA).

kube2iam. kube2iam is an open source (OSS) tool that acts as a proxy between running workloads and the AWS EC2 metadata API. The architecture is shown in Figure 10-8.

kube2iam requires that every node in the cluster be able to assume a superset of all the roles that Pods may require. This security model means that the scope of access provided should a container breakout occur is potentially huge. For this reason it is strongly advised not to use kube2iam. We are discussing it here as we regularly encounter it in the field and want to ensure that you are aware of the limitations of the implementation before diving in.

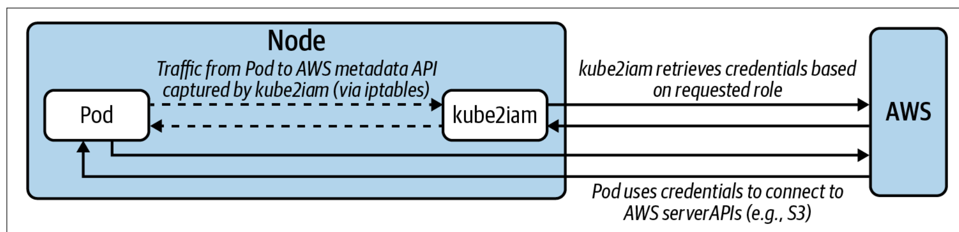


Figure 10-8. kube2iam architecture and data flow.

kube2iam Pods run on every node via a DaemonSet. Each Pod injects an iptables rule to capture outbound traffic to the metadata API and redirect it to the running instance of kube2iam on that node.

Pods that want to interact with AWS APIs should specify the role they want to assume as an annotation in the spec. For example, in the following Deployment spec you can see the role is specified in the `iam.amazonaws.com/role` annotation:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      annotations:
        iam.amazonaws.com/role: <role-arn>
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.1
        ports:
        - containerPort: 80
```

kiam. Like kube2iam, kiam is an open source (OSS) tool that acts as a proxy to the AWS EC2 metadata API, although its architecture (and as a result, its security model) are different and slightly improved, as shown in Figure 10-9.

While safer than kube2iam, kiam also introduces a potentially serious security flaw. This section describes a mitigation of the flaw, but you should still use caution and understand the attack vector when using kiam.

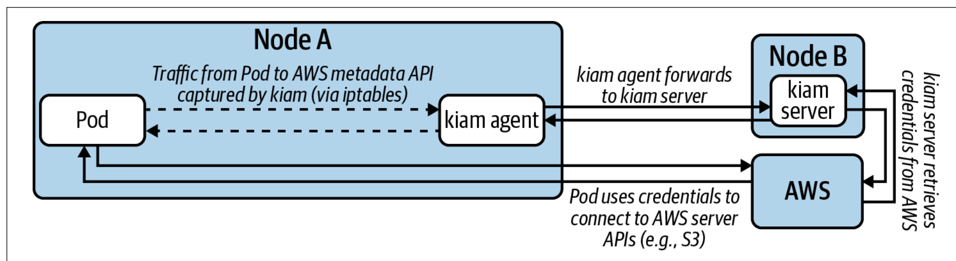


Figure 10-9. kiam architecture and data flow.

kiam has both server and agent components. The agents run as a DaemonSet on every node in the cluster. The server component can (and should) be restricted to the either the control-plane nodes or a subset of cluster nodes. Agents capture EC2 metadata API requests and forward them to the server components to complete the appropriate authentication with AWS. Only the server nodes require access to assume AWS IAM roles (again, a superset of all roles that Pods may require), as shown in Figure 10-10.

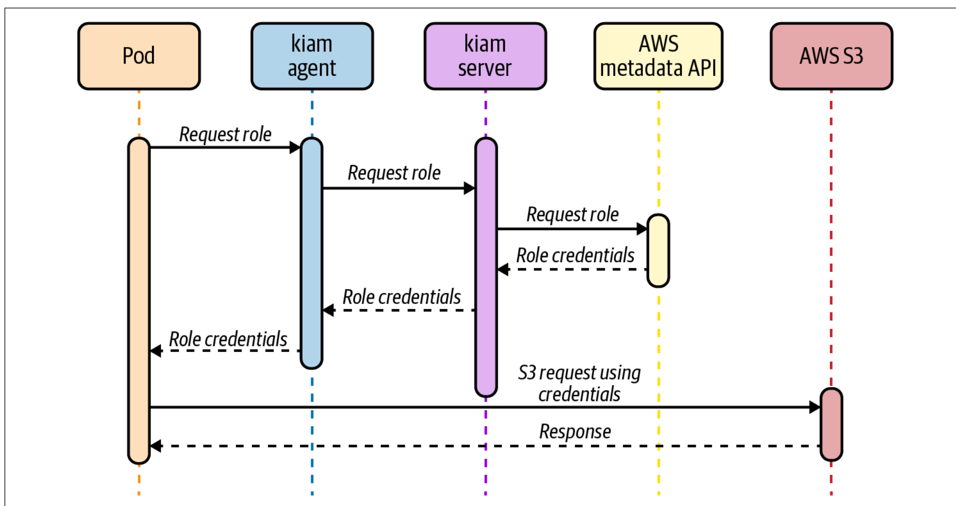


Figure 10-10. kiam flow.

In this model, there should be controls in place to ensure that no workloads are able to run on the server nodes (and thereby obtain unfettered AWS API access). Assumption of roles is achieved (like kube2iam) by annotating Pods with the desired role:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  template:
    metadata:
      annotations:
        iam.amazonaws.com/role: <role-arn>
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.9.1
```

```
        ports:
        - containerPort: 80
```

While the security model is better than kube2iam, kiam still has a potential attack vector whereby if a user is able to directly schedule a Pod onto a node (by populating its `nodeName` field, bypassing the Kubernetes scheduler and any potential guards) they would have unrestricted access to the EC2 metadata API.

The mitigation for this issue is to run a mutating or validating admission webhook that ensures the `nodeName` field is not prepopulated on Pod create and update requests to the Kubernetes API.

kiam provides a strong story for enabling individual Pods to access AWS APIs, using a model that existing AWS users will be familiar with (role assumption). This is a viable solution in many cases, provided the preceding mitigation is put in place prior to use.

IAM Roles for Service Accounts (IRSA). Since late 2019, AWS has provided a native integration between Kubernetes and IAM called IAM Roles for Service Accounts (IRSA).

At a high level, IRSA exposes a similiar experience to kiam and kube2iam, in that users can annotate their Pods with an AWS IAM role they want it to assume. The implementation is very different, though, eliminating the security concerns of the earlier approaches.

AWS IAM supports federating identity out to a third-party OIDC provider, in this case the Kubernetes API server. As you saw already with PSATs, Kubernetes is capable of creating and signing short-lived tokens on a per-Pod basis.

AWS IRSA combines these features with an additional credential provider in their SDKs that calls `sts:AssumeRoleWithWebIdentity`, passing the PSAT. The PSAT and desired role need to be injected as environment variables within the Pod (there is a webhook that will do this automatically based on the `serviceAccountName` desired):

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: myapp
spec:
  serviceAccountName: my-serviceaccount
  containers:
  - name: myapp
    image: myapp:1.2
    env:
    - name: AWS_ROLE_ARN
      value: "arn:aws:iam::123456789012:role/\
        eksctl-irptest-addon-iamsa-default-my-\
        serviceaccount-Role1-UCGG6NDYZ3UE"
    - name: AWS_WEB_IDENTITY_TOKEN_FILE
```

```
          value: /var/run/secrets/eks.amazonaws.com/serviceaccount/token
        volumeMounts:
        - mountPath: /var/run/secrets/eks.amazonaws.com/serviceaccount
          name: aws-iam-token
          readOnly: true
      volumes:
      - name: aws-iam-token
        projected:
          defaultMode: 420
          sources:
          - serviceAccountToken:
              audience: sts.amazonaws.com
              expirationSeconds: 86400
              path: token
```

Kubernetes does not natively expose a `.well-known` OIDC endpoint, so there is some additional work required to configure this at a public location (static S3 bucket) so that AWS IAM can verify the token using Kubernetes' public Service Account signing key.

Once verified, AWS IAM responds to the application's request, exchanging the PSAT for the desired IAM role credentials as shown in Figure 10-11.

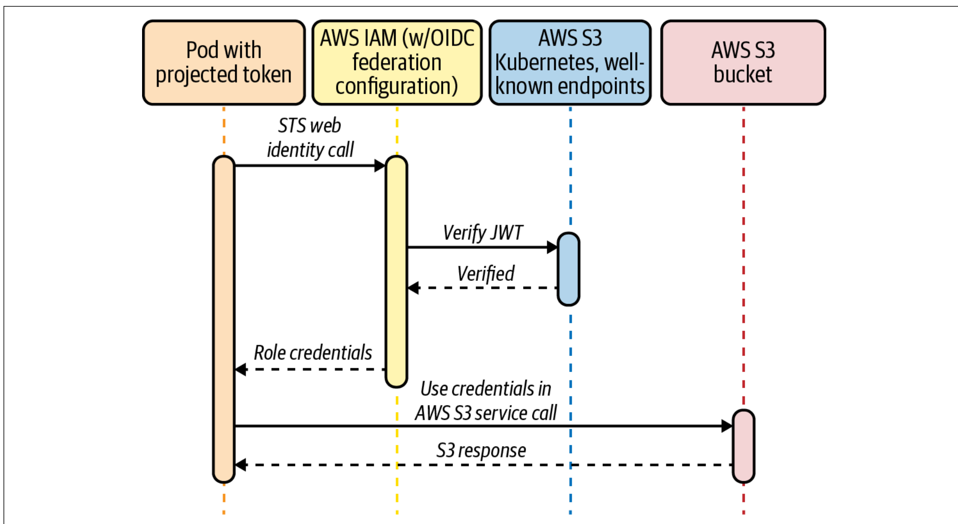


Figure 10-11. IAM roles for Service Accounts.

Although the setup for IRSA is a little clunky, it possesses the best security model of all approaches to Pod IAM Role assumption.

IRSA is a strong choice for organizations already leveraging AWS services as it uses patterns and primitives that will be familiar with your operations and development teams. The model employed (mapping Service Accounts to IAM roles) is also a straightforward one to understand with a strong security model.

The main downside is that IRSA can be somewhat cumbersome to deploy and configure if you are not utilizing the Amazon Elastic Kubernetes Service (EKS). However, recent additions to Kubernetes itself will alleviate some of the technical challenges here, such as exposing Kubernetes itself as an OIDC provider.

As we saw in this section, mediating identity through a common platform (AWS in this case) has many strengths. In the next section we'll dive into tooling that is aiming to implement this same model but capable of spanning *multiple* underlying platforms. This brings the control of a centralized identity system with the flexibility of running it for any workload across any cloud or platform.

Cross-platform identity with SPIFFE and SPIRE

Secure Production Identity Framework for Everyone (SPIFFE) is a standard that specifies a syntax for identity (SPIFFE Verifiable Identity Document, SVID) that can leverage existing cryptographic formats such as x509 and JWT. It also specifies a number of APIs for providing and consuming these identities. A SPIFFE ID takes the form `spiffe://trust-domain/hierarchical/workload`, where all sections after the `spiffe://` are arbitrary string identifiers that can be used in multiple ways (although creating some kind of hierarchy is most common).

SPIFFE Runtime Environment (SPIRE) is the reference implementation of SPIFFE and has a number of SDKs and integrations to allow applications to make use of (both providing, and consuming) SVIDs.

This section will assume use of SPIFFE and SPIRE together unless otherwise noted.

Architecture and concepts. SPIRE runs a server component that acts as a signing authority for identities and maintains a registry of all workload identities and the conditions required for an identity document to be issued.

SPIRE agents run on every node as a DaemonSet where they expose an API for workloads to request identity via a Unix socket. The agent is also configured with read-only access to the kubelet to determine metadata about Pods on the node. The SPIRE architecture is shown in Figure 10-12.

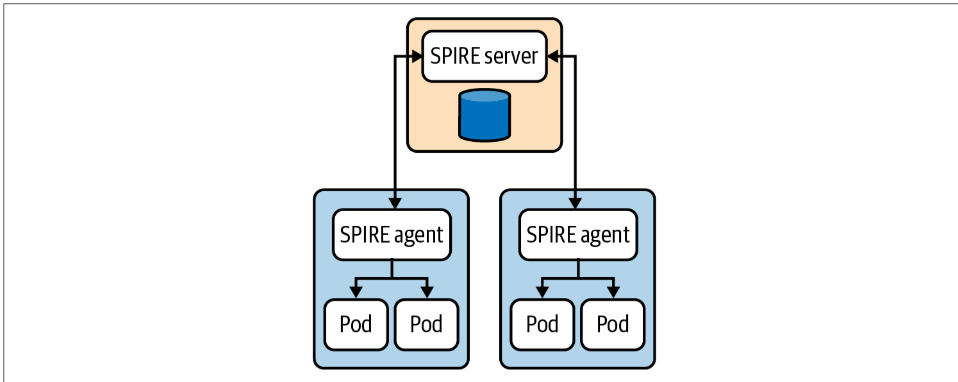


Figure 10-12. SPIRE Architecture. Reproduced from the official SPIRE documentation.

When agents come online they verify and register themselves to the server by a process called *node attestation* (as shown in Figure 10-13). This process utilizes environmental context (for example, the AWS EC2 metadata API or Kubernetes PSATs) to identify a node and assign it a SPIFFE ID. The server then issues the node an identity in the form of an x509 SVID. Following is an example registration for a node:

```
/opt/spire/bin/spire-server entry create \
  -spiffeID spiffe://production-trust-domain/nodes \
  -selector k8s_psat:cluster:production-cluster \
  -selector k8s_psat:agent_ns:spire \
  -selector k8s_psat:agent_sa:spire-agent \
  -node
```

This tells the SPIRE server to assign the SPIFFE ID `spiffe://production-trust-domain/nodes` to any node where the agent Pod satisfies the selectors specified; in this case, we are selecting when the Pod is running in the SPIRE Namespace on the `production-cluster` under the `spire-agent` Service account (verified via the PSAT).

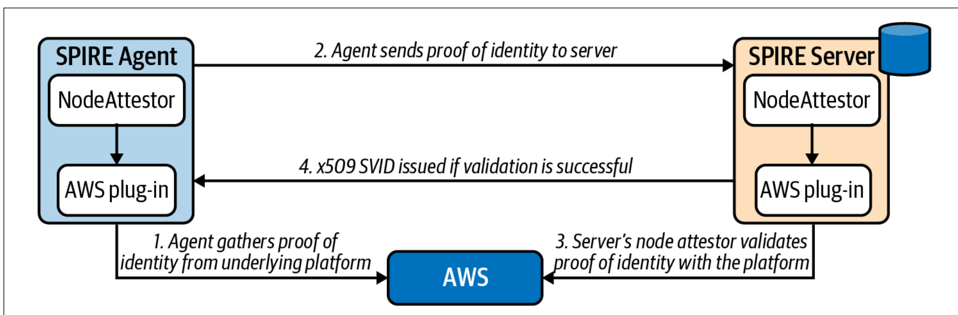


Figure 10-13. Node attestation. Reproduced from the official SPIRE documentation.

When workloads come online they call the node-local workload API to request an SVID. The SPIRE agent uses information available to it on the platform (from the

kernel, kubelet, etc.) to determine the properties of the calling workload. This process is referred to as *workload attestation* (as shown in Figure 10-14). The SPIRE server then matches the properties against known workload identities based on their selectors and returns an SVID to the workload (via the agent) that can be used for authentication against other systems:

```
/opt/spire/bin/spire-server entry create \
  -spiffeID spiffe://production-trust-domain/service-a \
  -parentID spiffe://production-trust-domain/nodes \
  -selector k8s:ns:default \
  -selector k8s:sa:service-a \
  -selector k8s:pod-label:app:frontend \
  -selector k8s:container-image:docker.io/johnharris85/service-a:v0.0.1
```

This tells the SPIRE server to assign the SPIFFE ID `spiffe://production-trust-domain/service-a` to any workload that:

- Is running on a node with ID `spiffe://production-trust-domain/nodes`.
- Is running in the `default` Namespace.
- Is running under the `service-a` Service Account.
- Has the Pod label `app: frontend`.
- Was built using the `docker.io/johnharris85/service-a:v0.0.1` image.

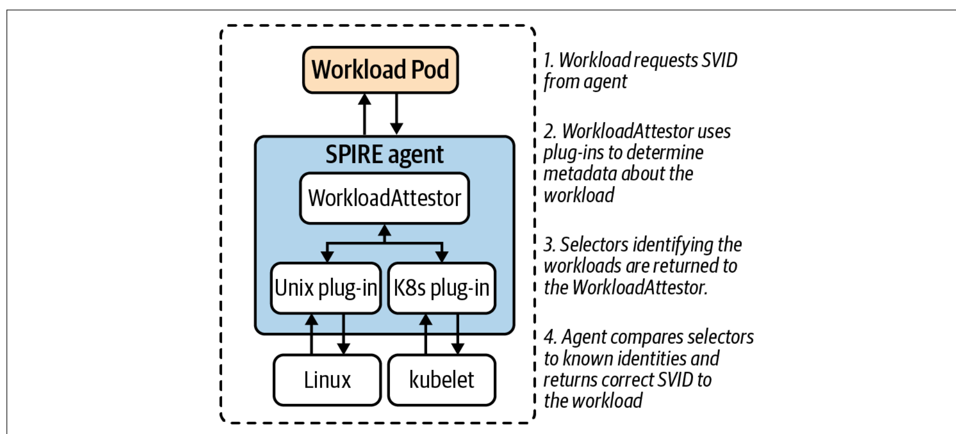
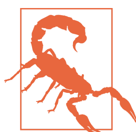


Figure 10-14. Workload attestation. Reproduced from the official SPIRE documentation.



Note that the workload attestor plug-in can query the kubelet (to discover workload information) using its Service Account. The kubelet then uses the `TokenReview` API to validate bearer tokens. This requires reachability to the Kubernetes API server. Therefore, API server downtime can interrupt workload attestation.

The `--authentication-token-webhook-cache-ttl` kubelet flag controls how long the kubelet caches TokenReview responses and may help to mitigate this issue. A large cache TTL value is not recommended, however, as that can impact permission revocation. See the SPIRE workload attestor documentation for more details.

The patterns described in this section have significant advantages when trying to build a robust identity system for your workloads, both on and off Kubernetes. The SPIFFE specification leverages well-understood and widely supported cryptographic standards in x509 and JWT, and the SPIRE implementation also supports many different methods of application integrations. Another key property is the ability to scope identity to a very granular level by combining projected service account tokens with its own selectors to identify individual Pods. This can be especially useful in scenarios where sidecar containers are present in a Pod and each container needs varying levels of access.

This approach is also undeniably the most labor-intensive and requires expertise in the tooling and effort to maintain another component in the environment. There may also be work required to register each workload, although this could be automated (and work is already underway in the community around the area of automated registration of workloads).

SPIFFE/SPIRE have a number of integration points with workload applications. Which integration point is appropriate will depend on the desired level of coupling to the platform and the amount of control users have over the environment.

Direct application access. SPIRE provides SDKs for Go, C, and Java for applications to directly integrate with the SPIFFE workload API. These wrap existing HTTP libraries but provide native support for obtaining and verifying identities. Following is an example in Go calling a Kubernetes Service `service-b` and expecting a specific SPIFFE ID to be presented (through an x509 SVID):

```
err := os.Setenv("SPIFFE_ENDPOINT_SOCKET",
  "unix:///run/spire/sockets/agent.sock")
conn, err := spiffe.DialTLS(ctx, "tcp", "service-b",
  spiffe.ExpectPeer("spiffe://production-trust-domain/service-b"))
if err != nil {
  log.Fatalf("Unable to create TLS connection: %v", err)
}
```

The SPIRE agent also exposes a gRPC API for those users who want a tighter integration with the platform but are working in a language without SDK availability.

Direct integration (as described in this subsection) is *not* a recommended approach for end-user applications for the following reasons:

- It tightly couples the application with the platform/implementation.
- It requires mounting the SPIRE agent Unix socket into the Pod.

- It's not easily extensible.

The main area where using these libraries directly is appropriate is if building out some intermediate platform tooling that wraps or extends some of the existing functionality of the toolset.

Sidecar proxy. SPIRE natively supports the Envoy SDS API for publishing certificates to be consumed by an Envoy proxy. Envoy can then use the SVID x509 certificate to establish TLS connections with other Services and use the trust bundle to verify incoming connections.

Envoy also supports verifying that only specific SPIFFE IDs (encoded into the SVID) should be able to connect. There are two methods to implement this verification:

- By specifying a list of `verify_subject_alt_name` values in the Envoy configuration.
- By utilizing Envoy's External Authorization API to delegate admission decisions to an external system (for example, Open Policy Agent). Following is an example of a Rego policy to achieve this:

```
package envoy.authz

import input.attributes.request.http as http_request
import input.attributes.source.address as source_address

default allow = false

allow {
    http_request.path == "/api"
    http_request.method == "GET"
    svc_spiffe_id == "spiffe://production-trust-domain/frontend"
}

svc_spiffe_id = client_id {
    [_, _, uri_type_san] := split(
      http_request.headers["x-forwarded-client-cert"], ";")
    [_, client_id] := split(uri_type_san, "=")
}
```

In this example, Envoy verifies the request's TLS certificate against the SPIRE trust bundle, then delegates authorization to Open Policy Agent (OPA). The Rego policy inspects the SVID and allows the request if the SPIFFE ID matches `spiffe://production-trust-domain/frontend`. The architecture for this flow is shown in Figure 10-15.

This approach inserts OPA into the critical request path, so that should be taken into consideration when designing the flow/architecture.

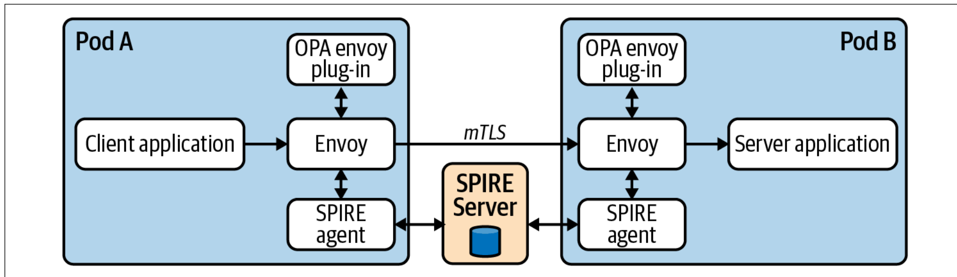
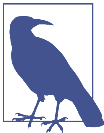


Figure 10-15. SPIRE with Envoy.

Service mesh (Istio). Istio's CA creates SVIDs for all Service Accounts, encoding a SPIFFE ID in the format `spiffe://cluster.local/ns/<namespace>/sa/<service_account>`. Therefore, Services in an Istio mesh can leverage SPIFFE-aware endpoints.



While service meshes are out of scope for this chapter, many attempt to address the issue of identity and authentication. Most of these attempts include or build on the methods and tooling detailed in this chapter.

Other application integration methods. In addition to the primary methods just discussed, SPIRE also supports the following:

- Pulling SVIDs and trust bundles directly to a filesystem, enabling applications to detect changes and reload. While this enables applications to be somewhat agnostic to SPIRE, it also opens an attack vector for certificates to be stolen from the filesystem.
- Nginx module that allows for certificates to be streamed from SPIRE (similiar to the Envoy integration described earlier). There are custom modules for Nginx that enable users to specify the SPIFFE IDs that should be allowed to connect to the server.

Integration with secrets store (Vault). SPIRE can be used to solve the secure introduction problem when an application needs to obtain some shared secret material from HashiCorp Vault. Vault can be configured to authenticate clients using OIDC federation with the SPIRE server as an OIDC provider.

Roles in Vault can be bound to specific subjects (SPIFFE IDs) so that when a workload requests a JWT SVID from SPIRE, that is valid to obtain a role and therefore accessor credentials to Vault.

Integration with AWS. SPIRE can also be used to establish identity and authenticate to AWS services. This process utilizes the same OIDC federation idea in the AWS IRSA and Vault sections. Workloads request JWT SVIDs that are then verified by AWS by validating against the federated OIDC provider (SPIRE server). The downside of this approach is that SPIRE must be publicly accessible for AWS to discover the JSON Web Key Set (JWKS) material required to validate the JWTs.

Summary

In this chapter we have dived into the patterns and tooling that we have successfully seen and implemented in the field.

Identity is a multilayered topic, and your approach will evolve over time as you become more comfortable with the complexity of the different patterns and how that fits with each individual organization's requirements. Typically on the user identity side you will already have a third-party SSO of some kind, but directly integrating this into Kubernetes via OIDC might seem nontrivial. In these situations we've seen Kubernetes sit *outside* of the main organizational identity strategy. Depending on requirements this may be fine, but integrating directly will give greater visibility and control over environments, especially those with multiple clusters.

On the workload/application side we have often experienced this being treated as an afterthought (beyond default Service Accounts). Again, depending on internal requirements this may be fine. It's definitely true that implementing a robust solution for workload identity both in-cluster and cross-platform introduces (in some cases) significant complexity and requires deeper knowledge of external tooling. However, when organizations reach a level of maturity with Kubernetes, we think implementing the patterns described in this chapter can significantly increase the security posture of your Kubernetes environments and provide additional layers of defense in depth should breaches occur.

CHAPTER 11

Building Platform Services

Platform services are those components that are installed in order to add features to the application platform. They are usually deployed as containerized workloads into some `*-system` Namespace and are maintained by the platform engineering team. These platform services are distinct from the workloads managed by platform tenants, which are maintained by the application development teams.

The cloud native ecosystem is rich with projects you can use as part of your application platform. Additionally, there are throngs of vendors that will be happy to provide platform service solutions. Use these wherever they pass the cost benefit analysis. They may even take you all the way to your app platform destination. But we have found that it's common for enterprise users of Kubernetes-based platforms to build custom components. You may have to integrate your Kubernetes-based platform with some existing in-house system. You may have some unique, sophisticated workload requirements to meet. You may have some edge cases to account for that are uncommon or specific to your business needs. Whatever the circumstance, this chapter addresses the details of extending your application platform with custom solutions to fill these gaps.

Central to this notion of building custom platform services is the effort to remove human toil. There is more to this than just automation. Automation is the keystone, but integration of automated components is the mortar. Smooth, reliable interaction of systems is both challenging and critical. This concept of API-driven software is powerful because it fosters integration of software systems. This is part of why Kubernetes has achieved such wide adoption: it enables API-driven behavior for your entire platform without the need to build and expose an API for every piece of software you add to your platform. This software can leverage the Kubernetes API by either managing core resources or adding custom resources to represent the state of new objects. If we follow these patterns of integrated automation as we build our platform

services, we stand to remove tremendous human toil. And if we succeed in this effort, we will open up greater opportunities for innovation, development, and advancement.

In this chapter we are addressing how to extend the Kubernetes control plane. We are taking the effective engineering patterns used by Kubernetes and using those same patterns to build upon those systems. We will spend much of this chapter exploring Kubernetes operators, their design pattern and use cases, and how to develop them. However, it's important that we first take a tour of the points of extension of Kubernetes so that we can maintain a holistic view of building platform services. It's important that we have a clear context and apply solutions that are harmonious with the broader system. Finally, we will examine how we may extend possibly the most important Kubernetes controller in the ecosystem: the scheduler.

Points of Extension

Kubernetes is a wonderfully extensible system. This is certainly one of its most powerful features. A common critical error in software development is attempting to add features to meet every imaginable use case. The system can quickly become a maze of options with unclear paths to outcomes. Furthermore, it often becomes unstable as internal dependencies grow and brittle connections between components of the system erode reliability. There is good reason behind the central tenets of the Unix philosophy to do one thing well and make it interoperable. Kubernetes could never provide for every possible requirement users might encounter while orchestrating their containerized workloads. That would be an impossible system to build. The core functions it does provide are challenging enough. As it is, Kubernetes is a relatively complex distributed software system, even with a pretty narrow set of concerns. It could never meet every requirement, and it doesn't need to since it offers points of extension that allow specialized needs to be met by specialized solutions that may be readily integrated. It can be extended and customized to meet virtually any requirements you may have.

The context of what we are calling plug-in extensions that satisfy defined interfaces with Kubernetes are largely covered elsewhere in this book, as are some of the popular webhook extension solutions. We present a quick review of these here to paint a picture around the operator extension topic, which is where we will spend considerable time in this chapter.

Plug-in Extensions

This is a broad class of extensions that generally help integrate Kubernetes with adjacent systems that are important, and often essential, to running workloads on Kubernetes. They are specifications that third parties can use to implement solutions, rather than implementations themselves:

Network
: The Container Network Interface (CNI) defines the interface that must be satisfied by a plug-in to provide a network for containers to connect to. There are many plug-ins that exist to fulfill this requirement but they all must satisfy the CNI. This topic is covered in Chapter 5.

Storage
: The Container Storage Interface (CSI) provides a system for exposing storage systems to containerized workloads. Again, there are many different volume plug-ins that expose storage from different providers. This topic is explored in Chapter 4.

Container Runtime
: The Container Runtime Interface (CRI) defines a standard for the operations that need to be exposed by a container runtime such that the kubelet does not care what runtime is in use. Docker has historically been the most popular, but there are now others with their own strengths that have become popular. We discuss this topic in detail in Chapter 3.

Devices
: The Kubernetes device plug-in framework allows workloads to access devices on underlying nodes. The most common example of this we have found in the field is for graphics processing units (GPUs) used by compute-intensive workloads. Node pools are often added to clusters for nodes that have these specialized devices, allowing workloads to be assigned to them. See Chapter 2 for more on this topic.

The development of these plug-ins is usually carried out by vendors that either support or sell products that are integrated. It is very rare in our experience to find platform developers building custom solutions in this area. Instead, it is generally a matter of evaluating the available options and leveraging those that meet your requirements.

Webhook Extensions

Webhook extensions act as a backend server for the Kubernetes API server to call to fulfill custom renditions of core API functionality. There are several steps that each request goes through when it arrives at the API server. The client is authenticated to ensure they are permitted access (AuthN). The API checks to ensure the client is authorized to perform the action they are requesting (AuthZ). The API server mutates the resource as called for by enabled admission plug-ins. The resource schema is validated, and any specialized or custom validation is performed by validating admission control. Figure 11-1 illustrates the relationship between the clients, the Kubernetes API, and the webhook extensions leveraged by the API.

Authentication extensions

Authentication extensions, such as OpenID Connect (OIDC), provide the opportunity to offload the task of authenticating requests to the API server. This topic is covered in depth in Chapter 10.

You can also have the API server call out to a webhook to authorize actions that can be taken on resources by authenticated users. This is an uncommon implementation since Kubernetes has a capable Role-Based Access Control system built in. However, if you find this system inadequate for any reason you have this option available to you.

Admission control

Admission control is a particularly useful and widely used extension point. If in use, when a request is sent to the API server to perform an action, the API server calls any applicable admission webhooks according to the validating and mutating admission webhook configs. This topic is covered in Chapter 8.

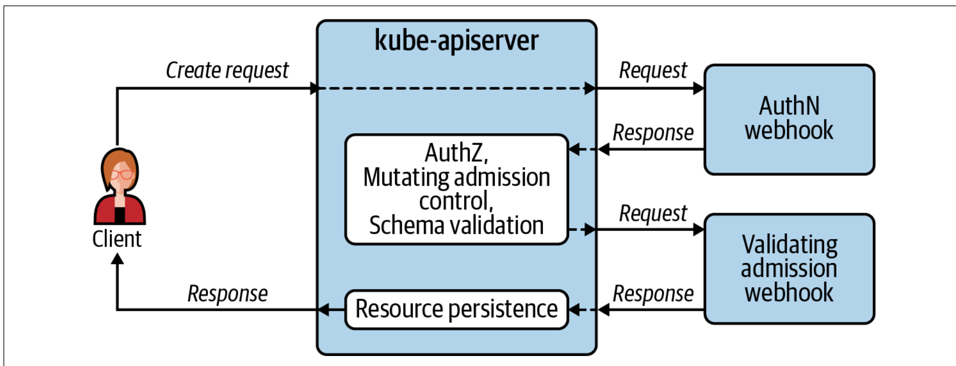


Figure 11-1. Webhook extensions are backend servers leveraged by the Kubernetes API server.

Operator Extensions

Operators are clients of, as opposed to backend webhooks for, the API server. As shown in Figure 11-2, software operators interact as clients of the Kubernetes API, just like human operators. These software operators are often called *Kubernetes Operators* and follow the officially documented operator pattern. Their primary purpose is to relieve toil from human operators and perform operations on their behalf. These operator extensions follow the same engineering principles as the core Kubernetes control plane components. When developing operator extensions as platform services, think of them as custom extensions of the control plane for your application platform.

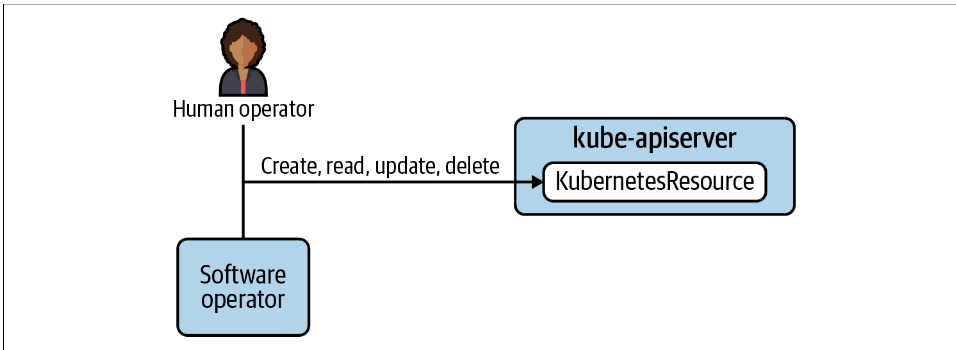


Figure 11-2. Operator extensions are clients of the Kubernetes API server.

The Operator Pattern

You could say that the operator pattern boils down to extending Kubernetes with Kubernetes. We create new Kubernetes resources and develop Kubernetes controllers to reconcile the state defined in them. We use Kubernetes resources called Custom Resource Definitions (CRDs) to define our new resources. These CRDs create new API types and tell the Kubernetes API how to validate these new objects. Then, we take the very same principles and designs that make Kubernetes controllers so effective and use those principles to build software extensions to the system. These are the two core mechanisms that we employ when building operators: custom resources and controllers.

The notion of an operator was introduced in November 2016 by Brandon Phillips, one of the founders of CoreOS. This early definition of an operator was that an operator was an app-specific controller that managed a complex stateful application. This is still a very useful definition but has broadened somewhat over the years to where the Kubernetes docs now classify any controller that uses CRDs as an operator. This more general definition is the one we will use as it applies to building platform services. Your platform services may not be "complex stateful applications" but still can benefit from using this powerful operator pattern.

The following section will look at Kubernetes controllers, which provide a model for the functionality we will use in our custom controllers. Then we will examine the custom resources that store the desired and existing state our controllers reconcile for us.

Kubernetes Controllers

Kubernetes' core features and functionality are provided by controllers. They watch resource types and respond to creation, mutation, and deletion of resources by fulfilling that desired state. For example, there is a controller that comes bundled with the kube-controller-manager that watches for ReplicaSet resource types. When a

ReplicaSet is created, the controller creates a number of identical Pods—as many as were defined by the `replicas` field in the ReplicaSet. Then at some point later, if you change that value, the controller will create or delete Pods to satisfy the new desired state.

This watch mechanism is central to the functionality of all Kubernetes controllers. It is an etcd feature that is exposed by the Kubernetes API server to the controllers that need to respond to changes in resources. The controllers hold a connection open with the API server, which allows the API server to notify the controller when a change has occurred to a resource it cares about or manages.

This enables very powerful behavior. The user can declare the desired state of the system by submitting resource manifests. The controllers responsible for fulfilling the desired state are notified and begin working to make the existing state match the declared, desired state. Furthermore, in addition to users submitting manifests, controllers can also do the same which, in turn, triggers operations in other controllers. In this way you end up with a system of controllers that can provide sophisticated functionality that is stable and reliable.

One important feature of these controllers is that if they cannot fulfill the desired state due to some impediment, they will continue to try on an infinite loop. The duration between attempts to fulfill desired state may increase over time so that undue load is not placed on the system, but try it will. This provides a self-healing behavior that is incredibly important in complex distributed systems such as this.

For example, the scheduler is responsible for assigning Pods to nodes in the cluster. The scheduler is just another Kubernetes controller, just with a particularly important and involved task. If there are insufficient compute resources available for one or more Pods, they will go into a "Pending" state and the scheduler will continue to attempt to schedule the Pod at some interval. As such, if compute resources free up or are added at any point, the Pod will be scheduled and run. So if another batch workload completes and resources free up, or if a cluster autoscaler adds some worker nodes, the Pending Pod will be assigned with no further action required from a human operator.

In following the operator pattern to build extensions to your application platform, it's essential to use these design principles used by Kubernetes controllers: (1) watch resources in the Kubernetes API to get notified when changes to their desired state occur, and (2) work to reconcile existing and desired state on a nonterminating loop.

Custom Resources

One of the most important features of the Kubernetes API is the ability to extend the resource types it will recognize. If you submit a valid CRD you will immediately have a new custom API type at your disposal. The CRD contains all the fields that you will

need in your custom resource both in the `spec` where you will provide the desired state for your resource and in the `status` where you can record important information about the observed, existing state.

Before diving further into this topic, let's briefly review Kubernetes resources. We are going to be talking a lot about resources in this chapter, so it's important to ensure we're crystal clear on this topic. When we talk about *resources* in Kubernetes, we are referring to the objects that are used to record state. An example of a common resource is the Pod resource. When you create a Pod manifest, you are defining the attributes of what will become a Pod resource. When you submit that to the API server with `kubectl apply -f pod.yaml` or similar, you are creating an instance of the Pod API type. On one hand you have the API type, or "kind," which refers to the definition and form of the object as provided in a CRD. On the other hand we have the resource that is an instantiation or instance of that kind. The Pod is an API type or kind. A Pod you create with the name "my-app" is a Kubernetes resource.

Unlike a relational database where relationships between objects are recorded and linked by foreign keys in the database itself, each object in the Kubernetes API exists independently. Relationships are established using labels and selectors, and it is the job of the controllers to manage relationships defined this way. You cannot query etcd for related objects the way you can with structured query language (SQL). So when we talk about resources, we're talking about actual instances of Namespaces, Pods, Deployments, Secrets, ConfigMaps, etc. When we talk about custom resources, we're referring to user-defined resources that have been added and defined with CRDs. When you create a CRD, you define new API types that allow you to create and manage custom resources as you would other core Kubernetes resources.

CRDs use the Open API v3 schema specification for defining fields. This allows for features like setting fields as optional or required as well as setting default values. This will provide validation instructions for the API server when it receives a request to create or update one of your custom resources. In addition, you can group your APIs for improved logical organization and, very importantly, version your API types as well.

To illustrate what a CRD is and what a manifest for the resulting custom resource looks like, let's look at a fictional example of a custom WebApp API type. In this example, a WebApp resource includes the desired state for a web application that consists of the following six Kubernetes resources:

Deployment
: A stateless application that provides a user interface for clients, processes requests, and stores data in a relational database

StatefulSet
: The relational database that provides the persistent data store for the web application

ConfigMap
: Contains the config file for the stateless application, which is mounted into each Pod of the Deployment

Secret
: Contains credentials for the application to connect to its database

Service
: Routes traffic to the Deployment's backend Pods

Ingress
: Contains routing rules for the Ingress controller to properly route client requests into the cluster

The creation of a WebApp resource would prompt a WebApp Operator to create these various child resources. These created resources would constitute a complete, functioning instance of the web application that serves clients that are end users and customers of the business.

Example 11-1 illustrates what a CRD might look like to define a new WebApp API type.

Example 11-1. WebApp CRD manifest

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: webapps.workloads.apps.acme.com ❶
spec:
  group: workloads.apps.acme.com
  names: ❷
    kind: WebApp
    listKind: WebAppList
    plural: webapps
    singular: webapp
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        description: WebApp is the Schema for the webapps API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of this
              representation of an object. Servers should convert recognized
```

```yaml
                schemas to the latest internal value, and may reject unrecognized
                values.'
              type: string
            kind:
              description: 'Kind is a string value representing the REST resource this
                object represents. Servers may infer this from the endpoint the client
                submits requests to. Cannot be updated. In CamelCase.'
              type: string
            metadata:
              type: object
            spec:
              description: WebAppSpec defines the desired state of WebApp
              properties:
                deploymentTier: ❸
                  enum:
                  - dev
                  - stg
                  - prod
                  type: string
                webAppHostname:
                  type: string
                webAppImage:
                  type: string
                webAppReplicas: ❹
                  default: 2
                  type: integer
              required: ❺
              - deploymentTier
              - webAppHostname
              - webAppImage
              type: object
            status:
              description: WebAppStatus defines the observed state of WebApp
              properties:
                created:
                  type: boolean
              type: object
          type: object
      served: true
      storage: true
```

❶ The name of the custom resource *definition*, as distinct from the name of the custom resource itself.

❷ The name of the custom resource, as distinct from the definition. This includes the variations of the name such as the plural version.

❸ The `deploymentTier` field must contain one of the values listed under `enum`. This validation will be carried out by the API server when it receives requests to create or update an instance of this custom resource.

❹ The `webAppReplicas` field includes a default value that will be applied if the field is not provided.

❺ The required fields are listed here. Note that `webAppReplicas` is not included and that it has a default value.

Now let's examine what a manifest for a WebApp would look like. Before submitting the manifest shown in Example 11-2 to the API server, you must first create the CRD shown in Example 11-1 so that Kubernetes has an API for it. Otherwise, it will not recognize what you are trying to create.

Example 11-2. An example of a manifest for a WebApp resource

```
apiVersion: workloads.apps.acme.com/v1alpha1
kind: WebApp
metadata:
  name: webapp-sample
spec:
  webAppReplicas: 2  ❶
  webAppImage: registry.acme.com/app/app:v1.4
  webAppHostname: app.acme.com
  deploymentTier: dev ❷
```

❶ This manifest specifies the default value for an optional field, which is unnecessary but fine to do if explicitness is desired.

❷ One of the permitted values for this field is used. Any value that is not permitted would prompt the API server to reject the request with an error.

When this WebApp manifest is submitted to the Kubernetes API, the WebApp Operator will be notified via its watch that a new instance of the WebApp kind has been created. It will fulfill the desired state expressed in the manifest by calling the API server to create the various child resources needed to spin up an instance of the web application.

While the custom resource model is powerful, do not overuse it. Do not use custom resources as the primary data store for an end-user application. Kubernetes is a container orchestration system. etcd should be storing the state of your software deployments, not the internal persistent data for your application. Doing so will put significant load on your cluster's control plane. Stick with relational databases, object stores, or whatever data store makes sense for your application. Leave the control plane to manage software deployments.

Operator Use Cases

When developing a Kubernetes-based platform, operators offer a compelling model for adding features to that platform. If you can represent the state of the system you need to implement in the fields of a custom resource, and if you can yield value from reconciling changes using a Kubernetes controller, an operator is often a great option.

In addition to platform features, operators may be used to facilitate the management of software deployments on your platform. They can provide the convenience of a generalized abstraction or can be custom-built for the needs of a particular sophisticated application. In either case, using software to manage software deployments is very useful.

In this section we're going to discuss the three general operator categories that you may consider using with your platform:

- Platform utilities
- General-purpose workload operators
- App-specific operators

Platform Utilities

Operators can be extremely useful in developing your platform. They allow you to add features to your cluster and build out functionality on top of Kubernetes in a way that leverages and integrates with the control plane seamlessly. There is a wealth of open source projects that utilize operators to provide platform services atop Kubernetes. These projects are already available and don't require you develop them. The reason we bring them up in a chapter about building them is that they help build a good mental model for how they work. Should you find yourself having to develop custom platform utilities, looking at existing successful projects will be helpful:

- The Prometheus Operator allows you to provide metrics collection, storage, and alerting platform services on your platform. In Chapter 9 we delve into the value that can be derived from this project.
- cert-manager provides certificate management as a service functionality. It removes significant toil and potential for downtime by offering x509 certificate creation and renewal services.
- Rook is storage operator that integrates with providers like Ceph to manage block, object, and filesystem storage as a service.

These open source solutions are examples of what is available in the community. There are also countless vendors that can provide and support similar platform utilities. However, when a solution is not available or not a good fit, enterprises sometimes build their own custom platform utilities.

A common example of a custom platform utility we see in the field is a Namespace operator. We have found it quite common for organizations to have a standard set of resources that are created with each Namespace, such as ResourceQuotas, LimitRanges, and Roles. And it has been a useful pattern to use a controller to take care of the routine tedium of creating these resources for each Namespace. In a later section, we will use this Namespace operator idea as an example to illustrate some implementation details when building operators.

General-Purpose Workload Operators

Application developers' core competency and concern is to add stability and features to the software they build. It is not writing YAML for deployment to Kubernetes. Learning how to properly define resource limits and requests, learning how to mount a ConfigMap volume or Secret as an environment variable, learning how to use label selectors to associate Services with Pods—none of these things add features or stability to their software.

In an organization that has developed common patterns for deployed workloads, the model of abstracting the complexity in a general-purpose manner has considerable promise. This is especially relevant in organizations that have embraced microservice architectures. In these environments there may be a considerable number of distinct pieces of software that are deployed with different functionality, but with very similar patterns of deployment.

For example, if your company has a large number of workloads that consist of a Deployment, Service, and Ingress resource, there are likely patterns that can be encoded into an operator that can abstract much of the resource manifests for these objects. In each case the Service references labels on the Pods of a Deployment. In each case the Ingress references the Service name. All these things can easily be handled by an operator—getting these details right is the definition of toil.

App-Specific Operators

This type of operator hits at the core of what a Kubernetes operator is: custom resources in conjunction with a custom Kubernetes controller for managing a complex stateful application. They are purpose-built to manage a particular application. Popular examples of this model are the various database operators in the community. We have operators for Cassandra, Elasticsearch, MySQL, MariaDB, PostgreSQL, MongoDB and many more. Commonly, they handle initial deployment as well as day 2 management concerns such as configuration updates, backups, and upgrades.

Operators for popular community- or vendor-supported projects have gained in popularity over the past few years. The area where this is approach is still in its infancy is in internal enterprise applications. In cases where your organization internally develops and maintains a sophisticated stateful application, an app-specific operator may be beneficial. For example, if your company maintains something like an ecommerce website, transaction processing app, or inventory management system that delivers critical business functionality, you may want to consider this option. There is tremendous opportunity for reducing human toil in the deployment and day 2 management of these kinds of workloads, especially when they are deployed widely and updated frequently.

This isn't to say that these app-specific operators are the universally right choice for managing your workloads. For simpler use cases, they are likely to be severe overkill. Production-ready operators are not trivial to develop, so weigh the trade-offs. How much time do you spend in routine toil managing deployment and day 2 concerns for an application? Is the engineering cost of building an operator likely to be less than that routine toil over the long term? Could simpler, existing tools such as Helm or Kustomize provide enough automation to sufficiently alleviate toil?

Developing Operators

Taking on the task of developing Kubernetes operators is not trivial, especially if tackling a full-featured, app-specific operator. The engineering investment to get one of these more involved projects into production can be considerable. As with other types of software development, if getting started in this area, begin with less involved projects while you become familiar with useful patterns and successful strategies. In this section we'll discuss some tools and design strategies that will help make developing operators a more efficient and successful endeavor.

We will cover some specific projects you can leverage to help in development of such tools. And then we'll break down the process of designing and implementing this kind of software in detail. We'll include some code snippets to illustrate the concepts and best practices.

Operator Development Tooling

If you have a compelling use case for a custom Kubernetes operator, there are a couple of community projects that can be very helpful in your endeavor. If you are—or have on staff—an experienced Go programmer that is familiar with the Kubernetes client-go library and with developing Kubernetes operators, you can certainly write your operators from scratch. However, there are common components to every operator, and using tools to generate boilerplate source code and utilities are expediencies that even seasoned operator developers commonly use. They just save time. Software development kits (SDKs) and frameworks can be helpful when they fit the pattern of

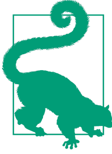
software you're developing. However, they can be a nuisance if they make assumptions that don't suit your purpose. If your project fits the standard model of using one or more custom resources to define configuration, and custom controllers to implement behavior associated with these objects, it is likely the tools we discuss here will be useful.

Kubebuilder

Kubebuilder can be described as an SDK for building Kubernetes APIs. This is an apt description but is not exactly what you might expect. Using kubebuilder begins with the command-line tool that you use to generate boilerplate. It stamps out the source code, a Dockerfile, a Makefile, sample Kubernetes manifests—all the things you need to write for every such project. As such, it saves a ton of time in getting a project started.

Kubebuilder also leverages a collection of tools in a related project called controller-runtime. The required imports and common implementations are included in the source code generated by the CLI. These help with much of the routine heavy lifting of running a controller and interacting with the Kubernetes API. It helps with setting up shared caches and clients to provide efficient interaction with the API server. The cache allows your controller to list and get objects without new requests to the API server for each query, which eases load on the API server and speeds up reconciliation. Controller-runtime also provides mechanisms for triggering reconcile requests in response to events such as resource changes. These reconciliations will be triggered by default for the parent custom resource. They can—and usually should—also be triggered when changes occur to child resources created by the controller and functions are available to do so. If running your controller in highly available (HA) mode, controller-runtime provides the opportunity to enable leader election to ensure just one controller is active at any given time. Furthermore, controller-runtime includes a package to implement webhooks, which are often used for admission control. Lastly, the library includes facilities to write structured logs and expose Prometheus metrics for observability.

Kubebuilder is a great choice if you are a Go programmer with Kubernetes experience. It is even a good choice if you are an experienced software developer but are new to the Go programming language. But it is exclusively for Go—it doesn't cater to other languages.



If you are going to be developing tools for Kubernetes, you should strongly consider learning Go if you don't know it already. You can certainly use other languages. Kubernetes offers a REST API, after all. And there are officially supported client libraries for Python, Java, C#, JavaScript, and Haskell, not to mention many other community-supported libraries. If you have important reasons for using these, you can certainly be successful. However, Kubernetes itself is written in Go, and the ecosystem for that language in the world of Kubernetes is rich and well-supported.

One of the features of Kubebuilder that make it such a time-saver is its generation of CRDs. Writing a CRD manifest by hand is no joke. The OpenAPI v3 spec that is used to define these custom APIs is pretty detailed and involved. The Kubebuilder CLI will generate the files where you will define the fields for your custom API types. You add the various fields to the struct definitions and tag them with special markers that provide metadata such as default values. Then you can use a make target to generate the CRD manifests. It is very handy.

On the subject of make targets, in addition to generating CRDs, you can generate the RBAC and sample custom resource manifests, install the custom resources in your development cluster, build and publish images for your operator, and run your controller locally against a cluster during development. Having conveniences for all these tedious, time-consuming tasks really is a productivity boost, especially early in the project.

For these reasons, we prefer and recommend Kubebuilder for building operators. It has been adopted and used with success in a variety of projects.

Metacontroller

If your comfort with a particular programming language besides Go compels you to stick with it, another useful option to aid in developing operators is Metacontroller. This is an entirely different approach to developing and deploying operators, but it is one that is worth considering if you want to use a variety of languages and expect to deploy a number of custom in-house operators with your platform. Engineers experienced in programming Kubernetes will also sometimes use Metacontroller for prototyping and then use Kubebuilder for the final project once design and implementation details have been established. And this alludes to one of the strengths of Metacontroller: once you have the Metacontroller add-on installed in your cluster, it is fast to get going.

That is essentially what Metacontroller is: a cluster add-on that abstracts away the interaction with the Kubernetes API. Your job is to write the controller webhook that contains your controller's logic. Metacontroller calls this the *lambda controller*. Your lambda controller makes decisions about what to do with the resources it cares about.

Metacontroller watches the resources under management and alerts your controller with an HTTP call when there is a change it needs to make a decision about. Metacontroller itself uses custom resources that define the characteristics of your webhook, e.g., its URL and the resources it manages. As such, once Metacontroller is running in your cluster, adding a controller consists of deploying your lambda controller webhook and adding a Metacontroller custom resource; e.g., composite controller resource. And all your new controller need do is expose an endpoint that will accept requests from Metacontroller, parse JSON payloads that contain the Kubernetes resource object in question, and then return a response to Metacontroller with any changes to be sent to the Kubernetes API. Figure 11-3 illustrates how these components interact when using Metacontroller.

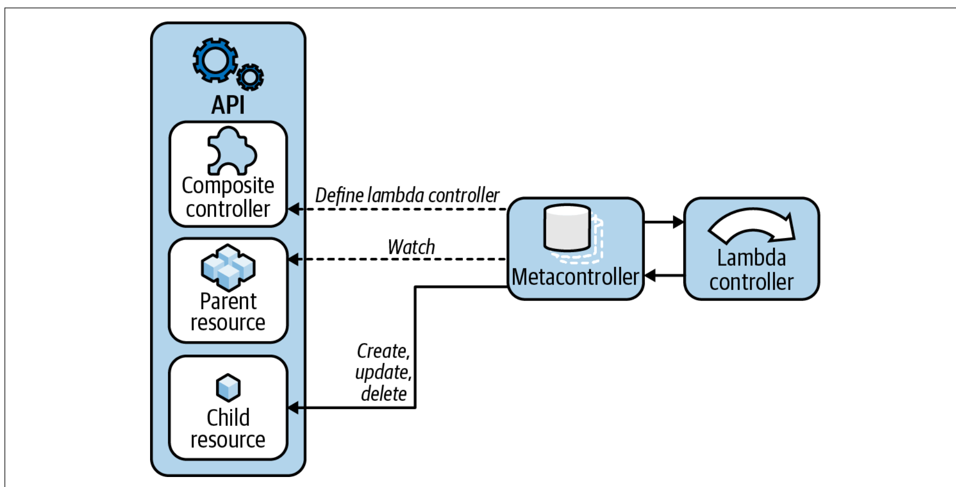


Figure 11-3. Metacontroller abstracts the Kubernetes API for your lambda controller.

What Metacontroller does not help with is the creation of any CRDs you may need to add to your cluster. You are on your own there. If you are writing a controller that responds to changes in core Kubernetes resources, this won't be an issue. But if you're developing custom resources, this is an area where Kubebuilder has a significant advantage.

Operator Framework

The Operator Framework is a collection of open source tools that originated at Red Hat and are now under the CNCF umbrella as an incubating project. This framework facilitates the development of operators. It includes the Operator SDK, which offers similar functionality to Kubebuilder when developing operators using Go. Like Kubebuilder, it provides a CLI to generate boilerplate for projects. Also like Kubebuilder, it uses the controller-runtime tools to help integrate with the Kubernetes API. In addition to Go projects, the Operator SDK allows developers to use Helm or Ansible to

manage operations. The framework also includes the Operator Lifecycle Manager, which is what it sounds like: an operator for your operators. It provides abstractions for installing and upgrading your operators. The project also maintains an operator hub, which offers a way for users to discover operators for software they use. We have not encountered platform teams using these tools in the field. Being a Red Hat-maintained project, it is likely more common among users of OpenShift, a Red Hat Kubernetes-based offering.

Data Model Design

Just as you might begin the design of a web application by defining the database schema the app will use to persist data, a great place to start when building an operator is the data model for the custom resources your operator will use. In fact, you will probably have some idea of what fields your custom resource will need in its spec before you get started. As soon as you recognize a problem to solve or gap to fill, the attributes of the object that defines desired state will begin to take shape.

In the example from earlier in this chapter, the Namespace operator, it could begin as an operator that will create a variety of resources: LimitRange, ResourceQuota, Roles, and NetworkPolicies to go along with a new Namespace for an app dev team. You may want to bind a team lead to the `namespace-admin` Role right away and then hand off management of the Namespace to that person. This would naturally lead you to add an `adminUsername` field to the spec of the custom resource. The custom resource manifest might look something like Example 11-3.

Example 11-3. An example of a manifest for an AcmeNamespace resource

```
apiVersion: tenancy.acme.com/v1alapha1
kind: AcmeNamespace
metadata:
  name: team-x
spec:
  namespaceName: app-y  ❶
  adminUsername: sam  ❷
```

❶ An arbitrary name for the Namespace—in this case it will host a workload, “app-y.”

❷ This username would correspond to that which is used by the company’s identity provider, usually an Active Directory system or similar.

Submitting the manifest in Example 11-3 would result in the username `sam` being added to the subjects of a RoleBinding to the `namespace-admin` Role in the manner shown in Example 11-4.

Example 11-4. Example of a Role and Rolebinding created for the `team-x` AcmeNamespace

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: namespace-admin
  namespace: app-y
rules:
- apiGroups:
  - "*"
  resources:
  - "*"
  verbs:
  - "*"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: namespace-admin
  namespace: app-y
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: namespace-admin
subjects:
- kind: User
  name: sam  ❶
  namespace: app-y
```

❶ The `adminUsername` provided in the AcmeNamespace manifest would get inserted here to be bound to the `namespace-admin` Role.

When thinking about the behavior you want, having Sam bound to the `namespace-admin` Role, the data needed to accomplish this becomes pretty clear: Sam's username and the name of the Namespace. So start with the obvious pieces of data that you will need to provide functionality and define fields in the spec for your CRD from that. What that would look like as part of a Kubebuilder project would be similar to what is shown in Example 11-5.

Example 11-5. Type definition for the `AcmeNamespaceSpec`

```go
// api/v1alpha1/acmenamespace_types.go

...

// AcmeNamespaceSpec defines the desired state of AcmeNamespace
type AcmeNamespaceSpec struct {
```

```
    // The name of the namespace
    NamespaceName string `json:"namespaceName"`

    // The username for the namespace admin
    AdminUsername string `json:"adminUsername"`
...
```

This is the source code from which Kubebuilder will generate your CRD manifest and the sample AcmeNamespace manifest to use for testing and demonstration.

Now that we have a data model that we *think* will allow us to adequately manage state for the behavior we want, it's time to start writing the controller. It's probable we will find our data model inadequate as we develop and find there are additional fields that will be necessary to bring about our desired outcomes. But this is a useful place to start for now.

Logic Implementation

The logic is implemented in our controller. The primary job of the controller is to manage one or more custom resources. The controller will keep a watch on those custom resources under management. This part is trivial to implement when using tools like Kubebuilder and Metacontroller. It is pretty straightforward even if just using the client-go library, the GitHub repo for which has excellent code examples to refer to. With a watch on its custom resource/s, your controller will be notified of any changes to resources of this type. The job of your controller at this point is as follows:

- Gather an accurate picture of the existing state of the system
- Examine the desired state of the system
- Take the required actions to reconcile the existing state with desired state

Existing state

There are essentially three places your controller may gather existing state information from:

- The `status` of your custom resource
- Other related resources in the cluster
- Relevant conditions outside the cluster or in other systems

The `status` field provides a place for controllers in Kubernetes to record observed, existing state. For example, Kubernetes uses a `status.phase` field on some resources, such as Pods and Namespaces, to keep track of whether the resource is `Running` (for Pods) or `Active` (for Namespaces).

Let's go back to the example of a Namespace operator. The controller is notified of a new AcmeNamespace resource along with the spec for it. The controller cannot assume it is a new resource and just robotically create the child Namespace and Role resources. What if it's a preexisting resource that has simply been updated with some change? Attempting to create the child resources again will get an error from the Kubernetes API. However, to follow the preceding Kubernetes example, if we include a `phase` field in the `status` of our CRD, the controller can check it to evaluate existing state. When it is first created, the controller will find the `status.phase` field empty. This will tell the controller it is a new resource creation, and should go ahead with creating all child resources. Once all children are created with successful responses from the API, the controller can populate the `status.phase` field with a value of `Created`. Then if the AcmeNamespace resource is later changed, when the controller is notified, it can see from this field that it has been previously created and move on to other reconciliation steps.

The use of the `status.phase` field to determine existing state as described so far has one critical flaw. It assumes the controller itself will never fail. What if a problem is encountered while creating the child resources? Say, for example, the controller gets notified of a new AcmeNamespace, creates the child Namespace, but then goes down before it can create the associated Role resources. When the controller comes back up, it will find the AcmeNamespace resource *without* `Created` in the `status.phase` field, attempt to create the child Namespace, and fail without a satisfactory way to reconcile the situation. In order to prevent this, the controller can add a `CreationInProgress` value to the `status.phase` as the very first step when it finds a new AcmeNamespace has been created. This way, if that failure during creation occurs, when the controller comes back up and sees the `CreationInProgress` phase, it will know that existing state cannot be accurately determined from the `status` alone. This is where it will need to look to other related resources in the cluster to determine existing state.

When existing state cannot be ascertained from the AcmeNamespace `status`, it can query the API server—or preferably the local cache of objects in the API server—for conditions it cares about. If it finds the phase of an AcmeNamespace set to `CreationInProgress` it can start querying the API server for the existence of child resources it expects to be there. In the failure example we're using, it would query for a child Namespace, find it exists, and move on. It would query for the Role resource, find it *doesn't* exist, and proceed with creating those resources. In this manner our controller can be tolerant of failure. And we should always assume failure will occur and develop controller logic accordingly.

Furthermore, sometimes our controller will be interested in existing state outside the cluster. Cloud infrastructure controllers are a good example of this. The status of infrastructure systems must be queried from cloud provider APIs outside the cluster.

What this existing state may be will be highly dependent on the purpose of the operator in question and will usually be clear.

Desired state

The desired state for a system is expressed in the `spec` for the relevant resources. In our Namespace operator, the desired state provided by the `namespaceName` informs the controller what the `metadata.name` field should be for the resulting Namespace. The `adminUsername` field determines what the `namespace-admin` RoleBinding's `subjects[0].name` should be. These are examples of direct mappings of desired state to fields in child resources. Often, the implementation is less direct.

We saw an example of this with the use of the `deploymentTier` field in the `AcmeStore` example earlier in this chapter. It allowed a user to specify a single variable that informed the controller logic on what default values to use. We can apply a very similar idea to the Namespace operator. Our new, modified AcmeNamespace manifest could look like Example 11-6.

Example 11-6. The AcmeNamespace manifest with new fields added

```
apiVersion: tenancy.acme.com/v1alapha1
kind: AcmeNamespace
metadata:
  name: team-x
spec:
  namespaceName: app-y
  adminUsername: sam
  deploymentTier: dev  ❶
```

❶ New addition to the data model for the AcmeNamespace API type.

This will prompt the controller to create a ResourceQuota that could look like Example 11-7.

Example 11-7. The ResourceQuota created for the `team-x` AcmeNamespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev
spec:
  hard:
    cpu: "5"
    memory: 10Gi
    pods: "10"
```

Whereas the default ResourceQuota for `deploymentTier: prod` may look like Example 11-8.

Example 11-8. An alternative ResourceQuota created when `deploymentTier: prod` is given in the AcmeNamespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: prod
spec:
  hard:
    cpu: "500"
    memory: 200Gi
    pods: "100"
```

Reconciliation

In Kubernetes, reconciliation is the process of altering the existing state to match the desired state. This can be as simple as the kubelet requesting the container runtime stop containers associated with a deleted Pod. Or it can be more complex, such as an operator creating an array of new resources in response to a custom resource that represents a stateful application. These are examples of reconciliation triggered by creating or deleting the resources that express desired state. But very often the reconciliation involves a response to a mutation.

A simple mutation example is if you update the number of replicas on a Deployment resource from 5 to 10. The existing state is 5 Pods for a workload. The desired state is 10 Pods. The reconciliation performed by the Deployment controller in this case involves updating the replicas on the relevant ReplicaSet. The ReplicaSet controller then reconciles state by creating 5 new Pod resources which are, in turn, scheduled by the scheduler, which prompts the applicable kubelets to request new containers from the container runtime.

Another mutation example that is a little more involved is if you change the image in a Deployment spec. This is usually to update the version of a running application. By default, the Deployment controller will perform a rolling update as it reconciles state. It will create a *new* ReplicaSet for the new version of the app, increment the replicas on the new ReplicaSet, and decrement the replicas on the old ReplicaSet so that the Pods are replaced one at a time. Once all new image versions are running with the desired number of replicas, reconciliation is complete.

What reconciliation looks like for a custom controller managing a custom resource is going to vary greatly according to what the custom resource represents. But one thing that should remain constant is that if reconciliation is not successful due to a condition that is beyond the domain of the controller, it should retry indefinitely. Generally

speaking, the reconciliation loop should implement increasing delays between iterations. For example, if it is reasonable to expect other systems in the cluster to be actively reconciling state that is preventing a controller from completing its operations, you may retry 1 second later. However, for the sake of preventing gratuitous resource consumption, it is recommended to exponentially increase the delay between each iteration until it reaches some reasonable limit of, say, 5 minutes. At that point, the controller will retry reconciliation once every 5 minutes. This allows for unattended resolving of systems while limiting resource consumption and network traffic in circumstances that aren't resolving quickly.

Implementation details

In broad terms, to implement initial controller functionality for the Namespace operator, we want to be able to:

- Write or generate a concise AcmeNamespace manifest like the earlier example
- Submit the manifest to the Kubernetes API
- Have a controller respond by creating a Namespace, ResourceQuota, LimitRange, Roles, and a RoleBinding.

In a kubebuilder project, the logic to create these resources will live in a `Reconcile` method. The initial implementation of creating a Namespace with the controller could look like Example 11-9.

Example 11-9. The `Reconcile` method for the AcmeNamespace controller

```go
// controllers/acmenamespace_controller.go

package controllers

import (
	"context"

	"github.com/go-logr/logr"
	corev1 "k8s.io/api/core/v1"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
	"k8s.io/apimachinery/pkg/runtime"
	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"

	tenancyv1alpha1 "github.com/lander2k2/namespace-operator/api/v1alpha1"
)

...

func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
	ctx := context.Background()
```

```
	log := r.Log.WithValues("acmenamespace", req.NamespacedName)

	var acmeNs tenancyv1alpha1.AcmeNamespace ❶
	r.Get(ctx, req.NamespacedName, &acmeNs) ❷

	nsName := acmeNs.Spec.NamespaceName
	adminUsername := acmeNs.Spec.AdminUsername

	ns := &corev1.Namespace{ ❸
		ObjectMeta: metav1.ObjectMeta{
			Name: nsName,
			Labels: map[string]string{
				"admin": adminUsername,
			},
		},
	}

	if err := r.Create(ctx, ns); err != nil { ❹
		log.Error(err, "unable to create namespace")
		return ctrl.Result{}, err
	}

	return ctrl.Result{}, nil
}
...
```

❶ The variable that will represent the AcmeNamespace object that has been created, updated, or deleted.

❷ Fetching the content of the AcmeNamespace object from the request. Error catching omitted for brevity.

❸ Creating the new Namespace object.

❹ Creating the new Namespace resource in the Kubernetes API.

This simplified snippet demonstrates the controller creating the new Namespace. Adding the Role and RoleBinding for the Namespace admin to the controller would look similar, as shown in Example 11-10.

Example 11-10. The creation of the Role and RoleBinding by the AcmeNamespace controller

```
// controllers/acmenamespace_controller.go
...
	role := &rbacv1.Role{
		ObjectMeta: metav1.ObjectMeta{
			Name:      "namespace-admin",
			Namespace: nsName,
```

```
		},
		Rules: []rbacv1.PolicyRule{
			{
				APIGroups: []string{"*"},
				Resources: []string{"*"},
				Verbs:     []string{"*"},
			},
		},
	}

	if err := r.Create(ctx, role); err != nil {
		log.Error(err, "unable to create namespace-admin role")
		return ctrl.Result{}, err
	}

	binding := &rbacv1.RoleBinding{
		ObjectMeta: metav1.ObjectMeta{
			Name:      "namespace-admin",
			Namespace: nsName,
		},
		RoleRef: rbacv1.RoleRef{
			APIGroup: "rbac.authorization.k8s.io",
			Kind:     "Role",
			Name:     "namespace-admin",
		},
		Subjects: []rbacv1.Subject{
			{
				Kind:      "User",
				Name:      adminUsername,
				Namespace: nsName,
			},
		},
	}

	if err := r.Create(ctx, binding); err != nil {
		log.Error(err, "unable to create namespace-admin role binding")
		return ctrl.Result{}, err
	}

	return ctrl.Result{}, nil
}
...
```

At this point we are able to submit an AcmeNamespace manifest to the API and our Namespace operator will create the Namespace, a Role for the Namespace admin, and a RoleBinding to the username we provided. As we discussed earlier, this will work fine when we create a new AcmeNamespace but will break when it tries to reconcile it at any other time in the future. This would occur if the AcmeNamespace was changed in any way. It would also happen if the controller restarted for any reason. When the controller restarts, it must re-list and reconcile all existing resources in case

something changed. So at this point, simply restarting our controller will break it. Let's fix that by adding a simple use of the status field. First, Example 11-11 shows the addition of the field to `AcmeNamespaceStatus`.

Example 11-11. Adding a field to the status of the AcmeNamespace

```
// api/v1alpha1/acmenamespace_types.go

// AcmeNamespaceStatus defines the observed state of AcmeNamespace
type AcmeNamespaceStatus struct {

        // Tracks the phase of the AcmeNamespace
        // +optional
        // +kubebuilder:validation:Enum=CreationInProgress;Created
        Phase string `json:"phase"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
...
```

Now we can leverage this field in our controller as shown in Example 11-12.

Example 11-12. Using the new status field in the AcmeNamespace controller

```
// controllers/acmenamespace_controller.go
...

const (
        statusCreated    = "Created"
        statusInProgress = "CreationInProgress"
)

...

func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
        ...

        switch acmeNs.Status.Phase {
        case statusCreated:
                // do nothing
                log.Info("AcmeNamespace child resources have been created")
        case statusInProgress:
                // TODO: query and create as needed
                log.Info("AcmeNamespace child resource creation in progress")
        default:
                log.Info("AcmeNamespace child resources not created")

                // set status to statusInProgress
                acmeNs.Status.Phase = statusInProgress
```

```
        if err := r.Status().Update(ctx, &acmeNs); err != nil {
                log.Error(err, "unable to update AcmeNamespace status")
                return ctrl.Result{}, err
        }

        // create namespace, role and role binding
        ...

        // set status to statusCreated
        acmeNs.Status.Phase = statusCreated
        if err := r.Status().Update(ctx, &acmeNs); err != nil {
                log.Error(err, "unable to update AcmeNamespace status")
                return ctrl.Result{}, err
        }
    }

    return ctrl.Result{}, nil
}

...
```

Now we have a controller that can restart safely. It also now has the beginnings of a system to examine existing state using the custom resource's status and to carry out reconciliation steps based on that existing state.

One other thing we should usually do is set ownership for the child resources. If we set the AcmeNamespace resource as the owner of the Namespace, Role, and RoleBinding, that allows us to delete all the children by simply deleting the owner AcmeNamespace resource. This ownership will be managed by the API server. Even if the controller is not running, if the owner AcmeNamespace resource is deleted, the children will be deleted, too.

This raises the question of scoping for our AcmeNamespace API type. When using Kubebuilder, it will default to Namespaced scoping. However, a Namespace-scoped API type cannot be an owner of a cluster-scoped resource, such as a Namespace. With Kubebuilder we can use convenient markers to generate the CRD manifest with the proper scoping for this usage, as shown in Example 11-13.

Example 11-13. Updated API definition in a Kubebuilder project

```
// api/v1alpha1/acmenamespace_types.go
package v1alpha1

import (
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
```

```go
// NOTE: json tags are required.  Any new fields you add must have json tags
// for the fields to be serialized.

// AcmeNamespaceSpec defines the desired state of AcmeNamespace
type AcmeNamespaceSpec struct {

	// The name of the namespace
	NamespaceName string `json:"namespaceName"`

	// The username for the namespace admin
	AdminUsername string `json:"adminUsername"`
}

// AcmeNamespaceStatus defines the observed state of AcmeNamespace
type AcmeNamespaceStatus struct {

	// Tracks the phase of the AcmeNamespace
	// +optional
	// +kubebuilder:validation:Enum=CreationInProgress;Created
	Phase string `json:"phase"`
}

// +kubebuilder:resource:scope=Cluster ❶
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

// AcmeNamespace is the Schema for the acmenamespaces API
type AcmeNamespace struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

	Spec   AcmeNamespaceSpec   `json:"spec,omitempty"`
	Status AcmeNamespaceStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true

// AcmeNamespaceList contains a list of AcmeNamespace
type AcmeNamespaceList struct {
	metav1.TypeMeta `json:",inline"`
	metav1.ListMeta `json:"metadata,omitempty"`
	Items           []AcmeNamespace `json:"items"`
}

func init() {
	SchemeBuilder.Register(&AcmeNamespace{}, &AcmeNamespaceList{})
```

❶ This marker will set the correct scoping on the CRD when the manifest is generated with `make manifests`.

This will generate a CRD that looks like Example 11-14.

Example 11-14. Cluster scoped CRD for AcmeNamespace API type

```
---
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: (devel)
  creationTimestamp: null
  name: acmenamespaces.tenancy.acme.com
spec:
  group: tenancy.acme.com
  names:
    kind: AcmeNamespace
    listKind: AcmeNamespaceList
    plural: acmenamespaces
    singular: acmenamespace
  scope: Cluster  ❶
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      description: AcmeNamespace is the Schema for the acmenamespaces API
      properties:
        apiVersion:
          description: 'APIVersion defines the versioned schema of this
            representation of an object. Servers should convert recognized
            schemas to the latest internal value, and may reject unrecognized
            values.'
          type: string
        kind:
          description: 'Kind is a string value representing the REST resource this
            object represents. Servers may infer this from the endpoint the client
            submits requests to. Cannot be updated. In CamelCase.'
          type: string
        metadata:
          type: object
        spec:
          description: AcmeNamespaceSpec defines the desired state of AcmeNamespace
          properties:
            adminUsername:
              description: The username for the namespace admin
              type: string
            namespaceName:
              description: The name of the namespace
              type: string
          required:
          - adminUsername
          - namespaceName
          type: object
        status:
          description: 'AcmeNamespaceStatus defines the observed state of
```

```
                            AcmeNamespace'
              properties:
                phase:
                  description: Tracks the phase of the AcmeNamespace
                  enum:
                  - CreationInProgress
                  - Created
                  type: string
              type: object
          type: object
      version: v1alpha1
      versions:
      - name: v1alpha1
        served: true
        storage: true
    status:
      acceptedNames:
        kind: ""
        plural: ""
      conditions: []
      storedVersions: []
```

❶ Resource scoping properly set.

Now we can set the AcmeNamespace as an owner for all child resources. This will introduce an `ownerReferences` field into the `metadata` for each child resource. At this point our `Reconcile` method looks like Example 11-15.

Example 11-15. Setting ownership for child resources of the AcmeNamespace

```
func (r *AcmeNamespaceReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
        ctx := context.Background()
        log := r.Log.WithValues("acmenamespace", req.NamespacedName)

        var acmeNs tenancyv1alpha1.AcmeNamespace
        if err := r.Get(ctx, req.NamespacedName, &acmeNs); err != nil {
                if apierrs.IsNotFound(err) { ❶
                        log.Info("resource deleted")
                        return ctrl.Result{}, nil
                } else {
                        return ctrl.Result{}, err
                }
        }

        nsName := acmeNs.Spec.NamespaceName
        adminUsername := acmeNs.Spec.AdminUsername

        switch acmeNs.Status.Phase {
        case statusCreated:
                // do nothing
                log.Info("AcmeNamespace child resources have been created")
```

```
case statusInProgress:
        // TODO: query and create as needed
        log.Info("AcmeNamespace child resource creation in progress")
default:
        log.Info("AcmeNamespace child resources not created")

        // set status to statusInProgress
        acmeNs.Status.Phase = statusInProgress
        if err := r.Status().Update(ctx, &acmeNs); err != nil {
                log.Error(err, "unable to update AcmeNamespace status")
                return ctrl.Result{}, err
        }

        ns := &corev1.Namespace{
                ObjectMeta: metav1.ObjectMeta{
                        Name: nsName,
                        Labels: map[string]string{
                                "admin": adminUsername,
                        },
                },
        }

        // set owner reference for the namespace ❷
        err := ctrl.SetControllerReference(&acmeNs, ns, r.Scheme)
if err != nil {
                log.Error(err, "unable to set owner reference on namespace")
                return ctrl.Result{}, err
        }

        if err := r.Create(ctx, ns); err != nil {
                log.Error(err, "unable to create namespace")
                return ctrl.Result{}, err
        }

        role := &rbacv1.Role{
                ObjectMeta: metav1.ObjectMeta{
                        Name:      "namespace-admin",
                        Namespace: nsName,
                },
                Rules: []rbacv1.PolicyRule{
                        {
                                APIGroups: []string{"*"},
                                Resources: []string{"*"},
                                Verbs:     []string{"*"},
                        },
                },
        }

        // set owner reference for the role ❸
        err = ctrl.SetControllerReference(&acmeNs, role, r.Scheme)
if err != nil {
                log.Error(err, "unable to set owner reference on role")
```

```
                        return ctrl.Result{}, err
                }

                if err := r.Create(ctx, role); err != nil {
                        log.Error(err, "unable to create namespace-admin role")
                        return ctrl.Result{}, err
                }

                binding := &rbacv1.RoleBinding{
                        ObjectMeta: metav1.ObjectMeta{
                                Name:      "namespace-admin",
                                Namespace: nsName,
                        },
                        RoleRef: rbacv1.RoleRef{
                                APIGroup: "rbac.authorization.k8s.io",
                                Kind:     "Role",
                                Name:     "namespace-admin",
                        },
                        Subjects: []rbacv1.Subject{
                                {
                                        Kind:      "User",
                                        Name:      adminUsername,
                                        Namespace: nsName,
                                },
                        },
                }

                // set owner reference for the role binding ❹
                err = ctrl.SetControllerReference(&acmeNs, binding, r.Scheme);
        if err != nil {
                        log.Error(err, "unable to set reference on role binding")
                        return ctrl.Result{}, err
                }

                if err := r.Create(ctx, binding); err != nil {
                        log.Error(err, "unable to create role binding")
                        return ctrl.Result{}, err
                }

                // set status to statusCreated
                acmeNs.Status.Phase = statusCreated
                if err := r.Status().Update(ctx, &acmeNs); err != nil {
                        log.Error(err, "unable to update AcmeNamespace status")
                        return ctrl.Result{}, err
                }
        }

        return ctrl.Result{}, nil
}
...
```

❶ Check to see if resource was not found so we don't try to reconcile when the AcmeNamespace has been deleted.

❷ Set owner reference on Namespace.

❸ Set owner reference on Role.

❹ Set owner reference on RoleBinding.

Notice that we had to add the error checking to see if the AcmeNamespace resource was found. This is because when it is deleted, normal reconciliation will fail due to there no longer being desired state to reconcile. And, in this case, we put an owner reference on the child resources so the API server takes care of reconciling state for deletion events.

This illustrates the point that reconciliation must not make assumptions about existing state. Reconciliation is triggered when:

- The controller starts or restarts
- A resource is created
- A change is made to a resource, including changes made by the controller itself
- A resource is deleted
- A periodic resync with the API is carried out to ensure an accurate view of the system

To this end, make sure your reconciliation doesn't make assumptions about the event that triggered reconciliation. Use the `status` field, determine relevant conditions in other resources as needed, and reconcile accordingly.

Admission webhooks

If you find your custom resource requires defaults or validation that cannot be implemented using the OpenAPI v3 spec in the CRD that creates your new API type, you can turn to validating and mutating admission webhooks. The Kubebuilder CLI has a `create webhook` command that caters specifically to these use cases by generating boilerplate to get you going faster.

An example of where a validating webhook may be useful with our Namespace operator example and its AcmeNamespace resource, is in validating the `adminUsername` field. As a convenience, your webhook could call out to your corporate identity provider to ensure the username provided is valid, preventing mistakes that require human intervention to correct.

A defaulting example could be to default the `deploymentTier` to the most common, least expensive `dev` option. This is particularly useful for maintaining backward compatibility with existing resource definitions when you make a change that adds new fields to a custom resource data model.

Admission webhooks are not often included in a prototype or pre-alpha release of an operator, but commonly come into play when refining the user experience for a stable release of a project. Chapter 8 covers the subject of admission control in depth.

Finalizers

We have looked at examples of a custom resource being set as the owner of child resources to ensure they will be deleted when the parent custom resource is removed. However, this mechanism is not always sufficient. If the custom resource has relationships with other resources in the cluster where an ownership is *not* appropriate, or if conditions outside the cluster need to be updated when your custom resource is deleted, finalizers will likely be important to use.

Finalizers are added to the metadata of a resource as shown Example 11-16.

Example 11-16. AcmeNamespace manifest with a finalizer

```yaml
apiVersion: tenancy.acme.com/v1alapha1
kind: AcmeNamespace
metadata:
  name: team-x
  finalizers:
  - namespace.finalizer.tenancy.acme.com  ❶
spec:
  namespaceName: app-y
  adminUsername: sam
```

❶ String value used as a finalizer.

The string value used as your finalizer is not important to anything else in the system besides your controller. Just use a value that will safely be unique in case other controllers need to apply finalizers to the same resource.

When any finalizers are present on a resource, the API server will not delete the resource. If a delete request is received, it will instead update the resource to add a `deletionTimestamp` field to its metadata. This update to the resource will trigger a reconciliation in your controller. A check for this `deletionTimestamp` will need to be added to your controller's `Reconcile` method so that any pre-delete operations can be completed. Once complete, your controller can remove the finalizer. This will inform the API server that it may now delete the resource.

Common examples of pre-delete operations are in systems outside the cluster. In the Namespace operator example, if there are corporate chargeback systems that track Namespace usage and need to be updated when Namespaces are deleted, a finalizer could prompt your operator to update that external system before removing the Namespace. Other common examples are when a workload uses managed services, such as databases or object storage, as a part of the application stack. When an instance of the application is deleted, these managed service instances will likely need to be cleaned up as well.

Extending the Scheduler

The scheduler delivers core functionality in Kubernetes. A huge part of the value proposition of Kubernetes is the abstraction of pools of machines on which to run workloads. It is the scheduler that makes the determination of where Pods will run. It is fair to say that, together with the kubelet, these two controllers form the heart of Kubernetes around which all else is built. The scheduler is a cornerstone platform service for your application platform. In this section we will explore customizing, extending, and replacing the behavior of the scheduler.

It's helpful to keep in mind the parallels between core control plane components, like the scheduler, and the custom operators we have examined so far in this chapter. In both cases we are dealing with Kubernetes controllers managing Kubernetes resources. With our custom operators, we develop entirely new custom controllers, whereas the scheduler is a core controller deployed with every Kubernetes cluster. With our custom operators, we design and create new custom resources, whereas the scheduler manages the core Pod resource.

We have found it uncommon for users of Kubernetes to find a need to extend the scheduler or modify its behavior. However, considering how important it is to a cluster's function, it is prudent to both understand how the scheduler reaches scheduling decisions it makes and how to modify those decisions if the need arises. It bears repeating one more time: a large part of the genius of Kubernetes is its extensibility and modularity. If you find the scheduler doesn't meet your needs, you can modify or augment its behavior, or replace it altogether.

In exploring this topic, we're going to examine how the scheduler determines where to assign Pods so that we can understand what goes into each scheduling decision, and then see how we can influence those decisions with scheduling policies. We're also going to address the option of running multiple schedulers and even writing your own custom scheduler.

Predicates and Priorities

Before we look at how to extend or modify the scheduler, we first need to understand how the scheduler makes its decisions. The scheduler uses a two-step process to determine which Node a Pod will get scheduled to.

The first is filtering. In this step, the scheduler filters out Nodes that are ineligible to host a Pod using a number of predicates. For example, there is a predicate that checks to see if the Pod being scheduled tolerates a Node's taints. The control plane nodes commonly use a taint to ensure regular workloads don't get scheduled there. If a Pod has no tolerations, any tainted Nodes will be filtered out as ineligible targets for a Pod. Another predicate checks to ensure the Node has sufficient CPU and memory resources for any Pod that requests values for these resources. As you'd expect, if the Node has insufficient resources to satisfy the Pod spec, it is filtered out as an eligible host. When all predicates have checked for Node eligibility, the filtering step is complete. At this point if there are no eligible Nodes, the Pod will remain in a `Pending` state until conditions change, such as a new eligible Node being added to the cluster. If the list of Nodes consists of a single Node, scheduling may occur at this point. If there are multiple eligible Nodes, the scheduler proceeds to the second step.

The second step is scoring. This step uses priorities to determine which Node is the *best* fit for a particular Pod. One priority that helps to score a Node higher is the presence of a container image that is being used by the Pod. Another priority that will score a Node higher is the lack of any Pods that share the same Service as the Pod being scheduled. That is to say that the scheduler will attempt to distribute the Pods that share a Service across multiple Nodes for improved Node failure tolerance. The scoring step is also where `preferred...` affinity rules on Pods are implemented. At the end of the scoring step, each eligible Node has a score associated. The highest scoring Node is deemed the best fit for the Pod and it is scheduled there.

Scheduling Policies

Scheduling policies are used to configure the predicates and priorities the scheduler is to use. You can write a config file containing a scheduling policy to disk on the control plane nodes and provide the scheduler the `--policy-config-file` flag, but the preferred method is to use a ConfigMap. Provide the scheduler the `--policy-configmap` flag and thereafter you can update the scheduling policy via the API server. Note that if you go with the ConfigMap method, you will likely need to update the `system:kube-scheduler` ClusterRole to add a rule for getting ConfigMaps.

At the time of this writing, both the `--policy-config-file` and `--policy-configmap` flags for the scheduler still work, but they are marked as deprecated in the official documentation. For this reason, if you are implementing new custom scheduling behavior, we recommend using the scheduling profiles discussed in the next section rather than the policies discussed here.

For example, the policy ConfigMap in Example 11-17 will make a Node eligible for selection with a `nodeSelector` by a Pod only if it has a label with the key: `selectable`.

Example 11-17. An example ConfigMap defining a scheduling policy

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: scheduler-policy-config
  namespace: kube-system
data:
  policy.cfg: |+ ❶
    apiVersion: v1
    kind: Policy
    predicates:
    - name: "PodMatchNodeSelector" ❷
      argument:
        labelsPresence:
          labels:
          - "selectable" ❸
          presence: true ❹
```

❶ The filename the scheduler will expect to use for policies.

❷ The predicate name that implements `nodeSelectors`.

❸ The label key you wish to use for adding a constraint to selection. In this example, if a node does not have this label key present, it will not be selectable by a Pod.

❹ This indicates the provided label must be present. If `false` it would need to be absent. If using the example configuration of `presence: true`, a Node without the label `selectable: ""` will not be eligible for selection by a Pod.

With this scheduling policy in place, a Pod defined with the manifest in Example 11-18 would be scheduled to an eligible Node only with *both* the `device: gpu` *and* the `selectable: ""` labels present.

Example 11-18. A Pod manifest using the `nodeSelector` field to direct scheduling

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: terminator
spec:
  containers:
  - image: registry.acme.com/skynet/t1000:v1
    name: terminator
  nodeSelector:
    device: gpu
```

Scheduling Profiles

Scheduling profiles allow you to enable or disable plug-ins that are compiled into the scheduler. You can specify a profile by passing a filename to the `--config` flag when running the scheduler. These plug-ins implement the various extension points that include—but are not limited to—the filter and scoring steps we covered earlier. In our experience it is rarely necessary to customize the scheduler in this way. But should you find a need for it, the Kubernetes documentation provides instructions.

Multiple Schedulers

It should be noted that you are not limited to one scheduler. You can deploy any number of schedulers that are either Kubernetes schedulers with different policies and profiles, or even custom-built schedulers. If running multiple schedulers you can supply the `schedulerName` in the spec for a Pod, which will determine which scheduler carries out the scheduling for that Pod. Given the added complexity of following this multischeduler model, consider using dedicated clusters for workloads with such specialized scheduling requirements.

Custom Scheduler

In the unlikely event that you are unable to use the Kubernetes scheduler, even with the use of policies and profiles, you have the option to develop and use your own scheduler. This would entail developing a controller that watches Pod resources and whenever a new Pod is created, determine where the Pod should run and update the `nodeName` field for that Pod. While this is a narrow scope, this is not a simple exercise. As we have seen in this section, the core scheduler is a sophisticated controller that routinely evaluates numerous complex factors into account when making scheduling decisions. If your requirements are specialized enough to demand a custom scheduler, it's likely you will have to spend considerable engineering effort to refine its behavior. We recommend proceeding with this approach only if you have exhausted

your options with the existing scheduler and have deep Kubernetes expertise to tap into for the project.

Summary

It's important to understand the points of extension available with Kubernetes and how best to add the platform services required to meet your tenants' needs. Study the operator pattern and the use cases for Kubernetes operators. If you find a compelling need to build an operator, decide what development tooling and language you will use, design your custom resource's data model, and then build a Kubernetes controller to manage that custom resource. Finally, if the default scheduler behavior does not meet your requirements, look into scheduling policies and profiles to modify its behavior. In extreme edge cases, you have the option to develop your own custom scheduler to replace, or run in conjunction with, the default scheduler.

Using the principles and practices laid out in this chapter, you are no longer constrained by the utilities and software provided by the community or your company's vendors. If you encounter important requirements for which there is no existing solution, you have at your disposal the tools and guidelines to add any specialized platform services your business needs may require.

CHAPTER 12

Multitenancy

When building a production application platform atop Kubernetes, you must consider how to handle the tenants that will run on the platform. As we've discussed throughout this book, Kubernetes provides a set of foundational features you can use to implement many requirements. Workload tenancy is no different. Kubernetes offers various knobs you can use to ensure tenants can safely coexist on the same platform. With that said, Kubernetes does not define a tenant. A tenant can be an application, a development team, a business unit, or something else. Defining a tenant is up to you and your organization, and we hope this chapter will help you with that task.

Once you establish who your tenants are, you must determine whether multiple tenants should run on the same platform. In our experience helping large organizations build application platforms, we've found that platform teams are usually interested in operating a multitenant platform. With that said, this decision is firmly rooted in the nature of the different tenants and the trust that exists between them. For example, an enterprise offering a shared application platform is a different story than a company offering containers-as-a-service to external customers.

In this chapter, we will first explore the degrees of tenant isolation you can achieve with Kubernetes. The nature of your workloads and your specific requirements will dictate how much isolation you need to provide. The stronger the isolation, the higher the investment you need to make in this area. We will then discuss Kubernetes Namespaces, an essential building block that enables a large portion of the multitenancy capabilities in Kubernetes. Finally, we will dig into the different Kubernetes features you can leverage to isolate tenants on a multitenant cluster, including Role-Based Access Control (RBAC), Resource Requests and Limits, Pod Security Policies, and others.

Degrees of Isolation

Kubernetes lends itself to various tenancy models, each with pros and cons. The most critical factor that determines which model to implement is the degree of isolation demanded by your workloads. For example, running untrusted code developed by different third parties usually requires more robust isolation than hosting your organization's internal applications. Broadly speaking, there are two tenancy models you can follow: single-tenant clusters and multitenant clusters. Let's discuss the strengths and weaknesses of each model.

Single-Tenant Clusters

The single-tenant cluster model (depicted in Figure 12-1) provides the strongest isolation between tenants, as there is no sharing of cluster resources. This model is rather appealing as you do not have to solve the complex multitenancy problems that can otherwise arise. In other words, there is no tenant isolation problem to solve.

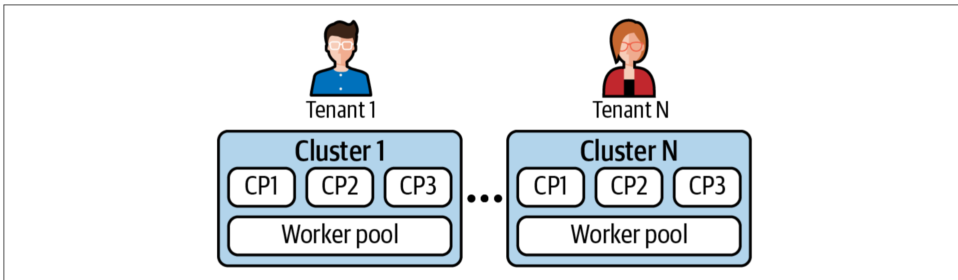


Figure 12-1. Each tenant runs in a separate cluster (CP represents a control plane node).

Single-tenant clusters can be viable if you have a small number of tenants. However, the model can suffer from the following downsides:

Resource overhead
: Each single-tenant cluster has to run its own control plane, which in most cases, requires at least three dedicated nodes. The more tenants you have, the more resources dedicated to cluster control planes—resources that you could otherwise use to run workloads. In addition to the control plane, each cluster hosts a set of workloads to provide platform services. These platform services also incur overhead as they could otherwise be shared among different tenants in a multitenant cluster. Monitoring tools, policy controllers (e.g., Open Policy Agent), and Ingress controllers are good examples.

Increased management complexity
: Managing a large number of clusters can become a challenge for platform teams. Each cluster needs to be deployed, tracked, upgraded, etc. Imagine having to

remediate a security vulnerability across hundreds of clusters. Investing in advanced tooling is necessary for platform teams to do this effectively.

Even with the drawbacks just mentioned, we have seen many successful implementations of single-tenant clusters in the field. And with cluster life cycle tooling such as Cluster API reaching maturity, the single-tenant model has become easier to adopt. With that said, most of our focus in the field has been helping organizations with multitenant clusters, which we'll discuss next.

Multitenant Clusters

Clusters that host multiple tenants can address the downsides of single-tenant clusters we previously discussed. Instead of deploying and managing one cluster per tenant, the platform team can focus on a smaller number of clusters, which reduces the resource overhead and management complexity (as seen in Figure 12-2). With that said, there is a trade-off being made. The implementation of multitenant clusters is more complicated and nuanced, as you have to ensure that tenants can coexist without affecting each other.

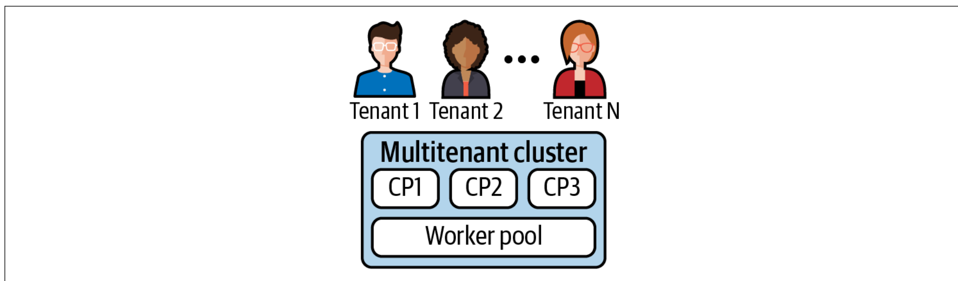


Figure 12-2. A single cluster shared by multiple tenants (CP represents a control plane node).

Multitenancy comes in two broad flavors, soft multitenancy and hard multitenancy. *Soft multitenancy*, sometimes referred to as "multiteam," assumes that some level of trust exists between the tenants on the platform. This model is usually viable when tenants belong to the same organization. For example, an enterprise application platform hosting different tenants can generally assume a soft multitenancy posture. This is because the tenants are incentivized to be good neighbors as they move their organization toward success. Nevertheless, even though the intent is positive, tenant isolation is still necessary given that unintentional issues can arise (e.g., vulnerabilities, bugs, etc.).

On the other hand, the *hard multitenancy* model establishes that there is no trust between tenants. From a security point of view, the tenants are even considered adversaries to ensure the proper isolation mechanisms are put in place. A platform running untrusted code that belongs to different organizations is a good example. In

this case, strong isolation between tenants is critical to ensure they can share the cluster safely.

Building on our housing analogy theme from Chapter 1, we can say that the soft multitenancy model is equivalent to a family living together. They share the kitchen, living room, and utilities, but each family member has their own bedroom. In contrast, the hard multitenancy model is better represented by an apartment building. Multiple families share the building, but each family lives behind a locked front door.

While the soft and hard multitenancy models can help guide conversations about multitenant platforms, the implementation is not as clear-cut. The reality is that multitenancy is best described as a spectrum. On the one end, we have no isolation at all. Tenants are free to do anything on the platform and consume all its resources. On the other end, we have full tenant isolation, where tenants are strictly controlled and isolated across all layers of the platform.

As you can imagine, establishing a production multitenant platform with no tenant isolation is not viable. At the same time, building a multitenant platform with complete tenant isolation can be a costly (or even futile) endeavor. Thus, it is important to find the sweet spot in the multitenancy spectrum that will work for your workloads and organization as a whole.

To determine the isolation required for your workloads, you must consider the different layers where you *can* apply isolation in a Kubernetes-based platform:

Workload plane
: The workload plane consists of the nodes where the workloads get to run. In a multitenant scenario, workloads are typically scheduled across the shared pool of nodes. Isolation at this level involves fair sharing of node resources, security and network boundaries, etc.

Control plane
: The control plane encompasses the components that make up a Kubernetes cluster, such as the API server, the controller manager, and the scheduler. There are different mechanisms available in Kubernetes to segregate tenants at this level, including authorization (i.e., RBAC), admission control, and API priority and fairness.

Platform services
: Platform services include centralized logging, monitoring, ingress, in-cluster DNS, and others. Depending on the workloads, these services or capabilities might also require some level of isolation. For example, you might want to prevent tenants from inspecting each other's logs or discovering each other's services via the cluster's DNS server.

Kubernetes provides different primitives you can use to implement isolation at each of these layers. Before digging into them, we will discuss the Kubernetes Namespace, the foundational boundary that allows you to segregate tenants on a cluster.

The Namespace Boundary

Namespaces enable a number of different capabilities in the Kubernetes API. They allow you to organize your cluster, enforce policy, control access, etc. More importantly, they are a critical building block when implementing a multitenant Kubernetes platform, as they provide the foundation to onboard and isolate tenants.

When it comes to tenant isolation, however, it is important to keep in mind that the Namespace is a logical construct in the Kubernetes control plane. Without additional policy or configuration, the Namespace has no implications on the workload plane. For example, workloads that belong to different Namespaces are likely to run on the same node unless advanced scheduling constraints are put in place. In the end, the Namespace is merely a piece of metadata attached to resources in the Kubernetes API.

Having said that, many of the isolation mechanisms that we will explore in this chapter hinge on the Namespace construct. RBAC, resource quotas, and network policies are examples of such mechanisms. Thus, one of the first decisions to make when designing your tenancy strategy is establishing how to leverage Namespaces. When helping organizations in the field, we have seen the following approaches:

Namespace per team
: In this model, each team has access to a single Namespace in the cluster. This approach makes it simple to apply policy and quota to specific teams. However, it can be challenging for teams to exist within a single Namespace if they own many services. Overall, we find that this model can be viable for small organizations that are getting started with Kubernetes.

Namespace per application
: This approach provides a Namespace for each application in the cluster, making it easier to apply application-specific policy and quota. The downside is that this model usually results in tenants having access to multiple Namespaces, which can complicate the tenant onboarding process and the ability to apply tenant-level policy and quota. With that said, this approach is perhaps the most viable for large organizations and enterprises building multitenant platforms.

Namespace per tier
: This pattern establishes different runtime tiers (or environments) using Namespaces. We usually avoid this approach, as we prefer to use separate clusters for development, staging, and production tiers.

The approach to use largely depends on your isolation requirements and the structure of your organization. If you are leaning toward the Namespace per team model, remember that all resources in the Namespace are accessible by all team members or workloads in the Namespace. For example, assuming Alice and Bob are on the same team, there's no way to prevent Alice from looking at Bob's Secrets if they are both authorized to get Secrets in the team's Namespace.

Multitenancy in Kubernetes

Up to this point, we have discussed the different tenancy models you can implement when building a Kubernetes-based platform. In the rest of this chapter, we will focus on multitenant clusters and the various Kubernetes capabilities you can leverage to safely and effectively host your tenants. As you read through these sections, you will find that we have covered some of these capabilities in other chapters. In those cases, we will brush up on them once more, but we will focus on the multitenancy aspect of them.

First, we will focus on the isolation mechanisms available in the control plane layer. Mainly, RBAC, resource quotas, and validating admission webhooks. We will then move onto the workload plane, where we will discuss resource requests and limits, Network Policies, and Pod Security Policies. Finally, we will touch on monitoring and centralized logging as example platform services that you can design with multitenancy in mind.

Role-Based Access Control (RBAC)

When hosting multiple tenants in the same cluster, you must enforce isolation at the API server layer to prevent tenants from modifying resources that do not belong to them. The RBAC authorization mechanism enables you to configure this policy. As we discussed in Chapter 10, the API server supports different mechanisms to establish a user's or tenant's identity. Once established, the tenant's identity is passed on to the RBAC system, which determines whether the tenant is authorized to perform the requested action.

As you onboard tenants onto the cluster, you can grant them access to one or more Namespaces in which they can create and manage API resources. To authorize each tenant, you must bind Roles or ClusterRoles with their identities. The binding is achieved with the RoleBinding resource. The following snippet shows an example RoleBinding that grants the `app1-viewer` Group view access to the `app1` Namespace. Unless you have a good use case, avoid using ClusterRoleBindings for tenants, as it authorizes the tenant to leverage the bound role across all Namespaces.

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: viewers
  namespace: app1
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: app1-viewer
```

You'll notice in the example that the RoleBinding references a ClusterRole named `view`. This is a built-in role that is available in Kubernetes. Kubernetes provides a set of built-in roles that cover common use cases:

view
: The view role grants tenants read-only access to Namespace-scoped resources. This role can be bound to all the developers in a team, for example, as it allows them to inspect and troubleshoot their resources in production clusters.

edit
: The edit role allows tenants to create, modify, and delete Namespace-scoped resources, in addition to viewing them. Given this role's abilities, binding of this role is highly dependent on your approach to application deployment.

admin
: In addition to viewing and editing resources, the admin role can create Roles and RoleBindings. This role is usually bound to the tenant administrator to delegate Namespace-management concerns.

These built-in roles are a good starting point. With that said, they can be considered too broad, as they grant access to a vast number of resources in the Kubernetes API. To follow the principle of least privilege, you can create tightly scoped roles that allow the minimum set of resources and actions required to get the job done. However, keep in mind that this can result in management overhead as you potentially need to manage many unique roles.

In most Kubernetes deployments, tenants are typically authorized to list all Namespaces on the cluster. This is problematic if you need to prevent tenants from knowing what other Namespaces exist, as there is currently no way of achieving this using the Kubernetes RBAC system. If you do have this requirement, you must build a higher-level abstraction to handle it (OpenShift's Project resource is an example abstraction that addresses this).

RBAC is a must when running multiple tenants in the same cluster. It provides isolation at the control plane layer, which is necessary to prevent tenants from viewing and modifying each other's resources. Make sure to leverage RBAC when building a multitenant Kubernetes-based platform.

Resource Quotas

As a platform operator offering a multitenant platform, you need to ensure that each tenant gets an appropriate share of the limited cluster resources. Otherwise, nothing prevents an ambitious (or perhaps malicious) tenant from consuming the entire cluster and effectively starving the other tenants.

To place a limit on resource consumption, you can use the resource quotas feature of Kubernetes. Resource quotas apply at the Namespace level, and they can limit two kinds of resources. On one hand, you can control the amount of compute resources available to a Namespace, such as CPU, memory, and storage. On the other hand, you can limit the number of API objects that can be created within a Namespace, such as the number of Pods, Services, etc. A common scenario that calls for limiting API objects is to control the number of LoadBalancer Services in cloud environments, which can get expensive.

Because quotas apply at the Namespace level, your Namespace strategy impacts how you configure quotas. If tenants get access to a single Namespace, applying quotas to each tenant is straightforward, as you can create a ResourceQuota for each tenant in their Namespace. The story is more complicated when tenants have access to multiple Namespaces. In this case, you need extra automation or an additional controller to enforce quota across different Namespaces. (The Hierarchical Namespace Controller is an attempt at addressing this issue).

To further explore ResourceQuotas, let's explore them in action. The following example shows a ResourceQuota that limits the Namespace to consume up to 1 CPU and 512 MiB of memory:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-mem
  namespace: app1
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 512Mi
    limits.cpu: "1"
    limits.memory: 512Mi
```

As Pods in the `app1` Namespace start to get scheduled, the quota is consumed accordingly. For example, if we create a Pod that requests 0.5 CPUs and 256 MiB, we can see the updated quota as follows:

```
$ kubectl describe resourcequota cpu-mem
Name:            cpu-mem
Namespace:       app1
Resource         Used   Hard
--------         ----   ----
limits.cpu       500m   1
limits.memory    512Mi  512Mi
requests.cpu     500m   1
requests.memory  512Mi  512Mi
```

Attempts to consume resources beyond the configured quota are blocked by an admission controller, as shown in the following error message. In this case, we were trying to consume 2 CPUs and 2 GiB of memory but were limited by the quota:

```
$ kubectl apply -f my-app.yaml
Error from server (Forbidden):
  error when creating "my-app.yaml": pods "my-app" is forbidden:
    exceeded quota: cpu-mem,
      requested: limits.cpu=2,limits.memory=2Gi,
                 requests.cpu=2,requests.memory=2Gi,
      used: limits.cpu=0,limits.memory=0,
                 requests.cpu=0,requests.memory=0,
      limited: limits.cpu=1,limits.memory=512Mi,
                 requests.cpu=1,requests.memory=512Mi
```

As you can see, ResourceQuotas give you the ability to control how tenants consume cluster resources. They are critical when running a multitenant cluster, as they ensure tenants can safely share the cluster's limited resources.

Admission Webhooks

Kubernetes has a set of built-in admission controllers that you can use to enforce policy. The ResourceQuota functionality we just covered is implemented using an admission controller. While the built-in controllers help solve common use cases, we typically find that organizations need to extend the admission layer to isolate and limit tenants further.

Validating and mutating admission webhooks are the mechanisms that enable you to inject custom logic into the admission pipeline. We will not dig into the implementation details of these webhooks, as we have already covered them in Chapter 8. Instead, we will explore some of the multitenancy use cases we've solved in the field with custom admission webhooks:

Standardized labels

You can enforce a standard set of labels across all API objects using a validating admission webhook. For example, you could require all resources to have an `owner` label. Having a standard set of labels is useful, as labels provide a way to query the cluster and even support higher-level features, such as network policies and scheduling constraints.

Require fields

Like enforcing a standard set of labels, you can use a validating admission webhook to mark fields of certain resources as required. For example, you can require all tenants to set the `https` field of their Ingress resources. Or perhaps require tenants to always set readiness and liveness probes in their Pod specifications.

Set guardrails

Kubernetes has a broad set of features that you might want to limit or even disable. Webhooks allow you to set guardrails around specific functionality. Examples include disabling specific Service types (e.g., NodePorts), disallowing node selectors, controlling Ingress hostnames, and others.

MultiNamespace resource quotas

We have experienced cases in the field where organizations needed to enforce resource quotas across multiple Namespaces. You can use a custom admission webhook/controller to implement this functionality, as the ResourceQuota object in Kubernetes is Namespace-scoped.

Overall, admission webhooks are a great way to enforce custom policy in your multitenant clusters. And the emergence of policy engines such as Open Policy Agent (OPA) and Kyverno make it even simpler to implement them. Consider leveraging such engines to isolate and limit tenants in your clusters.

API Priority and Fairness

The API Priority and Fairness feature in Kubernetes is another mechanism you can leverage to isolate tenants at the control plane layer. This feature prevents the API server from being overloaded by limiting the number of concurrent requests it handles according to a configurable policy.

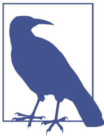
The API server sits at the heart of control plane functionality. If one tenant overloads it, this is likely to have significant consequences for other tenants. The API Priority and Fairness capability can thwart any attempt from a malicious tenant or buggy API client from causing this overload. Instead, the client's requests are queued or rejected according to the configured policy.

The API Priority and Fairness feature is relatively new. As of this writing, the feature is in alpha and we have yet to see it implemented in the field. Thus, we would recommend holding off on enabling it unless you have a strong reason to use it. Even then, if you find you need this capability, we would encourage you to evaluate whether running multiple clusters instead of leveraging this capability would result in a simpler implementation.

Resource Requests and Limits

Kubernetes schedules workloads onto a shared pool of cluster nodes. Commonly, workloads from different tenants get scheduled onto the same node and thus share the node's resources. Ensuring that the resources are shared fairly is one of the most critical concerns when running a multitenant platform. Otherwise, tenants can negatively affect other tenants that are colocated on the same node.

Resource requests and limits in Kubernetes are the mechanisms that isolate tenants from one another when it comes to compute resources. Resource requests are generally fulfilled at the Kubernetes scheduler level (CPU requests are also reflected at runtime, as we will see later). In contrast, resource limits are implemented at the node level using Linux control groups (cgroups) and the Linux Completely Fair Scheduler (CFS).



While requests and limits provide adequate isolation for production workloads, it should be known that this isolation is not as strict as that provided by a hypervisor. Completely removing noisy neighbor symptoms from workloads can be challenging in containerized environments. Be sure to experiment and understand the implication of multiple workloads under load on a given Kubernetes node.

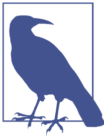
In addition to providing resource isolation, resource requests and limits determine a Pod's Quality of Service (QoS) class. The QoS class is important because it determines the order in which the kubelet evicts Pods when a node is running low on resources. Kubernetes offers the following QoS classes:

Guaranteed
: Pods with CPU limits equal to CPU requests and memory limits equal to memory requests. This must be true across all containers. The kubelet seldom evicts Guaranteed Pods.

Burstable
: Pods that do not qualify as Guaranteed and have at least one container with CPU or memory requests. The kubelet evicts Burstable Pods based on how many

resources they are consuming above their requests. Pods bursting higher above their requests are evicted before Pods bursting closer to their requests.

BestEffort
: Pods that have no CPU or memory limits or requests. These Pods run on a "best effort" basis. They are the first to be evicted by the kubelet.



Pod eviction is a complex process. In addition to using QoS classes to rank Pods, the kubelet also considers Pod priorities when making eviction decisions. The Kubernetes documentation has an excellent article that discusses "Out of Resource" handling in greater detail.

Now that we know that resource requests and limits provide tenant isolation and determine a Pod's QoS class, let's dive into the details of resource requests and limits. Even though Kubernetes supports requesting and limiting different resources, we will focus our discussion on CPU and memory, the essential resources that all workloads need at runtime. Let's discuss memory requests and limits first.

Each container in a Pod can specify memory requests and limits. When memory requests are set, the scheduler adds them up to get the Pod's overall memory request. With this information, the scheduler finds a node with enough memory capacity to host the Pod. If none of the cluster nodes have enough memory, the Pod remains in a pending state. Once scheduled, though, the containers in the Pod are guaranteed the requested memory.

A Pod's memory request represents a guaranteed lower bound for the memory resource. However, they can consume additional memory if it's available on the node. This is problematic because the Pod uses memory that the scheduler can assign to other workloads or tenants. When a new Pod is scheduled onto the same node, the Pods may fight over the memory. To honor the memory requests of both Pods, the Pod consuming memory above its request is terminated. Figure 12-3 depicts this process.

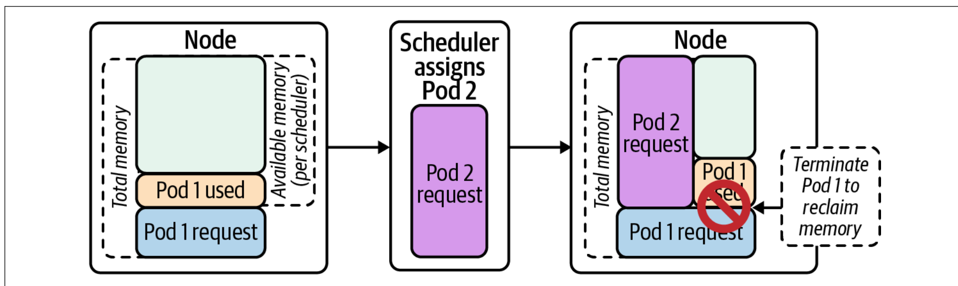


Figure 12-3. Pod consuming memory above its request is terminated to reclaim memory for the new Pod.

In order to control the amount of memory that tenants can consume, we must include memory limits on the workloads, which enforce an upper bound on the amount of memory available to a given workload. If the workload attempts to consume memory above the limit, the workload is terminated. This is because memory is a noncompressible resource. There is no way to throttle memory, and thus the process must be terminated when the node's memory is under contention. The following snippet shows a container that was out-of-memory killed (OOMKilled). Notice the "Reason" in the "Last State" section of the output:

```
$ kubectl describe pod memory
Name:         memory
Namespace:    default
Priority:     0
... <snip> ...
Containers:
  stress:
    ... <snip> ...
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    1
      Started:      Fri, 23 Oct 2020 10:11:51 -0400
      Finished:     Fri, 23 Oct 2020 10:11:56 -0400
    Ready:          True
    Restart Count:  1
    Limits:
      memory:  100Mi
    Requests:
      memory:     100Mi
```

A common question we encounter in the field is whether one should allow tenants to set memory limits higher than requests. In other words, whether nodes should be oversubscribed on memory. This question boils down to a trade-off between node density and stability. When you oversubscribe your nodes, you increase node density but decrease workload stability. As we've seen, workloads that consume memory above their requests get terminated when memory comes under contention. In most cases, we encourage platform teams to avoid oversubscribing nodes, as they typically consider stability more important than tightly packing nodes. This is especially the case in clusters hosting production workloads.

Now that we've covered memory requests and limits, let's shift our discussion to CPU. In contrast to memory, CPU is a compressible resource. You can throttle processes when CPU is under contention. For this reason, CPU requests and limits are somewhat more complex than memory requests and limits.

CPU requests and limits are specified using CPU units. In most cases, 1 CPU unit is equivalent to 1 CPU core. Requests and limits can be fractional (e.g., 0.5 CPU) and they can be expressed using millis by adding an `m` suffix. 1 CPU unit equals 1000m CPU.

When containers within a Pod specify CPU requests, the scheduler finds a node with enough capacity to place the Pod. Once placed, the kubelet converts the requested CPU units into cgroup CPU shares. CPU shares is a mechanism in the Linux kernel that grants CPU time to cgroups (i.e., the processes within the cgroup). The following are critical aspects of CPU shares to keep in mind:

- CPU shares are relative. 1000 CPU shares does not mean 1 CPU core or 1000 CPU cores. Instead, the CPU capacity is proportionally divided among all cgroups according to their relative shares. For example, consider two processes in different cgroups. If process 1 (P1) has 2000 shares, and process 2 (P2) has 1000 shares, P1 will get twice the CPU time as P2.
- CPU shares come into effect only when the CPU is under contention. If the CPU is not fully utilized, processes are not throttled and can consume additional CPU cycles. Following the preceding example, P1 will get twice the CPU time as P2 only when the CPU is 100% busy.

CPU shares (CPU requests) provide the CPU resource isolation necessary to run different tenants on the same node. As long as tenants declare CPU requests, the CPU capacity is shared according to those requests. Consequently, tenants are unable to starve other tenants from getting CPU time.

CPU limits work differently. They set an upper bound on the CPU time that each container can use. Kubernetes leverages the bandwidth control feature of the Completely Fair Scheduler (CFS) to implement CPU limits. CFS bandwidth control uses time periods to limit CPU consumption. Each container gets a quota within a configurable period. The quota determines how much CPU time can be consumed in every period. If the container exhausts the quota, the container is throttled for the rest of the period.

By default, Kubernetes sets the period to 100 ms. A container with a limit of 0.5 CPUs gets 50 ms of CPU time every 100 ms, as depicted in Figure 12-4. A container with a limit of 3 CPUs gets 300 ms of CPU time in every 100 millisecond period, effectively allowing the container to consume up to 3 CPUs every 100 ms.

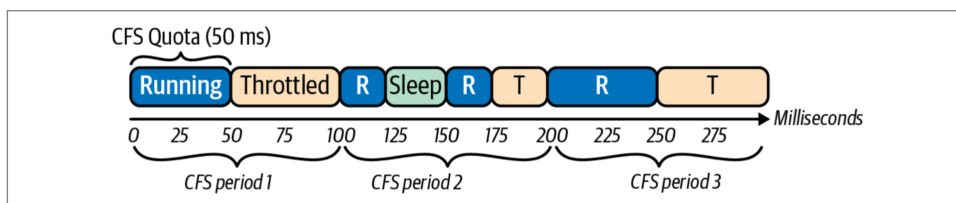


Figure 12-4. CPU consumption and throttling of a process running in a cgroup that has a CFS period of 100 milliseconds and a CPU quota of 50 milliseconds.

Due to the nature of CPU limits, they can sometimes result in surprising behavior or unexpected throttling. This is usually the case in multithreaded applications that can consume the entire quota at the very beginning of the period. For example, a container with a limit of 1 CPU will get 100 ms of CPU time every 100 ms. Assuming the container has 5 threads using CPU, the container consumes the 100 ms quota in 20 ms and gets throttled for the remaining 80 ms. This is depicted in Figure 12-5.

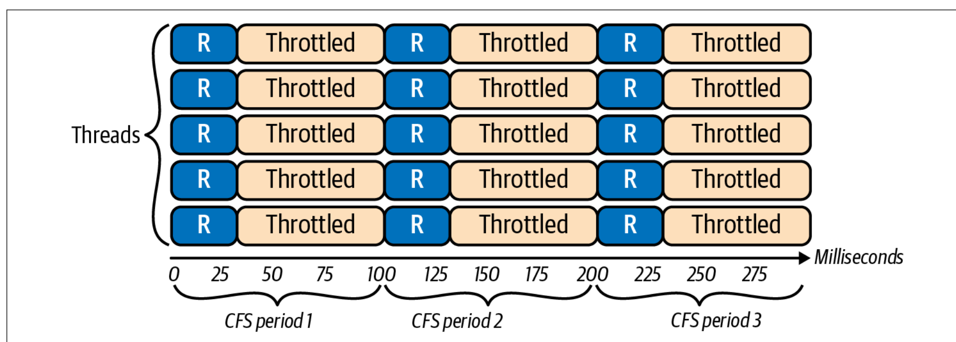


Figure 12-5. Multithreaded application consumes the entire CPU quota in the first 20 milliseconds of the 100-millisecond period.

Enforcing CPU limits is useful to minimize the variability of an application's performance, especially when running multiple replicas across different nodes. This variability in performance stems from the fact that, without CPU limits, replicas can burst and consume idle CPU cycles, which *might* be available at different times. By setting the CPU limits equal to the CPU requests, you remove the variability as the workloads get precisely the CPU they requested. (Google and IBM published an excellent whitepaper that discusses CFS bandwidth control in more detail.) In a similar vein, CPU limits play a critical role in performance testing and benchmarking. Without any CPU limits, your benchmarks will produce inconclusive results, as the CPU available to your workloads will vary based on the nodes where they got scheduled and the amount of idle CPU available.

If your workloads require predictable access to CPU (e.g., latency-sensitive applications), setting CPU limits equal to CPU requests is helpful. Otherwise, placing an upper bound on CPU cycles is not necessary. When the CPU resources on a node are under contention, the CPU shares mechanism ensures that workloads get their fair share of CPU time, according to their container's CPU requests. When the CPU is not under contention, the idle CPU cycles are not wasted as workloads opportunistically consume them.

Linux Kernel Bug Impacting Kubernetes CPU Limits

Another issue with CPU limits is a Linux kernel bug that throttles containers unnecessarily. This has a significant impact on latency-sensitive workloads, such as web services. To avoid this issue, Kubernetes users resorted to different workarounds, including:

- Removing CPU limits from Pod specifications
- Disabling enforcement of CPU limits by setting the kubelet flag `--cpu-cfs-quota=false`
- Reducing the CFS period to 5–10ms by setting the kubelet flag `--cpu-cfs-quota-period`

Depending on your Linux kernel version, you might not have to implement these workarounds, as the bug has been fixed in version 5.4 of the Linux kernel and backported to versions 4.14.154+, 4.19.84+, and 5.3.9+. If you need to enforce CPU limits, consider upgrading your Linux kernel version to avoid this bug.

Network Policies

In most deployments, Kubernetes assumes all Pods running on the platform can communicate with each other. As you can imagine, this stance is problematic for multitenant clusters, where you might want to enforce network-level isolation between the tenants. The NetworkPolicy API is the mechanism you can leverage to ensure tenants are isolated from each other at the network layer.

We explored Network Policies in Chapter 5, where we discussed the role of the Container Networking Interface (CNI) plug-ins in enforcing network policies. In this section, we will discuss the *default deny-all* network policy model, a common approach to Network Policy, especially in multitenant clusters.

As a platform operator, you can establish a default deny-all network policy across the entire cluster. By doing so, you take the strongest stance regarding network security and isolation, given that tenants are fully isolated as soon as they are onboarded onto the platform. Furthermore, you drive tenants to a model where they have to declare their workloads' network interactions, which improves their applications' network security.

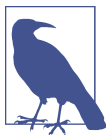
When it comes to implementing a default deny-all policy, you can follow two different paths, each with its pros and cons. The first approach leverages the NetworkPolicy API available in Kubernetes. Because this is a core API, this implementation is portable across different CNI plug-ins. However, given that the NetworkPolicy object is Namespace-scoped, it requires you to create and manage multiple default deny-all

NetworkPolicy resources, one per Namespace. Additionally, because tenants need the authorization to create their own NetworkPolicy objects, you must implement additional controls (usually via admission webhooks, as discussed earlier) to prevent tenants from modifying or deleting the default deny-all policy. The following snippet shows a default deny-all NetworkPolicy object. The empty Pod selector selects all the Pods in the Namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
  namespace: tenant-a
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

The alternative approach is to leverage CNI plug-in-specific Custom Resource Definitions (CRDs). Some CNI plug-ins, such as Antrea, Calico, and Cilium, provide CRDs that enable you to specify cluster-level or "global" network policy. These CRDs help you reduce the implementation and management complexity of the default deny-all policy, but they tie you to a specific CNI plug-in. The following snippet shows an example Calico GlobalNetworkPolicy CRD that implements the default deny-all policy:

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: default-deny
spec:
  selector: all()
  types:
  - Ingress
  - Egress
```



Typically, default deny-all network policy implementations make exceptions to allow fundamental network traffic, such as DNS queries to the cluster's DNS server. Additionally, they are not applied to the kube-system Namespace and any other system-level Namespaces to prevent breaking the cluster. The YAML snippets in the preceding code do not address these concerns.

As with most choices, whether to use the built-in NetworkPolicy object or a CRD results in a trade-off between portability and simplicity. In our experience, we've found that the simplicity gained by leveraging the CNI-specific CRD is usually worth the trade-off, given that switching CNI plug-ins is an uncommon event. With that said, you might not have to make this choice in the future, as the Kubernetes

Networking Special Interest Group (sig-network) is looking at evolving the Network-Policy APIs to support cluster-scoped network policies.

Once the default deny-all policy is in place, tenants are responsible for poking holes in the network fabric to ensure their applications can function. They achieve this using the NetworkPolicy resource, in which they specify ingress and egress rules that apply to their workloads. For example, the following snippet shows a NetworkPolicy that could be applied to a web service. It allows Ingress or incoming traffic from the web frontend, and it allows Egress or outgoing traffic to its database:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webservice
  namespace: reservations
spec:
  podSelector:
    matchLabels:
      role: webservice
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - podSelector:
        role: database
    ports:
    - protocol: TCP
      port: 3306
```

Enforcing a default deny-all network policy is a critical tenant isolation mechanism. As you build your platform atop Kubernetes, we strongly encourage you to follow this pattern, especially if you plan to host multiple tenants.

Pod Security Policies

Pod Security Policies (PSPs) are another important mechanism to ensure tenants can coexist safely on the same cluster. PSPs control critical security parameters of Pods at runtime, such as their ability to run as privileged, access host volumes, bind to the host network, and more. Without PSPs (or a similar policy enforcement mechanism), workloads are free to do virtually anything on a cluster node.

Kubernetes enforces most of the controls implemented via PSPs using an admission controller. (The rule that requires a nonroot user is sometimes enforced by the kubelet, which verifies the runtime user of the container after downloading the image.) Once the admission controller is enabled, attempts to create a Pod are blocked unless they are allowed by a PSP. Example 12-1 shows a restrictive PSP that we typically define as the *default* policy in multitenant clusters.

Example 12-1. Sample restrictive PodSecurityPolicy

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: default
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: |
      'docker/default,runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false ❶
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes: ❷
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false ❸
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot' ❹
  seLinux:
    rule: 'RunAsAny' ❺
  supplementalGroups: ❻
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  fsGroup: ❼
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  readOnlyRootFilesystem: false
```

❶ Disallow privileged containers.

❷ Control the volume types that Pods can use.

❸ Prevent Pods from binding to the underlying host's network stack.

❹ Ensure that containers run as a nonroot user.

❺ This policy assumes the nodes are using AppArmor rather than SELinux.

❻ Specify the allowed group IDs that containers can use. The root gid (0) is disallowed.

❼ Control the group IDs applied to volumes. The root gid (0) is disallowed.

The existence of a PSP that allows the Pod is not enough for the Pod to be admitted. The Pod must be authorized to *use* the PSP as well. PSP authorization is handled using RBAC. Pods can use a PSP if their Service Account is authorized to use it. Pods can also use a PSP if the actor creating the Pod is authorized to use the PSP. However, given that Pods are seldom created by cluster users, using Service Accounts for PSP authorization is the more common approach. The following snippet shows a Role and RoleBinding that authorizes a Service Account to use a specific PSP named `sample-psp`:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: sample-psp
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  resourceNames: ['sample-psp']
  verbs: ['use']
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: sample-psp
subjects:
- kind: ServiceAccount
  name: my-app
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sample-psp
```

In most cases, the platform team is responsible for creating and managing the PSPs and enabling tenants to use them. As you design the policies, always follow the

principle of least privilege. Only allow the minimum set of privileges and capabilities required for Pods to complete their work. As a starting point, we typically recommend creating the following policies:

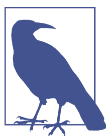
Default
: The default policy is usable by all tenants on the cluster. It should be a restrictive policy that blocks all privileged operations, drops all Linux capabilities, disallows running as the root user, etc. (See Example 12-1 for the YAML definition of this policy.) To make it the default policy, you can authorize all Pods in the cluster to use this PSP using a ClusterRole and ClusterRoleBinding.

Kube-system
: The kube-system policy is for the system components that exist within the kube-system Namespace. Due to the nature of these components, this policy needs to be more permissive than the default policy. For example, it must allow Pods to mount `hostPath` volumes and run as root. In contrast to the default policy, the RBAC authorization is achieved using a RoleBinding scoped to all Service Accounts in the kube-system Namespace.

Networking
: The networking policy is geared toward the cluster's networking components, such as the CNI plug-in. These Pods require even more privileges to manipulate the networking stack of cluster nodes. To isolate this policy to networking Pods, create a RoleBinding that authorizes only the networking Pods Service Accounts to use the policy.

With these policies in place, tenants can deploy unprivileged workloads into the cluster. If there is a workload that needs additional privileges, you must determine whether you can tolerate the risk of running that privileged workload in the same cluster. If so, create a different policy tailored to that workload. Grant the privileges required by the workload and only authorize that workload's Service Account to use the PSP.

PSPs are a critical enforcement mechanism in multitenant platforms. They control what tenants can and cannot do at runtime, as they run alongside other tenants on shared nodes. When building your platform, you should leverage PSPs to ensure tenants are isolated and protected from each other.



The Kubernetes community is discussing the possibility of removing the PodSecurityPolicy API and admission controller from the core project. If removed, you can leverage a policy engine such as Open Policy Agent or Kyverno to implement similar functionality.

Multitenant Platform Services

In addition to isolating the Kubernetes control plane and workload plane, you can enforce isolation in the different services you offer on the platform. These include services such as logging, monitoring, ingress, etc. A significant determining factor in implementing this isolation is the technology you use to provide the service. In some cases, the tool or technology might support multitenancy out of the box, vastly simplifying your implementation.

Another important consideration to make is whether you *need* to isolate tenants at this layer. Is it okay for tenants to look at each other's logs and metrics? Is it acceptable for them to freely discover each other's services over DNS? Can they share the ingress data path? Answering these and similar questions will help clarify your requirements. In the end, it boils down to the level of trust between the tenants you are hosting on the platform.

A typical scenario we run into when helping platform teams is multitenant monitoring with Prometheus. Out of the box, Prometheus does not support multitenancy. Metrics are ingested and stored in a single time-series database, which is accessible by anyone who has access to the Prometheus HTTP endpoint. In other words, if the Prometheus instance is scraping metrics from multiple tenants, there's no way to prevent different tenants from seeing each other's data. To address this issue, we need to deploy separate Prometheus instances per tenant.

When approaching this problem, we typically leverage the prometheus-operator. As discussed in Chapter 9, the prometheus-operator allows you to deploy and manage multiple instances of Prometheus using Custom Resource Definitions. With this capability, you can offer a monitoring platform service that can safely support various tenants. Tenants are completely isolated as they get a dedicated monitoring stack that includes Prometheus, Grafana, Alertmanager, etc.

Depending on the platform's target user experience, you can either allow tenants to deploy their Prometheus instance using the operator, or you can automatically create an instance when you onboard a new tenant. When the platform team has the capacity, we recommend the latter, as it removes the burden from the platform tenants and provides an improved user experience.

Centralized logging is another platform service that you can implement with multitenancy in mind. Typically, this involves sending logs for different tenants to different backends or datastores. Most log forwarders have routing features that you can use to implement a multitenant solution.

In the case of Fluentd and Fluent Bit, you can leverage their tag-based routing features when configuring the forwarder. The following snippet shows a sample Fluent Bit output configuration that routes Alice's logs (Pods in the `alice-ns` Namespace) to one backend and Bob's logs (Pods in the `bob-ns` Namespace) to another backend:

```
[OUTPUT]
    Name            es
    Match           kube.var.log.containers.**alice-ns**.log
    Host            alice.es.internal.cloud.example.com
    Port            ${FLUENT_ELASTICSEARCH_PORT}
    Logstash_Format On
    Replace_Dots    On
    Retry_Limit     False

[OUTPUT]
    Name            es
    Match           kube.var.log.containers.**bob-ns**.log
    Host            bob.es.internal.cloud.example.com
    Port            ${FLUENT_ELASTICSEARCH_PORT}
    Logstash_Format On
    Replace_Dots    On
    Retry_Limit     False
```

In addition to isolating the logs at the backend, you can also implement rate-limiting or throttling to prevent one tenant from hogging the log forwarding infrastructure. Both Fluentd and Fluent Bit have plug-ins you can use to enforce such limits. Finally, if you have a use case that warrants it, you can leverage a logging operator to support more advanced use cases, such as exposing the logging configuration via a Kubernetes CRD.

Multitenancy in the platform services layer is sometimes overlooked by platform teams. As you build your multitenant platform, consider your requirements and their implications on the platform services you want to offer. In some cases, it can drive decisions around approaches and tooling that are fundamental to your platform.

Summary

Workload tenancy is a crucial concern you must consider when building a platform atop Kubernetes. On one hand, you can operate single-tenant clusters for each of your platform tenants. While this approach is viable, we discussed its downsides, including resource and management overhead. The alternative is multitenant clusters, where tenants share the cluster's control plane, workload plane, and platform services.

When hosting multiple tenants on the same cluster, you must ensure tenant isolation such that tenants cannot negatively affect each other. We discussed the Kubernetes Namespace as the foundation upon which we can build the isolation. We then discussed many of the isolation mechanisms available in Kubernetes that allow you to build a multitenant platform. These mechanisms are available in different layers, mainly the control plane, the workload plane, and the platform services.

The control plane isolation mechanisms include RBAC to control what tenants can do, resource quotas to divvy up the cluster resources, and admission webhooks to enforce policy. On the workload plane, you can segregate tenants by using Resource Requests and Limits to ensure fair-sharing of node resources, Network Policies to segment the Pod network, and Pod Security Policies to limit Pods capabilities. Finally, when it comes to platform services, you can leverage different technologies to implement multitenant offerings. We explored monitoring and centralized logging as example platform service that you can build to support multiple tenants.

CHAPTER 13
Autoscaling

The ability to automatically scale workload capacity is one of the compelling benefits of cloud native systems. If you have applications that encounter significant changes in capacity demands, autoscaling can reduce costs and reduce engineering toil in managing those applications. Autoscaling is the process whereby we increase and decrease the capacity of our workloads without human intervention. This begins with leveraging metrics to provide an indicator for when application capacity should be scaled. It includes tuning settings that respond to those metrics. And it culminates in systems to actually expand and contract the resources available to an application to accommodate the work it must perform.

While autoscaling can provide wonderful benefits, it's important to recognize when you should *not* employ autoscaling. Autoscaling introduces complexity into your application management. Besides initial setup, you will very likely need to revisit and tune the configuration of your autoscaling mechanisms. Therefore, if an application's capacity demands do not change markedly, it may be perfectly acceptable to provision for the highest traffic volumes an app will handle. If your application load alters at predictable times, the manual effort to adjust capacity at those times may be trivial enough that investing in autoscaling may not be justified. As with virtually all technology, leverage them only when the long-term benefit outweighs the setup and maintenance of the system.

We're going to divide the subject of autoscaling into two broad categories:

Workload autoscaling
: The automated management of the capacity for individual workloads

Cluster autoscaling
: The automated management of the capacity of the underlying platform that hosts workloads

As we examine these approaches, keep in mind the common primary motivations for autoscaling:

Cost management
: This is most relevant when you are renting your servers from a public cloud provider or being charged internally for your usage of virtualized infrastructure. Cluster autoscaling allows you to dynamically adjust the number of machines you pay for. In order to achieve this elasticity in your infrastructure you will need to leverage workload autoscaling to manage the capacity of the relevant applications within the cluster.

Capacity management
: If you have a static set of infrastructure to leverage, autoscaling gives you an opportunity to dynamically manage the allocation of that fixed capacity. For example, an application that provides services to your business's end users will often have days and times when it is busiest. Workload autoscaling allows an application to dynamically expand its capacity and consume large amounts of a cluster when needed. It also allows it to contract and make room for other workloads. Perhaps you have batch workloads that can take advantage of the unused compute resources during off hours. Cluster autoscaling can remove considerable human toil in managing compute infrastructure capacity since the number of machines used by your clusters is adjusted without human intervention.

Autoscaling is compelling for applications that fluctuate in load and traffic. Without autoscaling, you have two options:

- Chronically overprovision your application capacity, incurring additional cost to your business.
- Alert your engineers for manual scaling operations, incurring additional toil in your operations.

In this chapter, we will first explore how to approach autoscaling, and how to design software to leverage these systems. Then we will dive into details of specific systems we can use to autoscale our applications in Kubernetes-based platforms. This will include horizontal and vertical autoscaling, including the metrics we should use to trigger scaling events. We will also look at scaling workloads in proportion to the cluster itself, as well as an example of custom autoscaling you might consider. Finally, in "Cluster Autoscaling" on page 389, we will address the scaling of the platform itself so that it can accommodate significant changes in demand from the workloads it hosts.

Types of Scaling

In software engineering, scaling generally falls into two categories:

Horizontal scaling
: This involves changing the number of identical replicas of a workload. This is either the number of Pods for a particular application or the number of nodes in a cluster that hosts applications. Future references to horizontal scaling will use the terms "out" or "in" when referring to increasing or decreasing the number of Pods or nodes.

Vertical scaling
: This involves altering the resource capacity of a single instance. For an application, this is changing the resource requests and/or limits for the containers of the application. For nodes of a cluster, this generally involves changing the amount of CPU and memory resources available. Future references to vertical scaling will use the terms "up" or "down" to refer to these changes.

In systems that have a need to dynamically scale, i.e., have frequent, significant changes in load, prefer horizontal scaling where possible. Vertical scaling is limited by the largest machine you have available to use. Furthermore, increasing capacity with vertical scaling involves a restart for the application. Even in virtualized environments where dynamic scaling of machines is possible, Pods will need to be restarted as resource requests and limits cannot be dynamically updated at this time. Compare this with horizontal scaling, where existing instances need not restart and capacity is dynamically increased by adding replicas.

Application Architecture

The topic of autoscaling is particularly important to service-oriented systems. One of the benefits of decomposing applications into distinct components is the ability to scale different parts of an application independently. We were doing this with n-tier architectures well before cloud native emerged. It became commonplace to separate web applications from their relational databases and scale the web app independently. With microservice architecture we can extend this further. For example, an enterprise website may have a service that powers its online store, which is distinct from a service that serves blog posts. When there is a marketing event, the online store can be scaled while the blog service is not affected and may remain unchanged.

With this opportunity to scale different services independently, you are able to more efficiently utilize the infrastructure used by your workloads. However, you introduce the management overhead of scaling many distinct workloads. Automating this scaling process becomes very imporant. At a certain point, it becomes essential.

Autoscaling lends itself well to smaller, more nimble workloads that have tiny image sizes and fast startup times. If the time required to pull a container image onto a given node is short, and if the time it takes for the application to start once the container is created is also short, the workload can respond to scaling events quickly.

Capacity can be adjusted much more readily. Applications with image sizes over a gigabyte and startup scripts that run for minutes are far less suited to responding to changes in load. Workloads like this are not good candidates for autoscaling, so keep this in mind when designing and building your apps.

It's also important to recognize that autoscaling will involve stopping instances of the app. This doesn't apply when workloads scale out, of course. However, that which scales out, must scale back in. That will involve stopping running instances. And with vertically scaled workloads, restarts are required to update resource allocations. In either case, your application's ability to gracefully shut down will be important. Chapter 14 covers this topic in detail.

Now that we've addressed the design concerns to keep in mind, let's dive into the details of autoscaling workloads in Kubernetes clusters.

Workload Autoscaling

This section will focus on autoscaling application workloads. This involves monitoring some metric and adjusting workload capacity without human intervention. While this sounds like a set-it-and-forget-it operation, don't treat it that way, especially in initial stages. Even after you load test your autoscaling configurations, you need to ensure the behavior you get in production is what you intended. Load tests don't always mimic production conditions accurately. So once in production, you will want to check in on the application to verify that it is scaling at the right thresholds and your objectives for efficiency and end-user experience are being met. Strongly consider setting alerts so that you get notified of significant scaling events to review and tweak its behavior as needed.

Most of this section will address the Horizontal Pod Autoscaler and the Vertical Pod Autoscaler. These are the most common tools used for autoscaling workloads on Kubernetes. We'll also dig into the metrics your workload uses to trigger scaling events and when you should consider custom application metrics for this purpose. We'll also look at the Cluster Proportional Autoscaler and the use cases where that makes sense. Lastly, we'll touch on custom methods beyond these particular tools you might consider.

Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler (HPA) is the most common tool used for autoscaling workloads in Kubernetes-based platforms. It is natively supported by Kubernetes with the HorizontalPodAutoscaler resource and a controller bundled into the kube-controller-manager. If you are using CPU or memory consumption as a metric for autoscaling your workload, the barrier to entry is low for using the HPA.

In this case, you can use the Kubernetes Metrics Server to make the PodMetrics available to the HPA. The Metrics Server collects CPU and memory usage metrics for containers from the kubelets in the cluster and makes them available through the resource metrics API in PodMetrics resources. The Metrics Server leverages the Kubernetes API aggregation layer. Requests for resources in the API group and version `metrics.k8s.io/v1beta1` will be proxied to the Metrics Server.

Figure 13-1 illustrates how the components carry out this function. The Metrics Server collects resource usage metrics for the containers on the platform. It gets this data from the kubelets running on each node in the cluster and makes that data available to clients that need to access it. The HPA controller queries the Kubernetes API server to retrieve that resource usage data every 15 seconds, by default. The Kubernetes API proxies the requests to the Metrics Server, which serves the requested data. The HPA controller maintains a watch on the HorizontalPodAutoscaler resource type and uses the configuration defined there to determine if the number of replicas for an application is appropriate. Example 13-1 demonstrates how this determination is made. The app is most commonly defined with a Deployment resource, and, when the HPA controller determines that the replica count needs to be adjusted, it updates the relevant Deployment through the API server. Subsequently, the Deployment controller responds by updating the ReplicaSet, which leads to a change in the number of Pods.

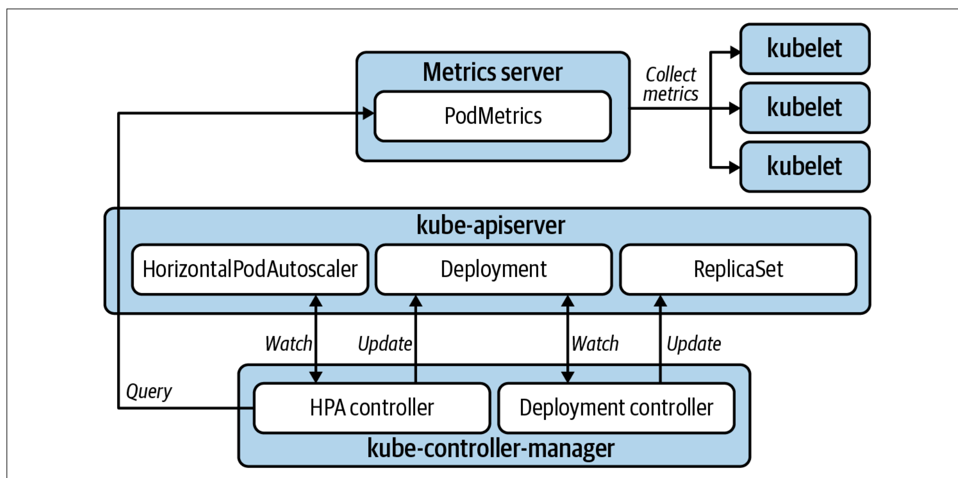


Figure 13-1. Horizontal Pod autoscaling.

The desired state for an HPA is declared in the HorizontalPodAutoscaler resource, as demonstrated in the following example. The `targetCPUUtilizationPercentage` is used to determine replica count for the target workload.

Example 13-1. An example of Deployment and HorizontalPodAutoscaler manifests

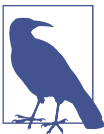
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample
spec:
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - name: sample
        image: sample-image:1.0
        resources:
          requests:
            cpu: "100m"  ❶
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: sample
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sample
  minReplicas: 1  ❷
  maxReplicas: 3  ❸
  targetCPUUtilizationPercentage: 75  ❹
```

❶ A `resources.requests` value must be set for the metric being used.

❷ The replicas will never scale in below this value.

❸ The replicas will never scale out beyond this value.

❹ The desired CPU utilization. If the actual utilization goes significantly beyond this value, the replica count will be increased; if significantly below, decreased.



If you have a use case to use multiple metrics, e.g., CPU *and* memory, to trigger scaling events, you can use the `autoscaling/v2beta2` API. In this case, the HPA controller will calculate the appropriate number of replicas based on each metric individually, and then apply the highest value.

This is the most common and readily used autoscaling method, is widely applicable, and is relatively uncomplicated to implement. However, it's important to understand the limitations of this method:

Not all workloads can scale horizontally
: For applications that cannot share load among distinct instances, horizontal scaling is useless. This is true for some stateful workloads and leader-elected applications. For these use cases you may consider vertical Pod autoscaling.

The cluster size will limit scaling
: As an application scales out, it may run out of capacity available in the worker nodes of a cluster. This can be solved by provisioning sufficient capacity ahead of time, using alerts to prompt your platform operators to add capacity manually, or by using cluster autoscaling, which is discussed in another section of this chapter.

CPU and memory may not be the right metric to use for scaling decisions
: If your workload exposes a custom metric that better identifies a need to scale, it can be used. We will cover that use case later in this chapter.

Avoid autoscaling your workload based on a metric that does not always change in proportion to the load placed on the application. The most common autoscaling metric is CPU. However, if a particular workload's CPU doesn't change significantly with added load, and instead consumes memory in more direct proportion to increased load, do not use CPU. A less obvious example is if a workload consumes added CPU at startup. During normal operation, CPU may be a perfectly useful trigger for autoscaling. However, a startup CPU spike will be interpreted by the HPA as a trigger for a scaling event even though traffic has not induced the spike. There are ways to mitigate this with kube-controller-manager flags such as `--horizontal-pod-autoscaler-cpu-initialization-period`, which will provide a startup grace period, or the `--horizontal-pod-autoscaler-sync-period`, which allows you to increase the time between scaling evaluations. But note that these flags are set on the kube-controller-manager. These will affect all HPAs across the entire cluster, which will impact workloads that do *not* have high startup CPU consumption. You could wind up reducing the responsiveness of the HPA for workloads cluster-wide. If you find your team employing workarounds to make CPU consumption work as a trigger for your autoscaling needs, consider using a more representative custom metric. Perhaps number of HTTP requests received would make a better measuring stick, for example.

That wraps up the Horizontal Pod Autoscaler. Next, we'll look at another form of autoscaling available in Kubernetes: vertical Pod autoscaling.

Vertical Pod Autoscaler

For reasons covered earlier in "Types of Scaling" on page 378, vertically scaling workloads is a less common requirement. Furthermore, automating vertical scaling is more complex to implement in Kubernetes. While the HPA is included in core Kubernetes, the VPA needs to be implemented by deploying three distinct controller components in addition to the Metrics Server. For these reasons, the Vertical Pod Autoscaler (VPA) is less commonly used than the HPA.

The VPA consists of three distinct components:

Recommender
: Determines optimum container CPU and/or memory request values based on usage in the PodMetrics resource for the Pod in question.

Admission plug-in
: Mutates the resource requests and limits on new Pods when they are created based on the recommender's recommendation.

Updater
: Evicts Pods so that they may have updated values applied by the admission plug-in.

Figure 13-2 illustrates the interaction of components with the VPA.

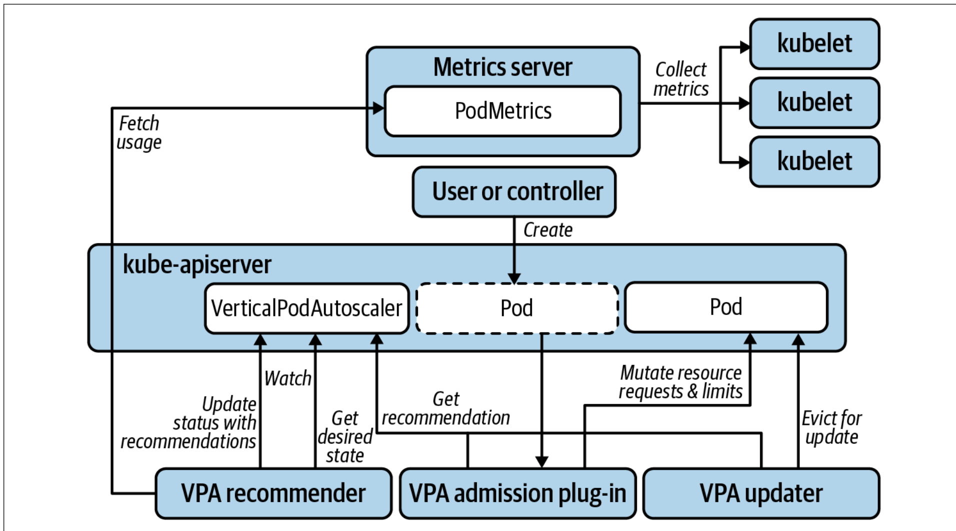


Figure 13-2. Vertical Pod autoscaling.

The desired state for a VPA is declared in the VerticalPodAutoscaler resource as demonstrated in Example 13-2.

Example 13-2. A Pod resource and the VerticalPodAutoscaler resource that configures vertical autoscaling

```
apiVersion: v1
kind: Pod
metadata:
  name: sample
spec:
  containers:
  - name: sample
    image: sample-image:1.0
    resources:  ❶
      requests:
        cpu: 100m
        memory: 50Mi
      limits:
        cpu: 100m
        memory: 50Mi
---
apiVersion: "autoscaling.k8s.io/v1beta2"
kind: VerticalPodAutoscaler
metadata:
  name: sample
spec:
  targetRef:
    apiVersion: "v1"
    kind: Pod
    name: sample
  resourcePolicy:
    containerPolicies:
      - containerName: '*'  ❷
        minAllowed:  ❸
          cpu: 100m
          memory: 50Mi
        maxAllowed:  ❹
          cpu: 1
          memory: 500Mi
        controlledResources: ["cpu", "memory"]  ❺
  updatePolicy:
    updateMode: Recreate  ❻
```

❶ The VPA will maintain the requests:limit ratio when updating values. In this guaranteed QOS example, any change to requests will result in an identical change to the limits.

❷ This scaling policy will apply to every container—just one in this example.

❸ Resources requests will not be set below these values.

❹ Resources requests will not be set above these values.

❺ Specifies the resources being autoscaled.

❻ There are three `updateMode` options. `Recreate` mode will activate autoscaling. `Initial` mode will apply admission control to set resource values when created, but will never evict any Pods. `Off` mode will recommend resource values but never automatically change them.

We very rarely see the VPA in full `Recreate` mode in the field. However, using it in `Off` mode can also be valuable. While comprehensive load testing and profiling of applications is recommended and preferable before they go to production, that's not always the reality. In corporate environments with deadlines, workloads are often deployed to production before the resource consumption profile is well understood. This commonly leads to overrequested resources as a safety measure, which often results in poor utilization of infrastructure. In these cases, the VPA can be used to recommend values that are then evaluated and manually updated by engineers once production load has been applied. This gives them peace of mind that workloads will not be evicted at peak usage times, which is particularly important if an app does not yet gracefully shut down. But, because the VPA recommends values, it saves some of the toil in reviewing resource usage metrics and determining optimum values. In this use case, it is not an autoscaler, but rather a resource tuning aid.

To get recommendations from a VPA in `Off` mode, run `kubectl describe vpa <vpa name>`. You will get an output similar to Example 13-3 under the `Status` section.

Example 13-3. Vertical Pod Autoscaler recommendation

```
Recommendation:
  Container Recommendations:
    Container Name:  coredns
    Lower Bound:
      Cpu:     25m
      Memory:  262144k
    Target:
      Cpu:     25m
      Memory:  262144k
    Uncapped Target:
      Cpu:     25m
      Memory:  262144k
    Upper Bound:
      Cpu:     427m
      Memory:  916943343
```

It will provide recommendations for each container. Use the `Target` value as a baseline recommendation for the CPU and memory requests.

Autoscaling with Custom Metrics

If CPU and memory consumption are not good metrics by which to scale a particular workload, you can leverage custom metrics as an alternative. We can still use tools like the HPA. However, we will change the source of metrics used to trigger the autoscaling. The first step is to expose the appropriate custom metrics from your application. Chapter 14 addresses how to go about doing this.

Next, you will need to expose the custom metrics to the autoscaler. This will require a custom metrics server that will be used instead of the Kubernetes Metrics Server that we looked at earlier. Some vendors, such as Datadog, provide systems to do this in Kubernetes. You can also do it with Prometheus, assuming you have a Prometheus server that is scraping and storing the app's custom metrics, which is covered in Chapter 10. In this case, we can use the Prometheus Adapter to serve the custom metrics.

The Prometheus Adapter will retrieve the custom metrics from Prometheus' HTTP API and expose them through the Kubernetes API. Like the Metrics Server, the Prometheus Adapter uses Kubernetes API aggregation to instruct Kubernetes to proxy requests for metrics APIs to the Prometheus Adapter. In fact, in addition to the custom metrics API, the Prometheus Adapter implements the resource metrics API that allows you to entirely replace the Metrics Server functionality with the Prometheus Adapter. Additionally, it implements the external metrics API that offers the opportunity to scale an application based on metrics external to the cluster.

When leveraging custom metrics for horizontal autoscaling, Prometheus scrapes those metrics from your app. The Prometheus Adapter gets those metrics from Prometheus and exposes them through the Kubernetes API server. The HPA queries those metrics and scales your application accordingly, as shown in Figure 13-3.

While leveraging custom metrics in this way introduces some added complexity, if you are already exposing useful metrics from your workloads and using Prometheus to monitor them, replacing Metrics Server with the Prometheus Adapter is not a huge leap. And the additional autoscaling opportunities it opens up make it well worth considering.

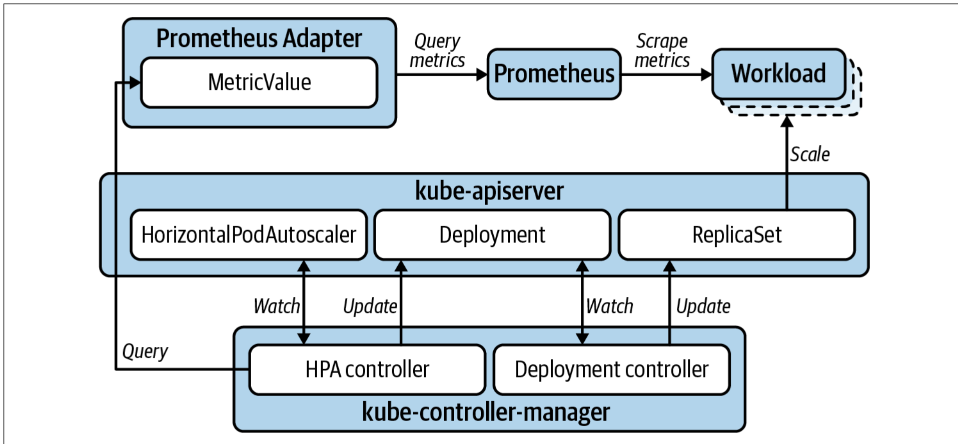


Figure 13-3. Horizontal Pod autoscaling with custom metrics.

Cluster Proportional Autoscaler

The Cluster Proportional Autoscaler (CPA) is a horizontal workload autoscaler that scales replicas based on the number of nodes (or a subset of nodes) in the cluster. So, unlike the HPA, it does not rely on any of the metrics APIs. Therefore, it does not have a dependency on the Metrics Server or Prometheus Adapter. Also, it is not configured with a Kubernetes resource, but rather uses flags to configure target workloads and a ConfigMap for scaling configuration. Figure 13-4 illustrates the CPA's much simpler operational model.

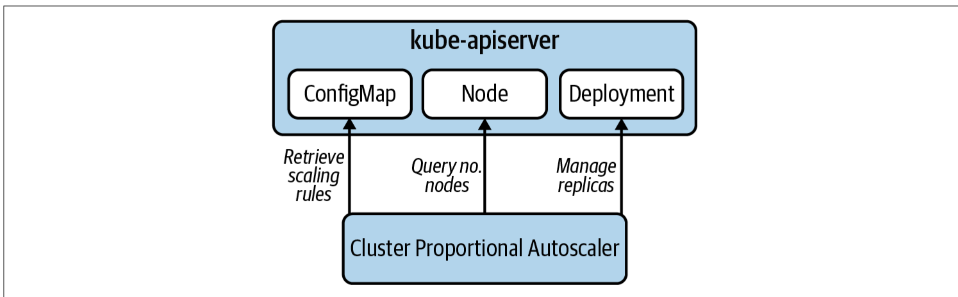


Figure 13-4. Cluster proportional autoscaling.

The CPA has a narrower use case. Workloads that need to scale in proportion to the cluster are generally limited to platform services. When considering the CPA, evaluate whether an HPA would provide a better solution, especially if you are already leveraging the HPA with other workloads. If you are already using HPAs, you will have the Metrics Server or Prometheus Adapter already deployed to implement the necessary metrics APIs. So deploying another autoscaler, and the maintenance overhead that goes with it, may not be the best option. Alternatively, in a cluster where HPAs

are *not* already in use, and the CPA provides the functionality you need, it becomes more attractive due to its simple operational model.

There are two scaling methods used by the CPA:

- The linear method scales your application in direct proportion to how many nodes or cores are in the cluster.
- The ladder method uses a step function to determine the proportion of nodes:replicas and/or cores:replicas.

We have seen the CPA used with success for services like cluster DNS where clusters are allowed to scale to hundreds of worker nodes. In cases such as this, the traffic and demand for a service at 5 nodes is going to be drastically different than at 300 nodes, so this approach can be quite useful.

Custom Autoscaling

On the subject of workload autoscaling, so far we've discussed some specific tools available in the community: the HPA, VPA, and CPA along with the Metrics Server and Prometheus Adapter. But autoscaling your workloads is not limited to this tool set. Any automated method you can employ that implements the scaling behavior you require falls into the same category. For example, if you know the days and times when traffic increases for your application, you can implement something as simple as a Kubernetes CronJob that updates the replica count on the relevant Deployment. In fact, if you can leverage a simple, straightforward method such as this, lean toward the simpler solution. A system with fewer moving parts is less likely to produce unexpected results.

This wraps up the approaches to autoscaling workloads. We've looked at several ways to approach this using core Kubernetes, community-developed add-on components, and custom solutions. Next, we're going to look at autoscaling the substrate that hosts these workloads: the Kubernetes cluster itself.

Cluster Autoscaling

The Kubernetes Cluster Autoscaler (CA) provides an automated solution for horizontally scaling the worker nodes in your cluster. It provides a solution to one of the limitations of HPAs and can alleviate significant toil around capacity and cost management for your Kubernetes infrastructure.

As platform teams adopt your Kubernetes-based platform, you will need to manage the clusters' capacity as new tenants are onboarded. This can be a manual, routine review process. It can also be alert-driven, whereby you use alerting rules on usage metrics to notify you of situations where workers need to be added or removed. Or

you can fully automate the operation such that you can simply add and remove tenants and let the CA manage the cluster scaling to accommodate.

Furthermore, if you are leveraging workload autoscaling with significant fluctuation in resource consumption, the story for CA becomes even more compelling. As load increases on an HPA-managed workload, its replica count will increase. If you run out of compute resources in your cluster, some of the Pods will not be scheduled and remain in a `Pending` state. CA looks for this exact condition, calculates the number of nodes needed to satisfy the shortage, and adds new nodes to your cluster. The diagram in Figure 13-5 shows the cluster scaling out to accommodate a horizontally scaling application.

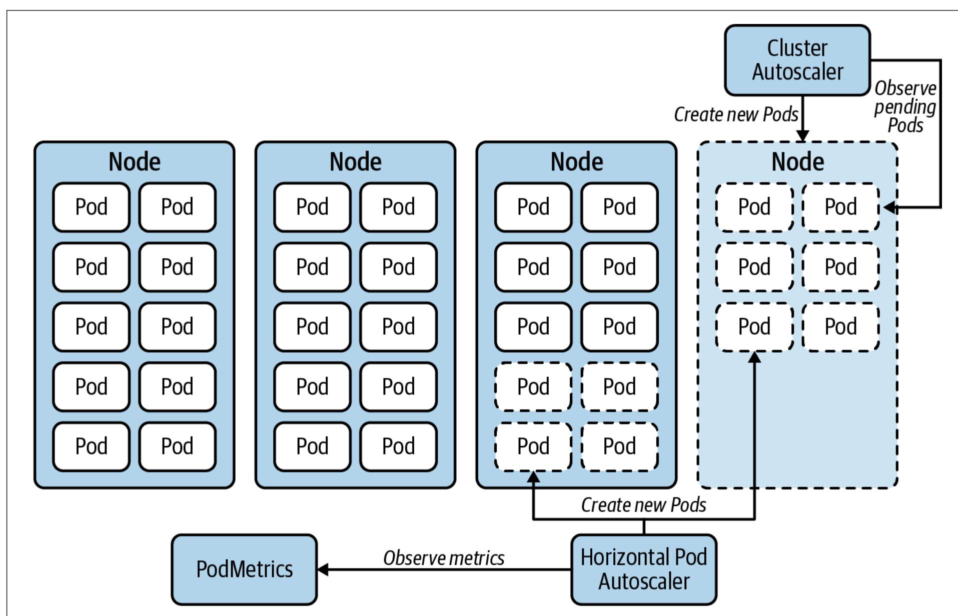


Figure 13-5. Cluster Autoscaler scaling out nodes in response to Pod replicas scaling out.

On the other side of the coin, when load reduces and the HPA scales in the Pods for an application, the CA will look for nodes that have been underutilized for an extended period. If the Pods on the underutilized nodes can be rescheduled to other nodes in the cluster, the CA will deprovision the underutilized nodes to scale the cluster in.

One thing to keep in mind when you invoke this dynamic management of worker nodes is that it will inevitably shuffle the distribution of Pods across your nodes. The Kubernetes scheduler will generally spread Pods evenly around your worker nodes when they are first created. However, once a Pod is running, the scheduling decision that determined its home will not be reevaluated unless it is evicted. So when a particular application horizontally scales out and then back in, you may end up with Pods

unevenly spread across your worker nodes. In some cases you may end up with many replicas for a Deployment clustered on just a few nodes. If this presents a threat to a workload's node failure tolerance, you can use the Kubernetes descheduler to evict them according to different policies. Once evicted, the Pods will be rescheduled. This will help rebalance their distribution across nodes. We have not found many cases where there was a genuine compelling need to do this, but it is an available option.

As you might imagine, there are infrastructure management concerns to plan for if you are considering cluster autoscaling. Firstly, you will need to use one of the supported cloud providers that are documented in the project repo. Next you will have to give permissions to CA to create and destroy machines for you.

These infrastructure management concerns change somewhat if you use the CA with the Cluster API project. Cluster API uses its own Kubernetes operators to manage cluster infrastructure. In this case, instead of connecting directly with the cloud provider to add and remove worker nodes, CA offloads this operation to Cluster API. The CA simply updates the replicas in a `MachineDeployment` resource, which is reconciled by Cluster API controllers. This removes the need to use a cloud provider that's compatible with CA (however, you *will* need to check whether there is a Cluster API provider for your cloud provider). The permissions issue is also offloaded to Cluster API components. This is a better model in many ways. However, Cluster API is commonly implemented using management clusters. This introduces external dependencies for cluster autoscaling that should be considered. This topic is covered further in "Management clusters" on page 33.

The scaling behavior of CA is quite configurable. The CA is configured using flags that are documented in the project's FAQ on GitHub. Example 13-4 shows a CA Deployment manifest for AWS and includes examples of how to set some common flags.

Example 13-4. CA Deployment manifest targeting an Amazon Web Services autoscaling group

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aws-cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: "aws-cluster-autoscaler"
  template:
    metadata:
      labels:
        app.kubernetes.io/name: "aws-cluster-autoscaler"
    spec:
```

```
    containers:
      - name: aws-cluster-autoscaler
        image: "us.gcr.io/k8s-artifacts-prod/autoscaling/cluster-autoscaler:v1.18"
        command:
          - ./cluster-autoscaler
          - --cloud-provider=aws  ❶
          - --namespace=kube-system
          - --nodes=1:10:worker-auto-scaling-group  ❷
          - --logtostderr=true
          - --stderrthreshold=info
          - --v=4
        env:
          - name: AWS_REGION
            value: "us-east-2"
        livenessProbe:
          httpGet:
            path: /health-check
            port: 8085
        ports:
          - containerPort: 8085
```

❶ Configures the supported cloud provider; AWS in this case.

❷ This flag configures the CA to update an AWS autoscaling group called `worker-auto-scaling-group`. It allows CA to scale the number of machines in this group between 1 and 10.

Cluster autoscaling can be extremely useful. It unlocks one of the compelling benefits offered by cloud native infrastructure. However, it introduces nontrivial complexity. Ensure you load test and understand well how the system will behave before you rely on it to autonomously manage the scaling of business-critical platforms in production. One important consideration is to clearly understand the upper limits that you will be reaching with your cluster. If your platform hosts significant workload capacity and you allow your cluster to scale to hundreds of nodes, understand where it will scale to before components of the platform start to introduce bottlenecks. More discussion around cluster sizing can be found in Chapter 2.

Another consideration with cluster autoscaling is the speed at which your clusters will scale when the need arises. This is where overprovisioning may help.

Cluster Overprovisioning

It's important to remember that Cluster Autoscaler responds to `Pending` Pods that couldn't be scheduled due to insufficient compute resources in the cluster. So at the moment the CA takes action to scale out the cluster nodes, your cluster is already full. This means that, if not managed properly, your scaling workloads could suffer from a shortage of capacity for the time it takes for new nodes to become available for scheduling. This is where cluster-overprovisioner can help.

First it's important to understand how long it takes for new nodes to spin up, join the cluster, and become ready to accept workloads. Once this is understood, you can address the best solution for your situation:

- Set the target utilization in your HPAs sufficiently low so that your workloads are scaled out well before the application is at full capacity. This could provide the buffer that allows for time to provision nodes. It relieves the need for overprovisioning the cluster, but if you need to account for particularly sharp increases in load, you may need to set that target utilization too low to guard against capacity shortages. This leads to a situation where you have chronically overprovisioned workload capacity to account for rare events.
- Another solution is to use cluster overprovisioning. With this method, you put empty nodes on standby to provide the buffer for workloads that are scaling out. This will relieve the need to set target utilization on HPAs artificially low in preparation for high load events.

Cluster overprovisioning works by deploying Pods that do the following:

- Request enough resources to reserve virtually all resources for a node
- Consume no actual resources
- Use a priority class that causes them to be evicted as soon as any other Pod needs it

With the resource requests on the overprovisioner Pod set to reserve an entire node, you can then adjust the number of standby nodes with the number of replicas on the overprovisioner Deployment. Overprovisioning for a particular event or marketing campaign can be achieved by simply increasing the number of replicas on the overprovisioner Deployment.

Figure 13-6 illustrates what this looks like. This illustration shows just a single Pod replica, but it can be as many as you need to provide adequate buffer for scaling events.

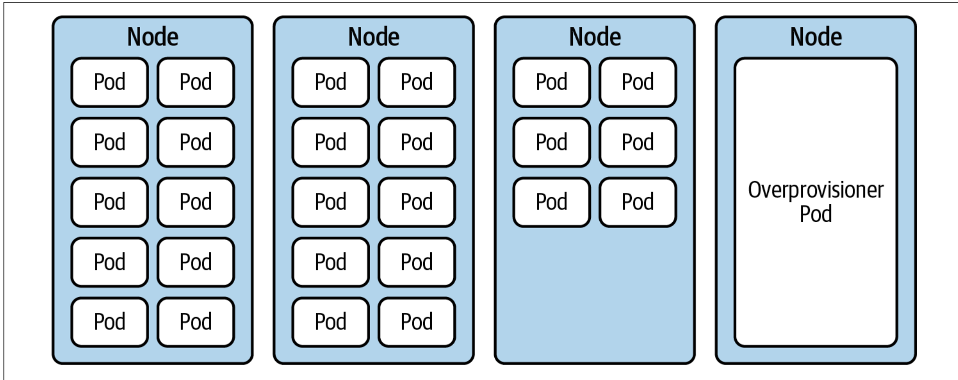


Figure 13-6. Cluster overprovisioning.

The node occupied by the overprovisioner Pod is now on standby for whenever it becomes needed by another Pod in the cluster. You can accomplish this by creating a priority class with `value: -1` and then applying this to the overprovisioner Deployment. This will make all other workloads a higher priority by default. Should a Pod from another workload need the resources, the overprovisioner Pod will be immediately evicted, making way for the scaling workload. The overprovisioner Pod will go into a `Pending` state, which will trigger the Cluster Autoscaler to provision a new node to sit on standby, as shown in Figure 13-7.

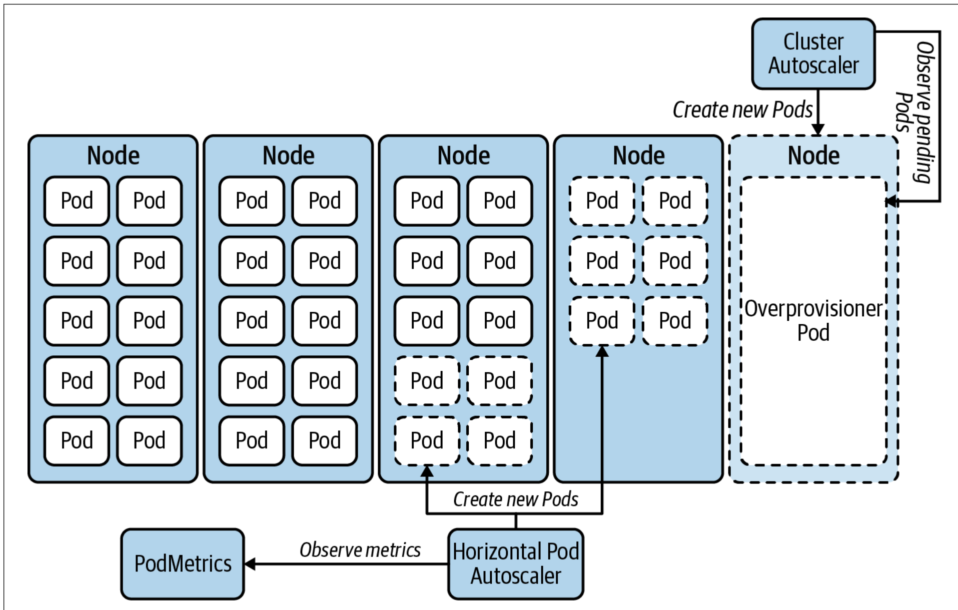


Figure 13-7. Scaling out with cluster-overprovisioner.

With Cluster Autoscaler and cluster-overprovisioner, you have effective mechanisms to horizontally scale your Kubernetes clusters, which dovetails very nicely with horizontally scaling workloads. We haven't covered vertically scaling clusters here because we have not found a use for it that isn't solved by horizontal scaling.

Summary

If you have applications that are subject to significant changes in capacity requirements, lean toward using horizontal scaling, if possible. Develop the apps that you will autoscale to play nicely with being stopped and started frequently and expose custom metrics if CPU or memory are not good metrics to trigger scaling. Test your autoscaling to ensure it behaves as you expect to optimize efficiency and end-user experience. If your workloads will scale beyond the capacity of your cluster, consider autoscaling the cluster itself. And if your scaling events are particularly sharp, consider putting nodes on standby with cluster-overprovisioner.

CHAPTER 14

Application Considerations

Kubernetes is rather flexible when it comes to the type of applications it can run and manage. Barring operating system and processor type limitations, Kubernetes can essentially run anything. Large monoliths, distributed microservices, batch workloads, you name it. The only requirement that Kubernetes imposes on workloads is that they are distributed as container images. With that said, there are certain steps you can take to make your applications better Kubernetes citizens.

In this chapter, we will pivot our discussions to focus on the application instead of the platform. If you are part of a platform team, don't skip this chapter. While you might think it only applies to developers, it also applies to you. As a platform team member, you will most likely get to build applications to provide custom services on your platform. Even if you don't, the discussions in this chapter will help you better align with development teams consuming the platform, and even educate those teams that might be unfamiliar with container-based platforms.

This chapter covers various considerations you should make when running applications on Kubernetes. Mainly:

- Deploying applications onto the platform, and mechanisms to manage deployment manifests, such as templating and packaging.
- Approaches to configure applications, such as using Kubernetes APIs (ConfigMaps/Secrets), and integrating with external systems for config and secret management.
- Kubernetes features that improve the availability of your workloads, such as prestop container hooks, graceful termination, and scheduling constraints.
- State probes, a feature of Kubernetes that enables you to surface application health information to the platform.

- Resource requests and limits, which are critical to ensure your applications run properly on the platform.
- Logs, metrics, and tracing as mechanisms to debug, troubleshoot, and operate your workloads effectively.

Deploying Applications to Kubernetes

Once your application is containerized and available in a container image registry, you are ready to deploy it onto Kubernetes. In most cases, deploying the application involves writing YAML manifests that describe the Kubernetes resources required to run the app, such as Deployments, Services, ConfigMaps, CRDs, etc. Then, you send the manifests to the API server, and Kubernetes takes care of the rest. Using raw YAML manifests is a great way to get started, but it can quickly become impractical, especially when deploying the application onto different clusters or environments. You will most likely encounter questions similar to the following:

- How do I provide different credentials when running in staging versus production?
- How can I use a different image registry when deploying in various datacenters?
- How do I set different replica counts in development versus production?
- How can I ensure all port numbers match up across the different manifests?

The list goes on and on. And while you could have multiple sets of manifests to solve for each of these concerns, the permutations make it rather challenging to manage. In this section, we will discuss approaches you can take to address the issue of manifest management. Mainly, we will cover templating manifests and packaging applications for Kubernetes. We will not, however, discuss the gamut of tools available in the community. More often than not, we find that teams get stuck in analysis paralysis when considering the different options. Our advice is to choose *something* and move on to solving higher-value concerns.

Templating Deployment Manifests

Templating involves introducing placeholders in your deployment manifests. Instead of hardcoding values in the manifests, the placeholders provide a mechanism for you to inject values as necessary. For example, the following templated manifest enables you to set replica counts to different values. Perhaps you need one replica in development, but five in production.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
```

Packaging Applications for Kubernetes

Creating self-contained software packages is another mechanism you can use to deploy your application while addressing the manifest management. Packaging solutions usually build upon templating, but they introduce additional functionality that can be useful, such as the ability to push the package to OCI-compatible registries, life cycle management hooks, and more.

Packages are a great mechanism to consume software maintained by a third party or deliver software to third parties. If you've used Helm to install software into a Kubernetes cluster, you've already leveraged the benefits of packaging. If you are unfamiliar with Helm, the following snippet gives you an idea of what it takes to install a package:

```
$ helm repo add hashicorp https://helm.releases.hashicorp.com
"hashicorp" has been added to your repositories

$ helm install vault hashicorp/vault
```

As you can see, packages can be a great way to deploy and manage software on Kubernetes. With that said, packages can fall short when it comes to complex applications that require advanced life cycle management. For such applications, we find operators to be a better solution. We discuss operators extensively in Chapter 2. Even though the chapter focuses on platform services, the concepts discussed apply when building operators for complex applications.

Ingesting Configuration and Secrets

Applications typically have configuration that tells them how to behave at runtime. Configuration commonly includes logging levels, hostnames of dependencies (e.g., DNS record for a database), timeouts, and more. Some of these settings can contain sensitive information, such as passwords, that we usually call secrets. In this section, we will discuss the different methods you can use to configure applications on a Kubernetes-based platform. First, we will review the ConfigMap and Secret APIs available in core Kubernetes. Then, we will explore an alternative to the Kubernetes API, mainly integrating with an external system. Finally, we will provide guidance on these approaches based on what we've seen work best in the field.

Before digging in, it is worth mentioning that you should avoid bundling configuration or secrets inside your application's container image. The tight coupling between the application binary and its configuration defeats the purpose of runtime configuration. Furthermore, it poses a security problem in the case of secrets, as the image might be accessible to actors that should otherwise not have access to the secrets. Instead of including config in the image, you should leverage platform features to inject configuration at runtime.

Kubernetes ConfigMaps and Secrets

ConfigMaps and Secrets are core resources in the Kubernetes API that enable you to configure your applications at runtime. As with any other resource in Kubernetes, they are created via the API server and are usually declared in YAML, such as the following example:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  debug: "false"
```

Let's discuss how you can consume ConfigMaps and Secrets in your applications.

The first method is to mount ConfigMaps and Secrets as files in the Pod's filesystem. When building your Pod specification, you can add volumes that reference ConfigMaps or Secrets by name and mount them into containers at specific locations. For example, the following snippet defines a Pod that mounts the ConfigMap named `my-config` into the container named `my-app` at `/etc/my-app/config.json`:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - image: my-app
    name: my-app:v0.1.0
    volumeMounts:
    - name: my-config
      mountPath: /etc/my-app/config.json
  volumes:
  - name: my-config
    configMap:
      name: my-config
```

Leveraging volume mounts is the preferred method when it comes to consuming ConfigMaps and Secrets. The reason is that the files in the Pod are dynamically updated, which allows you to reconfigure applications without restarting the app or re-creating the Pod. With that said, this is something that the application must support. The application must watch the configuration files on disk and apply new configuration when the files change. Many libraries and frameworks make it easy to implement this functionality. When this is not possible, you can introduce a sidecar container that watches the config files and signals the main process (with a SIGHUP, for example) when new configuration is available.

Consuming ConfigMaps and Secrets via environment variables is another method you can use. If your application expects configuration through environment variables, this is the natural approach to follow. Environment variables can also be helpful if you need to provide settings via command-line flags. In the following example, the Pod sets the `DEBUG` environment variable using a ConfigMap named `my-config`, which has a key called `debug` that contains the value:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app
      image: my-app:v0.1.0
      env:
        - name: DEBUG
          valueFrom:
            configMapKeyRef:
              name: my-config
              key: debug
```

One of the downsides of using environment variables is that changes to the ConfigMaps or Secrets are not reflected in the running Pod until it restarts. This might not be a problem for some applications, but you should keep it in mind. Another downside, mainly for Secrets, is that some applications or frameworks may dump the environment details into logs during startup or when they crash. This poses a security risk as secrets can be leaked into logfiles inadvertently.

These first two ConfigMap and Secret consumption methods rely on Kubernetes injecting the configuration into the workload. Another option is for the application to communicate with the Kubernetes API to get its configuration. Instead of using config files or environment variables, the application reads ConfigMaps and Secrets straight from the Kubernetes API server. The app can also watch the API so that it can act whenever the configuration changes. Developers can use one of the many Kubernetes libraries or SDKs to implement this functionality or leverage application frameworks that support this capability, such as Spring Cloud Kubernetes.

While leveraging the Kubernetes API for application configuration can be convenient, we find that there are important downsides you should consider. First, the need to connect to the API server to get configuration creates a tight coupling between the application and the Kubernetes platform. This coupling raises some interesting questions. What happens if the API server goes down? Will your application experience downtime when your platform team upgrades the API server?

Second, for the application to get its configuration from the API, it needs credentials, and it needs to have the right permissions. These requirements increase your deployment complexity, as you now have to provide a Service Account and define RBAC roles for your workload.

Last, the more applications using this method to get config, the more undue load is imposed on the API server. Since the API server is a critical component of the cluster's control plane, this approach to app configuration can be at odds with the overall scalability of the cluster.

Overall, when it comes to consuming ConfigMaps and Secrets, we prefer using volume mounts and environment variables over integrating with the Kubernetes API directly. In this way, the applications remain decoupled from the underlying platform.

Injecting Workload Metadata

There are certain scenarios where workloads need to get information about themselves. Perhaps they need the Namespace they're running in, their labels, or their resource limits. Kubernetes provides the Downward API, which allows you to inject Pod metadata without the workload having to interact or know about Kubernetes. Similar to ConfigMaps and Secrets, you can provide the metadata via environment variables or volume mounts.

The following example shows the Downward API in action. In this case, the Pod needs to know its memory limit, which is made available as an environment variable named MEM_LIMIT:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app
      image: my-app:0.1.0
      command: [ "my-app" ]
      env:
        - name: MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: my-app
              resource: limits.memory
```

Obtaining Configuration from External Systems

ConfigMaps and Secrets can be convenient when it comes to configuring applications. They are built into the Kubernetes API and are readily available for you to consume. With that said, configuration and secrets have been a concern that application developers had faced well before Kubernetes existed. While Kubernetes provides features to solve this concern, nothing is stopping you from using external systems instead.

One of the most prevalent examples of an external configuration or secrets management system we run into in the field is HashiCorp Vault. Vault provides advanced secret management functionality that is unavailable in Kubernetes Secrets. For example, Vault provides dynamic secrets, secret rotation, time-based tokens, and more. If your application is already leveraging Vault, you can continue to do so when running your application on Kubernetes. Even if not yet using Vault, it is worth evaluating as a more robust alternative to Kubernetes Secrets. We discussed secret management considerations and the Vault integration with Kubernetes extensively in Chapter 7. If you want to learn more about secret management in Kubernetes and the lower-level details of the Vault integration, we recommend you check out that chapter.

When leveraging an external system for configuration or secrets, we find that offloading the integration (as much as possible) to the platform is beneficial. Integrations with external systems such as Vault can be offered as a platform service to expose Secrets as volumes or environment variables in Pods. The platform service abstracts the external system and enables your application to consume the Secret without worrying about the implementation details of the integration. Overall, leveraging

such a platform service reduces the application's complexity and results in standardization across your applications.

Handling Rescheduling Events

Kubernetes is a highly dynamic environment where workloads are moved around for different reasons. Cluster nodes can come and go; they can run out of resources or even fail. Platform teams can drain, cordon, or remove nodes to perform cluster life cycle operations (e.g., upgrades). These are examples of situations in which your workload might be killed and rescheduled, and there are many others.

Regardless of the reason, the dynamic nature of Kubernetes can impact your application's availability and operation. Even though the application's architecture has the highest bearing in determining the impact of disturbances, there are features in Kubernetes you can leverage to minimize that impact. We will explore these features in this section. First, we will dig into pre-stop container life cycle hooks. As indicated by the name, these hooks enable you to act before Kubernetes stops your containers. We will then discuss how you can shut down containers gracefully, which involves handling signals from within the application in response to shutdown events. Finally, we will review Pod anti-affinity rules, a mechanism you can use to spread your application across failure domains. As mentioned before, these mechanisms can help *minimize* the impact of disturbances but cannot eliminate the potential for failure. Keep that in mind as you read through this section.

Pre-stop Container Life Cycle Hook

Kubernetes can terminate workloads for any number of reasons. If you need to perform an action before your container is terminated, you can leverage the pre-stop container life cycle hook. Kubernetes provides two types of hooks. The `exec` life cycle hook runs a command within the container, while the `HTTP` life cycle hook issues an HTTP request against an endpoint you specify (typically the container itself). Which hook to use depends on your specific requirements and what you are trying to achieve.

The pre-stop hook in the Contour Ingress controller is a great example that showcases the power of pre-stop hooks. To avoid dropping in-flight client requests, Contour includes a container pre-stop hook that tells Kubernetes to execute a command before stopping the container. The following snippet from the Contour Deployment YAML file shows the pre-stop hook configuration:

```yaml
# <... snip ...>
    spec:
      containers:
      - command:
        - /bin/contour
        args:
          - envoy
          - shutdown-manager
        image: docker.io/projectcontour/contour:main
        lifecycle:
          preStop:
            exec:
              command:
                - /bin/contour
                - envoy
                - shutdown
# <... snip ...>
```

Container pre-stop hooks enable you to take action before Kubernetes stops your container. They allow you to run commands or scripts that exist within the container but are otherwise not part of the running process. One key consideration to keep in mind is that these hooks are executed only in the face of planned life cycle or rescheduling events. The hooks will not run if a node fails, for example. Furthermore, any action performed as part of the pre-stop hook is governed by the Pod's graceful shutdown period, which we will discuss next.

Graceful Container Shutdown

After executing pre-stop hooks (when provided), Kubernetes initiates the container shutdown process by sending a SIGTERM signal to the workload. This signal lets the container know that it is being stopped. It also starts running down the clock of the termination shutdown period, which is 30 seconds by default. You can tune this period using the `terminationGracePeriodSeconds` field of the Pod specification.

During the graceful termination period, the application can complete any necessary actions before shutting down. Depending on the application, these actions can be persisting data, closing open connections, flushing files to disk, etc. Once done, the application should exit with a successful exit code. The graceful termination is illustrated in Figure 14-1, where we can see the kubelet sending the SIGTERM signal and waiting for the container(s) to terminate within the grace period.

If the application shuts down within the termination period, Kubernetes completes the shutdown process and moves on. Otherwise, it forcefully stops the process by sending a SIGKILL signal. Figure 14-1 also shows this forceful termination toward the bottom right of the diagram.

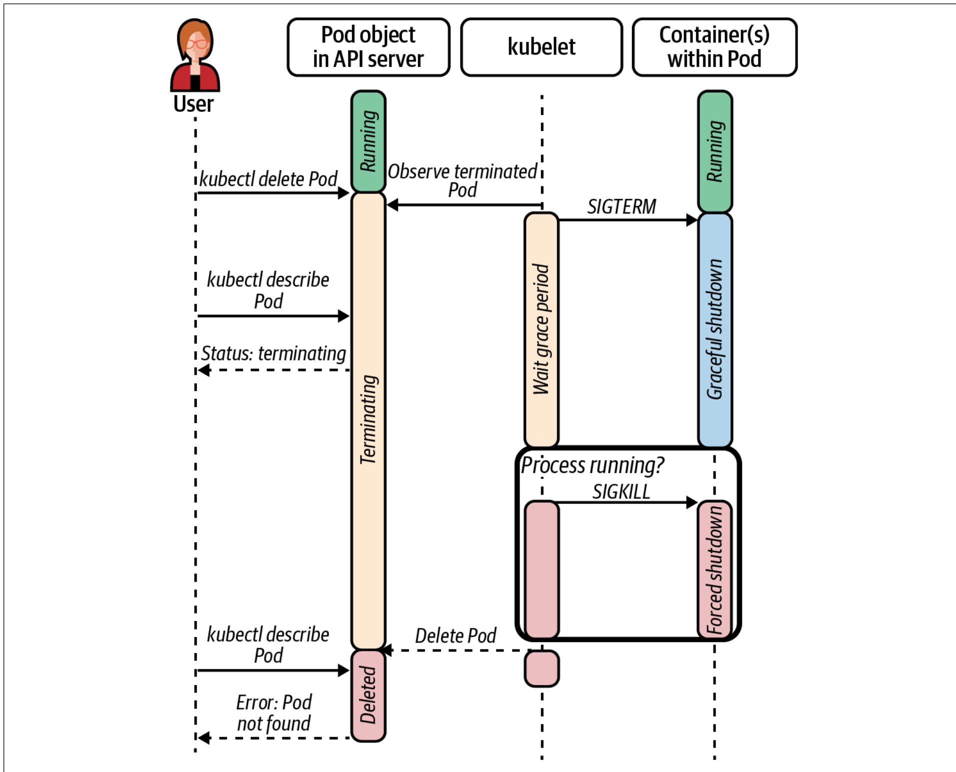


Figure 14-1. Application termination in Kubernetes. The kubelet first sends a SIGTERM signal to the workload and waits up to the configured graceful termination period. If the process is still running after the period expires, the kubelet sends a SIGKILL to terminate the process.

For your application to terminate gracefully, it must handle the SIGTERM signal. Each programming language or framework has its own way of configuring signal handlers. Some application frameworks might even take care of it for you. The following snippet shows a Go application that configures a SIGTERM signal handler, which stops the application's HTTP server upon receipt of the signal:

```
func main() {
        // App initialization code here...
        httpServer := app.NewHTTPServer()

        // Make a channel to listen for an interrupt or terminate signal
        // from the OS.

        // Use a buffered channel because the signal package requires it.
        shutdown := make(chan os.Signal, 1)
        signal.Notify(shutdown, os.Interrupt, syscall.SIGTERM)
```

```
    // Start the application and listen for errors
    errors := make(chan error, 1)
    go httpServer.ListenAndServe(errors)

    // Block main and waiting for shutdown.
    select {
    case err := <-errors:
            log.Fatalf("http server error: %v", err)

    case <-shutdown:
    log.Printf("shutting down http server")
            httpServer.Shutdown()
    }
}
```

When running your applications on Kubernetes, we recommend you configure signal handlers for the SIGTERM signal. Even if there are no shutdown actions to take, handling the signal makes your workload a better Kubernetes citizen, as it reduces the time it takes to stop the application and thus free up the resources for other workloads.

Satisfying Availability Requirements

Container pre-stop hooks and graceful termination are concerned with a single instance or replica of your application. If your application is horizontally scalable, you will most likely have multiple replicas running in the cluster to satisfy availability requirements. Running more than one instance of your workload can provide increased fault tolerance. For example, if a cluster node fails and takes one of the application instances with it, the other replicas can pick up the work. With that said, having multiple replicas does not help if they are running in the same failure domain.

One way to ensure your Pods are spread across failure domains is by using Pod anti-affinity rules. With Pod anti-affinity rules, you tell the Kubernetes scheduler that you want to schedule your Pods according to constraints you define in the Pod definition. More specifically, you ask the scheduler to avoid placing your Pod on nodes that are already running a replica of your workload. Consider a web server that has three replicas. To ensure the three replicas are not placed in the same failure domain, you can use Pod anti-affinity as in the following snippet. In this case, the anti-affinity rule tells the scheduler that it should prefer placing Pods across zones, as determined by the `zone` label on cluster nodes:

```
# ... <snip> ...
      affinity:
        PodAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
```

```yaml
                    operator: In
                    values:
                    - my-web-server
                topologyKey: "zone"
# ... <snip> ...
```

In addition to Pod anti-affinity, Kubernetes provides Pod Topology Spread Constraints, which are an improvement to Pod anti-affinity rules when it comes to spreading Pods across failure domains. The problem with anti-affinity rules is that there is no way to guarantee Pods are spread *evenly* across the domains. You can either "prefer" scheduling them based on the topology key, or you can guarantee a single replica per failure domain.

The Pod Topology Spread Constraints provide a way for you to tell the scheduler to spread your workload. Similar to Pod anti-affinity rules, they are only evaluated against new Pods that need scheduling, and thus they are not retroactively enforced. The following snippet shows an example Pod Topology Spread Constraint that results in Pods spread across zones (based on the `zone` label of nodes). If the constraint cannot be satisfied, Pods will not be scheduled.

```yaml
# ... <snip> ...
spec:
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        foo: bar
# ... <snip> ...
```

When running multiple instances of an application, you should leverage these Pod placement features to improve the application's tolerance of infrastructure failure. Otherwise, you risk Kubernetes scheduling your workload in a way that does not achieve the failure tolerance you are looking for.

State Probes

Kubernetes uses many signals to determine the state and health of applications running on the platform. When it comes to health, Kubernetes treats workloads as opaque boxes. It knows whether the process is up or not. While this information is helpful, it is typically not enough to run and manage applications effectively. This is where probes come in. Probes provide Kubernetes with increased visibility of the application's condition.

Kubernetes provides three probe types: liveness, readiness, and startup probes. Before discussing each type in detail, let's review the different probing mechanisms that are common to all probe types:

Exec
: The kubelet executes a command inside the container. The probe is deemed successful if the command returns a zero exit code. Otherwise, the kubelet considers the container unhealthy.

HTTP
: The kubelet sends an HTTP request to an endpoint in the Pod. As long as the HTTP response code is greater than or equal to 200 and less than 400, the probe is deemed successful.

TCP
: The kubelet establishes a TCP connection with the container on a configurable port. The container is deemed healthy if the connection is established successfully.

In addition to sharing the probing mechanisms, all probes have a common set of parameters that you can use to tune the probe according to your workload. These parameters include success and failure thresholds, timeout periods, and others. The Kubernetes documentation describes each setting in detail, so we will not dive into them here.

Liveness Probes

Liveness probes help Kubernetes understand the health of Pods on the cluster. At the node level, the kubelet continuously probes Pods that have the liveness probe configured. When the liveness probe exceeds the failure threshold, the kubelet deems the Pod unhealthy and restarts it. Figure 14-2 shows a flowchart that depicts an HTTP liveness probe. The kubelet probes the container every 10 seconds. If the kubelet finds that the last 10 probes have failed, it restarts the container.

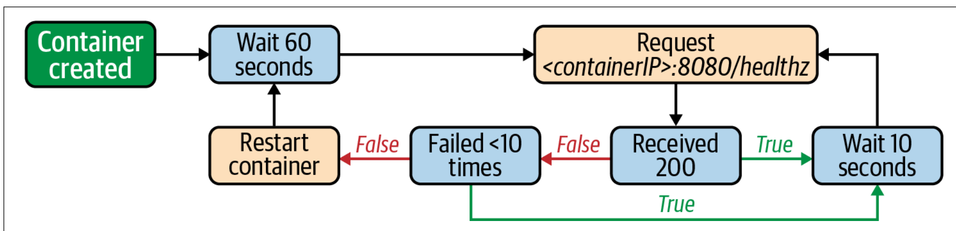
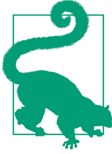


Figure 14-2. Flowchart that shows an HTTP-based liveness probe with a period of 10 seconds. If the probe fails 10 times consecutively, the Pod is deemed unhealthy and the kubelet restarts it.



Given that liveness probe failures result in container restarts, we typically suggest that liveness probe implementations should not check for the workload's external dependencies. By keeping the liveness probe local to your workload and not checking external dependencies, you prevent cascading failures that could otherwise occur. For example, a service that interacts with a database should not perform a "database availability" check as part of its liveness probe, as restarting the workload will most likely not fix the problem. If the app detects an issue with the database, the app can enter a read-only mode or gracefully disable the functionality that depends on the database. Another option is for the app to fail its readiness probe, which we discuss next.

Readiness Probes

The readiness probe is perhaps the most common and most important probe type in Kubernetes, especially for services that handle requests. Kubernetes uses readiness probes to control whether to route Service traffic to Pods. Thus, readiness probes provide a mechanism for the application to tell the platform that they're ready to accept requests.

As with liveness probes, the kubelet is responsible for probing the application and updating the Pod's status according to the probe results. When the probe fails, the platform removes the failing Pod from the list of available endpoints, effectively diverting traffic to other replicas that are ready. Figure 14-3 shows a flowchart that explains an HTTP-based readiness probe. The probe has a 5-second initial delay and a probing period of 10 seconds. On startup, the application begins receiving traffic only when the readiness probe succeeds. Then, the platform stops sending traffic to the Pod if the probe fails twice consecutively.

When deploying service-type workloads, make sure that you configure a readiness probe to avoid sending requests to replicas that cannot handle them. The readiness probe is not only critical when the Pod is starting up but also important during the lifetime of the Pod to prevent routing clients to replicas that have become unready.

Figure 14-3. Flowchart that shows an HTTP-based readiness probe with a period of 10 seconds. If the probe fails twice consecutively, the Pod is deemed not ready and it is taken out of the set of ready endpoints.

Startup Probes

Liveness and readiness probes have been available since the first version of Kubernetes. As the system gained popularity, the community identified a need to implement an additional probe, the startup probe. The startup probe provides extra time for slow-starting applications to initialize. Similar to liveness probes, failed startup probes result in the container being restarted. Unlike liveness probes, however, startup probes are executed only until they succeed, at which point the liveness and readiness probes take over.

If you are wondering why a liveness probe is not enough, let's consider an application that takes, on average, 300 seconds to initialize. You could indeed use a liveness probe that waits 300 seconds before stopping the container. During startup, this liveness probe would work. But what about later on when the application is running? If the application entered an unhealthy state, the platform would wait 300 seconds before restarting it! This is the problem that the startup probe solves. It looks after the workload during startup but then gets out of the way. Figure 14-4 shows a flowchart that walks through a startup probe like the one we just discussed. It has a failure threshold of 30 times and a probing period of 10 seconds.

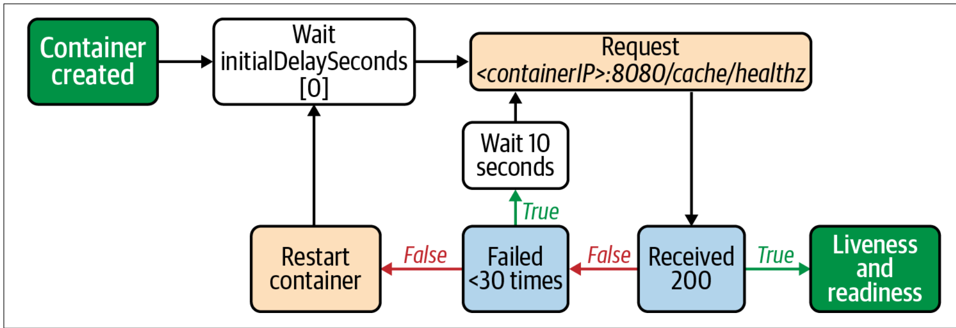


Figure 14-4. Flowchart that shows an HTTP-based startup probe with a period of 10 seconds. If the probe returns a successful response, the startup probe is disabled and the liveness/readiness probes are enabled. Otherwise, if the probe fails 30 times consecutively, the kubelet restarts the Pod.

While startup probes can be useful for certain applications, we usually recommend avoiding them unless absolutely necessary. We find liveness and readiness probes to be appropriate most of the time.

Implementing Probes

Now that we have covered the different probe types, let's dive into how you should approach them in your application, specifically liveness and readiness probes. We know that failed liveness probes result in the platform restarting the Pod, while failed readiness probes prevent traffic from being routed to the Pod. Given these different outcomes, we find that most applications that leverage both liveness and readiness probes should configure different probe endpoints or commands.

Ideally, the liveness probe fails only when there is a problem that requires a restart, such as a deadlock or some other condition that permanently prevents the app from making progress. Applications that expose an HTTP server commonly implement a liveness endpoint that unconditionally returns a 200 status code. As long as the HTTP server is healthy and the app can respond, there's no need to restart it.

In contrast to the liveness endpoint, the readiness endpoint can check for different conditions within the application. For example, if the application warms internal caches on startup, the readiness endpoint can return false unless the caches are warm. Another example is service overload, a condition under which the app can fail the readiness probe as a mechanism to shed load. As you can probably imagine, the checked conditions vary from one application to the next. In general, however, they are temporary conditions that resolve with the passing of time.

To summarize, we typically recommend using readiness probes for workloads that handle requests, given that readiness probes are meaningless in other application types, such as controllers, jobs, etc. When it comes to liveness probes, we recommend considering them only when restarting the application would help fix the problem. Lastly, we tend to avoid startup probes unless absolutely necessary.

Pod Resource Requests and Limits

One of Kubernetes' primary functions is to schedule applications across cluster nodes. The scheduling process involves, among other things, finding candidate nodes that have enough resources to host the workload. To place workloads effectively, the Kubernetes scheduler first needs to know the resource needs of your application. Typically, these resources encompass CPU and memory, but can also include other resource types such as ephemeral storage and even custom or extended resources.

In addition to scheduling your applications, Kubernetes also needs resource information to guarantee those resources at runtime. After all, the platform has limited resources that are shared across applications. Providing resource requirements is critical to your application's ability to *use* those resources.

In this section, we will discuss resource requests and resource limits, and how they can impact your application. We will not dig into the details of how the platform implements resource requests and limits, as we have already discussed it in Chapter 12.

Resource Requests

Resource requests specify the minimum amount of resources your application needs to run. In most cases, you should specify resource requests when deploying applications to Kubernetes. By doing so, you ensure your workload will have access to the requested resources at runtime. If you don't specify resource requests, you might find your application's performance diminishes significantly when resources on the node come under contention. You even risk the possibility of your application being terminated if the node needs to reclaim memory for other workloads. Figure 14-5 shows the termination of an application because another workload with memory requests starts consuming additional memory.

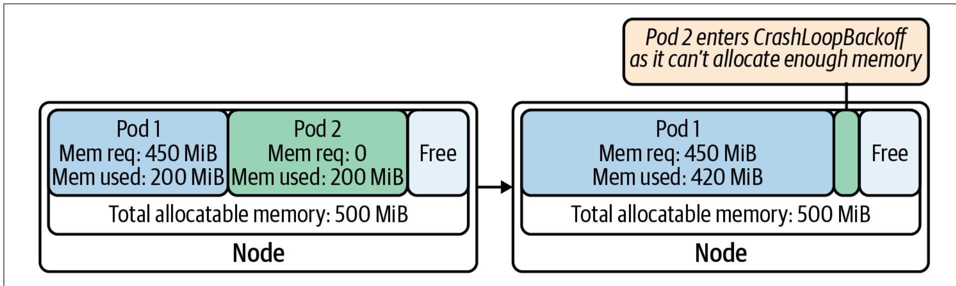


Figure 14-5. Pod 1 and Pod 2 share the node's memory. Each Pod is initially consuming 200 MiB out of the total 500 MiB. Pod 2 is terminated when Pod 1 needs to consume additional memory, as Pod 2 does not have memory requests in its specification. Pod 2 enters a crash loop as it cannot allocate enough memory to start up.

One of the main challenges with resource requests is finding the right numbers to use. If you are deploying an existing application, you might already have data you can analyze to determine the app's resource requests, such as the application's actual utilization over time or perhaps the size of the VMs hosting it. When you don't have historical data, you will have to use an educated guess and gather data over time. Another option is to use the Vertical Pod Autoscaler (VPA), which can suggest values for CPU and memory requests and even adjust those values over time. For more information about the VPA, see Chapter 13.

Resource Limits

Resource limits allow you to specify the maximum amount of resources your workload can consume. You might be wondering why you would impose an artificial limit. After all, the more resources available, the better. While this is true for some workloads, having unbound access to resources can result in unpredictable performance, as the Pod will have access to extra resources when available but will not when other Pods need the resources on the node. It gets even worse with memory. Given that memory is an incompressible resource, the platform has no other choice but to kill Pods when it needs to reclaim memory that was opportunistically consumed.

An important consideration to make when setting resource limits is whether you need to propagate those limits to the workload itself. Java applications are a good example. If the application uses an older version of Java (JDK version 8u131 or earlier), you need to propagate the memory limit down to the Java Virtual Machine (JVM). Otherwise, the JVM remains unaware of the limit and attempts to consume more memory than allowed. In the case of Java, you can configure the memory settings of the JVM using the `JAVA_OPTIONS` environment variable. Another option, although not always feasible, is to update the version of the JVM, as more recent versions gained the ability to detect the memory limits within containers. If you are

deploying an application that leverages a runtime, consider whether you need to propagate the resource limits for the application to understand them.

Limits are also important if you are trying to run performance tests or benchmarks against your workload. As you can imagine, it is likely that each test run will execute against Pods scheduled on different nodes at different times. If resource limits are not enforced on the workload, the test results can be highly variable as the workload under test can burst above its resource requests when the nodes have idle resources.

Usually, you should set your resource limits equal to your resource requests, which ensures your application will always have the same amount of resources, no matter what's happening with other Pods running beside it.

Application Logs

Application logs are critical to troubleshoot and debug applications both during development and in production. Applications running on Kubernetes should, as much as possible, log to the standard out and standard error streams (STDOUT/STDERR). This not only removes complexity in the application but also is the least complex solution from the platform's perspective when it comes to shipping logs to a central location. We covered this concern in Chapter 9, where we also discussed different log processing strategies, systems, and tools. In this section, we will touch on some of the considerations to make when thinking about application logs. The first thing we'll talk about is what you should log in the first place. Then, we will discuss unstructured versus structured logs. Finally, we will touch on improving the usefulness of logs by including contextual information in log messages.

What to Log

One of the first things to figure out when it comes to application logs is *what* to include in the logs. While development teams typically have their own philosophy, we have found that they tend to go overboard with logs. If you log too much, you run the risk of having too much noise and missing out on important information. On the flip side, if you log too little, it can become difficult to troubleshoot your application effectively. As with most things, there's a balance to strike here.

While working with application teams, we have found that a good rule of thumb that helps determine whether to log something is to ask the question, Is this log message actionable? If the answer is yes, this is a good indicator that it is worth logging that message. Otherwise, it is an indicator that the log message might not be useful.

Unstructured Versus Structured Logs

Application logs can be categorized as either unstructured or structured. Unstructured logs are, as the name suggests, strings of text that lack a specific format. They are arguably the most prevalent, as there is zero upfront planning that teams need to make. While the team might have generic guidelines, developers get to log messages in whatever format they like.

Structured logs, on the other hand, have predetermined fields that must be provided when logging events. They are typically formatted as JSON lines or key-value lines (e.g., `time="2015-08-09T03:41:12-03:21" msg="hello world!" thread="13" batchId="5"`). The main benefit of structured logs is that they are written in a machine-readable format, making them easier to query and analyze. With that said, structured logs tend to be harder for humans to read, so you must carefully consider this trade-off as you implement logging in your application.

Contextual Information in Logs

The primary purpose of logs is to provide insight into what happened within your application at a certain point in time. Perhaps you are troubleshooting a production issue in a live application, or maybe you are performing a root cause analysis to understand why something happened. To be able to complete such tasks, you typically need contextual information in log messages, in addition to what happened.

Let's consider a payment application as an example. When the application request serving pipeline encounters an error, in addition to logging the error itself, try to include the context surrounding the error as well. For example, if an error occurs because the payee was not found, include the payee name or ID, the user ID attempting to make the payment, the payment amount, etc. Such contextual information will improve your experience troubleshooting issues and will help you prevent such problems in the future. Having said that, avoid including sensitive information in your logs. You don't want to leak a user's password or credit card information.

Exposing Metrics

In addition to logs, metrics provide critical insight about how your application is behaving. Once you have application metrics, you can configure alerts to let you know when your application needs attention. Furthermore, by aggregating metrics over time, you can discover trends, improvements, and regressions as you roll out new versions of your software. This section discusses application instrumentation and some of the metrics you can capture, including RED (Rate, Errors, Duration), USE (Utilization, Saturation, Errors), and app-specific metrics. If you are interested in the platform components that enable monitoring and more additional discussions on metrics, check out Chapter 9.

Instrumenting Applications

In most cases, the platform can measure and surface metrics about your application's externally visible behavior. Metrics such as CPU usage, memory usage, disk IOPS, and others are readily available from the node that is running your application. While these metrics are useful, instrumenting your application to expose key metrics from within is worthwhile.

Prometheus is one of the most popular monitoring systems for Kubernetes-based platforms that we run into in the field. We have extensively covered Prometheus and its components in Chapter 9. In this section, we will focus our discussions on instrumenting apps for Prometheus.

Prometheus pulls metrics from your application using an HTTP request on a configurable endpoint (typically `/metrics`). This means that your application must expose this endpoint for Prometheus to scrape. More importantly, the endpoint's response must contain Prometheus-formatted metrics. Depending on the type of software you want to monitor, there are two approaches you can take to expose metrics:

Native instrumentation
: This option involves instrumenting your application using the Prometheus client libraries so that metrics are exposed from within the application process. This is an excellent approach when you have control over the source code of the application.

Out-of-process exporter
: This is an additional process running beside your workload that transforms preexisting metrics and exposes them in a Prometheus-compatible format. This approach is best suited for off-the-shelf software that you cannot instrument directly and is typically implemented using the sidecar container pattern. Examples include the NGINX Prometheus Exporter and the MySQL Server Exporter.

The Prometheus instrumentation libraries supports four metric types: Counters, Gauges, Histograms, and Summaries. Counters are metrics that can only increase, while Gauges are metrics that can go up or down. Histograms and Summaries are more advanced metrics than Counters and Gauges. Histograms place observations into configurable buckets that you can then use to compute quantiles (e.g., 95th percentile) on the Prometheus server. Summaries are similar to Histograms, except that they compute quantiles on the client side over a sliding time window. The Prometheus documentation explains the metric types in more depth.

There are three primary things you must do to instrument an application with the Prometheus libraries. Let's work through an example of instrumenting a Go service. First, you need to start an HTTP server to expose the metrics for Prometheus to scrape. The library provides an HTTP handler that takes care of encoding the metrics into the Prometheus format. Adding the handler would look something like this:

```go
func main() {
  // app code...

  http.Handle("/metrics",
    promhttp.HandlerFor(
      prometheus.DefaultGatherer,
      promhttp.HandlerOpts{},
  ))

  log.Fatal(http.ListenAndServe("localhost:8080", nil))
}
```

Next, you need to create and register metrics. For example, if you wanted to expose a Counter metric called `items_handled_total`, you would use code similar to the following:

```go
// create the counter
var totalItemsHandled = prometheus.NewCounter(
  prometheus.CounterOpts{
    Name: "items_handled_total",
    Help: "Total number of queue items handled.",
  },
)

// register the counter
prometheus.MustRegister(totalItemsHandled)
```

Finally, you need to update the metric according to what's happening in the application. Continuing the Counter example, you would use the `Inc()` method of the Counter to increment it:

```go
func handleItem(item Item) {

  // item handling code...

  // increment the counter as we handle items
  totalItemsHandled.Inc()
}
```

Instrumenting an application using the Prometheus libraries is relatively simple. The more complicated task is to determine the metrics that your application should expose. In the following sections, we will discuss different methods or philosophies you can use as a starting point to select metrics.

USE Method

The USE method, proposed by Brendan Gregg, focuses on system resources. When using this method, you capture Utilization, Saturation, and Errors (USE) for each of the resources your application uses. These resources typically include CPU, memory, disk, etc. They can also include resources that exist within the application software, such as queues, thread pools, etc.

RED Method

In contrast to the USE method, the RED method focuses more on the services themselves instead of the underlying resources. Initially proposed by Tom Wilkie, the RED method captures the Rate, Errors, and Durations of requests that the service handles. The RED method can be better suited for online services, as the metrics provide insight into your users' experience and how they perceive the service from their standpoint.

The Four Golden Signals

Another philosophy you can adopt is to measure the four golden signals, as proposed by Google in *Site Reliability Engineering* (O'Reilly). Google suggests you measure four critical signals for every service: Latency, Traffic, Errors, and Saturation. You might notice that these are somewhat similar to the metrics captured as part of the RED method, with the addition of Saturation.

App-Specific Metrics

The USE method, RED method, and four golden signals capture generic metrics that are applicable across most if not all applications. There is an additional class of metrics that surface app-specific information. For example, how long does it take to add an item to the shopping cart? Or how much time does it take to connect a customer with an agent? Typically, these metrics are correlated with business key performance indicators (KPIs).

Regardless of the method you choose, exporting metrics from your application is critical to its success. Once you have access to those metrics, you can build dashboards to visualize the behavior of your system, set up alerts to notify your on-call teams when something goes wrong, and perform trend analysis to derive business intelligence that can advance your organization.

Instrumenting Services for Distributed Tracing

Distributed tracing enables you to analyze applications that are composed of multiple services. They provide visibility into the execution flow of a request as it traverses the different services that make up the application. As discussed in Chapter 9, Kubernetes-based platforms can offer distributed tracing as a platform service using systems such as Jaeger or Zipkin. However, similar to monitoring and metrics, you must instrument services to take advantage of distributed tracing. In this section, we will explore how to instrument services using Jaeger and OpenTracing. First, we will discuss how to initialize a tracer. Then, we will dive into how to create spans within a service. A *span* is a named, timed operation that is the building block of a distributed trace. Finally, we will explore how to propagate tracing context from one service to another. We will use Go and the Go libraries for examples, but the concepts are applicable to other programming languages.

Initializing the Tracer

Before being able to create spans within the service, you must initialize the tracer. Part of the initialization involves configuring the tracer according to the environment the application is running. The tracer needs to know the service name, the URL to send trace information, etc. For these settings, we recommend using the Jaeger client library environment variables. For example, you can set the service name using the `JAEGER_SERVICE_NAME` environment variable.

In addition to configuring the tracer, you can integrate the tracer with your metrics and logging libraries as you initialize the tracer. The tracer uses the metrics library to emit metrics about what's happening with the tracer, such as the number of traces and spans sampled, the number of successfully reported spans, and others. On the other hand, the tracer leverages the logging libraries to emit logs when it encounters errors. You can also configure the tracer to log spans, which is rather useful in development.

To initialize a Jaeger tracer in a Go service, you would add code to your application similar to the following. In this case, we are using Prometheus as the metrics library and Go's standard logging library:

```
package main

import (
	"log"

	jaeger "github.com/uber/jaeger-client-go"
	"github.com/uber/jaeger-client-go/config"
	"github.com/uber/jaeger-lib/metrics/prometheus"
)

func main() {
  // app initialization code...
```

```go
	metricsFactory := prometheus.New() ❶

	cfg := config.Configuration{} ❷
	tracer, closer, err := cfg.NewTracer( ❸
		config.Metrics(metricsFactory),
		config.Logger(jaeger.StdLogger),
	)
	if err != nil {
		log.Fatalf("error initializing tracer: %v", err)
	}

	defer closer.Close()

	// continue main()...
}
```

❶ Create a Prometheus metrics factory that Jaeger can use to emit metrics.

❷ Create a default Jaeger configuration with no hardcoded configuration (use environment variables instead).

❸ Create a new tracer from the configuration, and provide the metrics factory and the Go standard library logger.

With the tracer initialized, we can start creating spans in our service.

Creating Spans

Now that we have a tracer, we can start creating spans within our service. Assuming the service is somewhere in the middle of the request processing flow, the service needs to deserialize the incoming span information from the previous service and create a child span. Our example is an HTTP service, so the span context is propagated via HTTP headers. The following code extracts the context from the headers and creates a new span. Note that the tracer we initialized in the previous section must be in scope:

```go
package main

import (
	"github.com/opentracing/opentracing-go"
	"github.com/opentracing/opentracing-go/ext"
	"net/http"
)

func (s server) handleListPayments(w http.ResponseWriter, req *http.Request) {
	spanCtx, err := s.tracer.Extract( ❶
		opentracing.HTTPHeaders,
		opentracing.HTTPHeadersCarrier(req.Header),
	)
```

```
        if err != nil {
                // handle the error
        }

        span := opentracing.StartSpan( ❷
                "listPayments",
                ext.RPCServerOption(spanCtx),
        )
        defer span.Finish()
}
```

❶ Extract the context information from the HTTP headers.

❷ Create a new span using the extracted span context.

As the service handles the request, it can add child spans to the span we just created. As an example, let's assume the service calls a function to perform a SQL query. We can use the following code to create a child span for the function and set the operation name to `listPayments`:

```
func listPayments(ctx context.Context) ([]Payment, error) {
  span, ctx := opentracing.StartSpanFromContext(ctx, "listPayments")
  defer span.Finish()

  // run sql query
}
```

Propagate Context

Up to this point, we've created spans within the same service or process. When there are other services involved in processing a request, we need to propagate the trace context over the wire for the service on the other end. As discussed in the previous section, you can use HTTP headers to propagate the context.

The OpenTracing libraries provide helper functions you can use to inject the context into HTTP headers. The following code shows an example that uses the Go standard library HTTP client to create and send the request:

```
import (
    "github.com/opentracing/opentracing-go"
        "github.com/opentracing/opentracing-go/ext"

          "net/http"
)

// create an HTTP request
req, err := http.NewRequest("GET", serviceURL, nil)
if err != nil {
  // handle error
}
```

```
// inject context into the request's HTTP headers
ext.SpanKindRPCClient.Set(span) ❶
ext.HTTPUrl.Set(span, url)
ext.HTTPMethod.Set(span, "GET")
span.Tracer().Inject( ❷
  span.Context(),
  opentracing.HTTPHeaders,
  opentracing.HTTPHeadersCarrier(req.Header),
)

// send the request
resp, err := http.DefaultClient.Do(req)
```

❶ Adds a tag to mark the span as the client side of a service call.

❷ Injects the span context into the request's HTTP headers.

As we've discussed through these sections, instrumenting an application for tracing involves initializing a tracer, creating spans within the service, and propagating the span context to other services. There is additional functionality that you should explore, including tagging, logging, and baggage. If the platform team offers tracing as a platform service, now you have an idea of what it takes to take advantage of it.

Summary

There are multiple things you can do to make your applications run better in Kubernetes. While most require investing time and effort to implement, we find that they are critical to achieve production-grade outcomes for your applications. As you onboard applications to your platform, make sure to consider the guidance provided in this chapter, including injecting configuration and secrets at runtime, specifying resource requests and limits, exposing application health information using probes, and instrumenting applications with logs, metrics, and traces.

CHAPTER 15
Software Supply Chain

Implementing a Kubernetes platform should never be the goal of your team or company (assuming you are not a vendor or consultant!). This might seem a strange claim for a book exclusively devoted to Kubernetes to make, but let's step back a moment. All companies are in the business of delivering their *core-competency*. This might be an ecommerce platform, a SaaS monitoring system, or an insurance website. Platforms like Kubernetes (and almost any other tooling) exist to *enable* the delivery of core business value, a truth that is often forgotten by teams when designing and implementing IT solutions.

With that sentiment in mind, this chapter will focus on the actual process of getting code from developers to production on Kubernetes. To best cover each stage that we think is relevant, we'll follow the model of a pipeline that many are familiar with.

First we'll look at some of the considerations when building container images (our deployed assets) from source code. If you're already utilizing Kubernetes or other container platforms you'll probably be familiar with some of the concepts in this section, but hopefully we'll cover some questions that you may not have considered. If you're *new* to containers this will be a paradigm shift from the way that you currently build software (WAR files, Go binaries, etc.) to thinking about the container image and the nuances involved with building and maintaining them.

Once we have built our assets we need somewhere to store them. We'll discuss container registries (e.g., DockerHub, Harbor, Quay) and the functionality that we think is important when choosing one. Many of the attributes of container registries are related to security, and we'll discuss options like image scanning, updates, and signing.

Finally, we'll dedicate some time to reviewing continuous delivery and how those practices and associated tooling intersect with Kubernetes. We'll look at emerging ideas like GitOps (deployments through syncing cluster state from git repositories) and more traditional imperative pipeline approaches.

Even if you are not yet running Kubernetes, you will likely have considered and/or solved for all of the high-level areas just mentioned (build, asset storage, deployment). It's reasonable that everyone has investments and expertise in existing tooling and approaches, and we very rarely encounter a situation where an organization wants to start afresh with its entire software supply chain. One of the things we'll try to emphasize in this chapter is that there are clean handoff points in the pipeline and we can pick and choose the most effective approaches for each phase. As with many of the topics covered in this book, it is entirely possible (and recommended) to enact *incremental* positive change while remaining focused on delivering business value.

Building Container Images

Before containers we would package applications as a binary, compressed asset, or raw source code for it to be deployed onto a server. This would either run standalone or inside of an application server. Alongside the application itself we'd need to ensure the environment contained the correct dependencies and configuration available for it to run successfully in the environment we were deploying to.

In a container-based environment, the container image is the deployable asset. It contains not only the application binary itself but also the execution environment and any associated dependencies. The image itself is a compressed set of *filesystem layers* alongside some metadata, which together conform to the Open Container Initiative (OCI) Image Specification. This is an agreed standard within the cloud native community to ensure that image building can be implemented in many different ways (we'll see some of these in the following sections) while still producing an artifact that is runnable by all the different container runtimes (more information on this can be found in Chapter 3).

Typically, building a container image involves creating a Dockerfile that describes the image and using Docker Engine to execute the Dockerfile. With that said, there is an ecosystem of tools (each with their own approaches) that you can use to create container images in different scenarios. To borrow a concept from BuildKit (one such tool, built by Docker) we can think about building in terms of *frontend* and *backend*. The *frontend* is the method for defining the high-level process that should be used to build the image, e.g., a Dockerfile or Buildpack (more on these later in this chapter). The *backend* is the actual build engine that takes the definition generated by the *frontend* and executes commands on the filesystem to construct the image.

In many cases the *backend* is the Docker daemon, which may not be suitable for all cases. For example, if we want to run builds in Kubernetes we need either to run a Docker daemon inside a container (Docker in Docker) or mount the Docker Unix socket from the host machine into the build container. Both of these approaches have drawbacks, and in the latter case exposes potential security issues. In response to these issues, other build backends like Kaniko have emerged. Kaniko uses the same *frontend* (a Dockerfile) but utilizes different techniques to create the image under the hood, making it a solid choice for running in a Kubernetes Pod. When deciding how you want to build images, you should answer the following questions:

- Can we run our builder as root?
- Are we OK mounting the Docker socket?
- Do we care about running a daemon?
- Do we want to containerize builds?
- Do we want to run them among workloads in Kubernetes?
- How much do we intend to leverage layer caching?
- How will our tooling choice affect distributing builds?
- What *frontends* or image definition mechanisms do we want to use? What is supported?

In this section, we will first cover some of the patterns and antipatterns we've seen when building container images (Cloud Native Buildpacks) that will hopefully help you on your journey to build better container images. Then, we will review an alternative method for building container images, and how all of these techniques can be integrated into a pipeline.

One question that often comes up early on within organizations is, Who should be responsible for building images? Early on as Docker was becoming popular it was largely embraced as a developer-focused tool. In our experience, smaller organizations still have developers responsible for writing Dockerfiles and defining the build process for their application images. However, as organizations look to adopt containers (and Kubernetes) at scale, having individual developers or development teams all creating their own Dockerfiles becomes unsustainable. Firstly it creates extra work for developers, which pulls them away from their core responsibility, and secondly, it results in a huge variance in produced images with little to no standardization.

As a result, we are seeing a move toward abstracting the build process from development teams and instead moving the responsibility toward operations and platform teams to implement *source to image* patterns and tooling that receive a code repository as an input and are capable of producing a container image ready to move through the pipeline. We'll discuss this pattern more in "Cloud Native Buildpacks" on page 432. In the interim we have also commonly seen a pattern of platform teams

running workshops and/or assisting development teams with Dockerfile and image creation. As organizations scale this can be an effective first step but is usually not sustainable given the ratio of development teams to platform personnel.

The Golden Base Images Antipattern

In the field we have encountered several antipatterns that are usually the result of teams not adjusting their thinking to embrace patterns that have emerged in the container and cloud native landscape. Maybe the most common of these is the concept of pre-ordained, *gold* images. The scenario is that in a pre-container environment specific base images (for example, a preconfigured CentOS base) would be approved for use within an organization, and all applications going into production would have to be based on that image. This approach is usually adopted for *security* reasons, as the tools and libraries in the image have been well vetted. However, when moving to containers, teams found themselves consigned to reinventing the wheel by pulling useful upstream images from third parties and vendors and rebasing their applications and configurations onto them.

This introduces a few related issues. Firstly, there is the additional work involved with the initial conversion from the upstream image to an internal customized version. Secondly, there is now an onus of maintenance placed on the internal platform team to store and maintain these internal images. Because this situation can sprawl (given how many images are in use in a typical environment), this approach usually ends up resulting in a *worse* security posture as updates are performed infrequently (if at all) given the extra work involved.

Our recommendation in this area is usually to partner with security teams and identify what specific requirements the gold images are serving. Usually several of the following will apply:

- Ensure specific agents/software is installed
- Ensure no vulnerable libraries are present
- Ensure user accounts have the correct permissions

By understanding the reasoning behind the restrictions, we can instead codify these requirements into tooling that will sit in the pipeline and reject and/or alert on noncompliant images and maintain the desired security posture, while still broadly allowing teams to reuse images (and the work that has gone into crafting them) from the upstream community. We'll take a deeper look at one example workflow in "Image Registries" on page 434.

One of the more compelling reasons to specify a base OS is to ensure that operational knowledge exists in the organization should troubleshooting be required. However, when digging a little deeper this is not as useful as it may seem. Very rarely should it

be necessary to `exec` into containers to troubleshoot specific issues, and even then the differences between Linux-based operating systems are fairly trivial for the kinds of support required. Additionally, more and more applications are being packaged in ultra-lightweight scratch or distroless images to reduce the overhead inside the container.

Attempting to refactor all *upstream*/vendor images onto your own base(s) should be avoided for the reasons described in this section. However, we're not asserting that maintaining an internal set of curated base images is a bad idea. These can be great to use as a foundation for your own applications, and we talk about some of the considerations when building these internal bases in the next section.

Choosing a Base Image

The base image of the container determines the bottom layers on which the application's container image is to be built. The base image is critical as it usually contains operating system libraries and tools that will be part of your application container image. If you are not mindful when choosing a base image, it can be the source of unnecessary libraries and tools that not only bloat your container image but also can become security vulnerabilities.

Depending on your organization's maturity and security posture, you might not have a choice when it comes to base images. We have worked with many organizations that have a dedicated team responsible for curating and maintaining a set of approved base images that must be used across the organization. With that said, if you do have a choice or you are part of the team that is vetting base images, consider the following guidelines when evaluating base images:

- Ensure the images are published by a reputable vendor. You don't want to use a base image from a random DockerHub user. After all, these images will be the foundation for most, if not all, your applications.
- Understand the update cycle and prefer images that are updated continuously. As mentioned earlier, the base image typically contains libraries and tools that must be patched whenever new vulnerabilities are discovered.
- Prefer images that have an open source build process or specification. This is typically a Dockerfile that you can inspect to understand how the image is built.
- Avoid images that have unnecessary tools or libraries. Prefer minimal images that provide a small footprint that your developers can build upon, when necessary.

Most of the time if you are building your own images we've seen scratch or distroless to be a solid choice as both embody the preceding principles. The scratch image contains absolutely nothing, so with a simpler static binary scratch can be the leanest possible image. However, you may encounter issues if you need root CA certificates

or some other assets. These can be copied in, but are something to think about. The distroless base is what we'd recommend in most cases as they contain some sensible users precreated (`nonroot`, `nobody`, etc.) and a set of minimal required libraries that vary depending on the flavor of the base image chosen. Distroless has several language-specific base variants for you to choose from.

In the next few sections we'll continue to talk about best-practice patterns, starting with the importance of specifying an appropriate user for your application to run under.

Runtime User

Because of the container isolation model (mainly the fact that containers share the underlying Linux kernel), the runtime user of a container has important implications that some developers don't think about. In most cases, when the container's runtime user is left unspecified, the process runs as the root user. This is problematic because it increases the attack surface of the container. For example, if an attacker were to compromise the application and escape the container, they could gain root access on the underlying host.

When building your container image, it is critical for you to consider the runtime user of the container. Does the application need to run as root? Does the application depend on the contents of */etc/passwd*? Do you need to add a nonroot user to the container image? As you answer these questions, ensure that you specify the runtime user in the container image's configuration. If you are using a Dockerfile to build your image, you can use the `USER` directive to specify the runtime user, as in the following example, which runs the `my-app` binary with the user and group ID `nonroot` (which is configured by default as part of the distroless set of images):

```
FROM gcr.io/distroless/base
USER nonroot:nonroot
COPY ./my-app /my-app
CMD ["./my-app", "serve"]
```

Even though you can specify the runtime user in your Kubernetes deployment manifests, defining it as part of the container image specification is valuable as it results in a self-documenting container image. It also ensures that developers use the same user and group ID as they work with the container in their local or development environments.

Pinning Package Versions

If your application leverages external packages you will most likely install them using a package manager such as `apt`, `yum`, or `apk`. As you build your container image, it is important that you pin or specify the version of these packages. For example, the following example shows an application that depends on imagemagick. The `apk`

instruction in the Dockerfile pins imagemagick to the version that is compatible with the application:

```
FROM alpine:3.12
<...snip...>
RUN ["apk", "add", "imagemagick=7.0.10.25-r0"]
<...snip...>
```

If you leave package versions unspecified, you risk getting different packages that might break your application. Thus, always specify the versions of the packages you install in your container image. By doing so, you ensure that your container image builds are repeatable and produce container images with compatible package versions.

Build Versus Runtime Image

In addition to packaging applications for deployment, development teams can also leverage containers to build their applications. Containers can provide a well-defined build environment that can be codified into a Dockerfile, for example. This is useful as developers are not required to install the build tooling in their systems. More importantly, containers can provide a standardized build environment across the entire development team and their continuous integration (CI) systems.

While using containers to build applications can be useful, it is important to distinguish between the build container image and the runtime image. The build image contains all the tooling and libraries that are necessary to compile the application, whereas the runtime image contains the application to be deployed. For example in a Java application we might have a build image that contains the JDK, Gradle/Maven, and all of our compilation and testing tooling. Then our runtime image can contain only the Java runtime and our application.

Given that the application typically does not need the build tooling at runtime, the runtime image should not contain these tools. This results in a leaner container image that is faster to distribute and has a tighter attack surface. If you are using docker to build images, you can leverage its multistage build feature to separate the build from the runtime image. The following snippet shows a Dockerfile for a Go application. The build stage uses the `golang` image, which includes the Go toolchain, while the runtime stage uses the scratch base image and contains nothing more than the application binary:

```
# Build stage
FROM golang:1.12.7 as build ❶

WORKDIR /my-app

COPY go.mod . ❷
RUN go mod download
```

```
COPY main.go .
ENV CGO_ENABLED=0
RUN go build -o my-app

# Deploy stage
FROM gcr.io/distroless/base ❸
USER nonroot:nonroot ❹
COPY --from=build --chown=nonroot:nonroot /my-app/my-app /my-app ❺
CMD ["/my-app"]
```

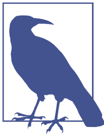
❶ The main `golang` image contains all the Go build tools, which are not required at runtime.

❷ We copy the *go.mod* file first and download so that we can cache this step if the code changes but the dependencies don't.

❸ We can use `distroless` as a runtime image to take advantage of a minimal base but with no unnecessary extra dependencies.

❹ We want to run our apps as a nonroot user if possible.

❺ Only the compiled file (*my-app*) is being copied from the build stage to the deploy stage.



Containers run a single process and usually have no supervisor or init system. For this reason you need to ensure that signals are handled correctly and orphaned processes are correctly reparented and reaped. There are several minimal init scripts capable of fulfilling these requirements and acting as a bootstrap for your application instance.

Cloud Native Buildpacks

An additional method of building container images involves tooling that analyze the application's source code and automatically produce a container image. Similar to application-specific build tools, this approach greatly simplifies the developer experience as developers don't have to create and maintain Dockerfiles. Cloud Native Buildpacks is an implementation of such an approach, and the high-level flow is shown in Figure 15-1.

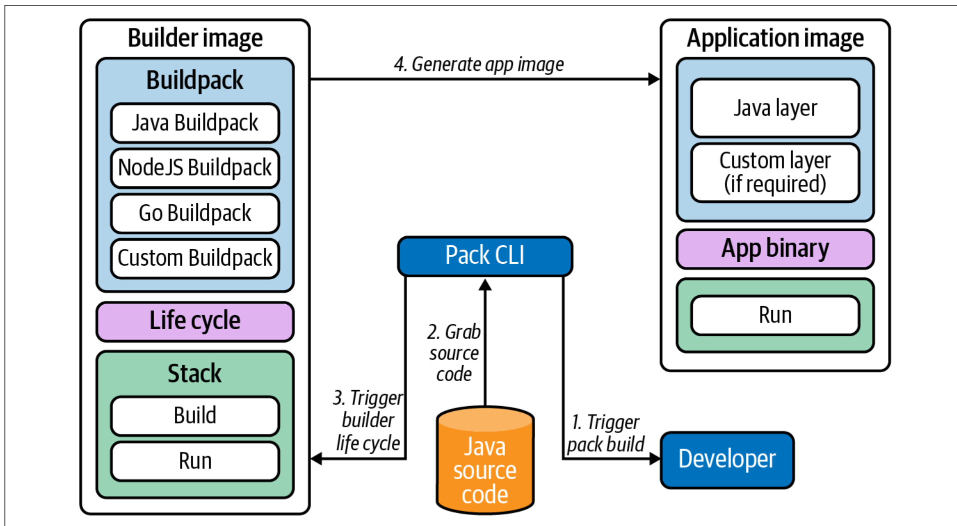


Figure 15-1. Buildpack flow.

Cloud Native Buildpacks (CNB) is a container-focused implementation of Buildpacks, a technology that Heroku and Cloud Foundry have used for years to package applications for those platforms. In the case of CNB, it packages applications into OCI container images, ready to run on Kubernetes. To build the image, CNB analyzes the application source code and executes the buildpacks accordingly. For example, the Go buildpack is executed if there are Go files present in your source code. Similarly, the Maven (Java) buildpack is executed if CNB finds a *pom.xml* file. This all happens behind the scenes, and developers can initiate this process using a CLI tool called `pack`. The great thing about this approach is that the buildpacks are tightly scoped, which enables building high-quality images that follow best practices.

In addition to improving the developer experience and lowering the barrier to platform adoption, platform teams can leverage custom buildpacks to enforce policy, ensure compliance, and standardize the container images running in their platform.

Overall, providing a solution that builds container images from source code can be a worthy endeavor. Furthermore, we find that the value of such a solution increases with the size of the organization. At the end of the day, development teams want to focus on building value in the application and not on how to containerize it.

Image Registries

If you are already using containers, then you'll likely have a registry you prefer. It's one of the core requirements for utilizing Docker and Kubernetes because we need somewhere to store the images we build on one machine and want to run on many others (either standalone or in a cluster). As is the case for images, the OCI also defines a standard spec for registry operations (to ensure interoperability), and there are many proprietary and open source solutions available, most of which share a common set of core features. Most image registries are comprised of three major components: a server (for the user interface and API logic), a blob store (for the images themselves), and a database (for user and image metadata). Usually, the storage backends are configurable, and this can impact how you design your registry architecture. We'll talk more about this in a minute.

In this section we'll look at some of the most important features offered by registries and some of the patterns for integrating them into your pipeline. We're not going to look deeply at any *specific* registry implementations, as functionality is generally similar; however, there are scenarios where you may want to lean in a certain direction based on your existing setup or requirements.

If you are already leveraging an artifact store like Artifactory or Nexus you may want to take advantage of their image hosting capabilities for ease of management. Similarly, if your environments are heavily cloud-based there may be cost benefits to utilizing cloud provider registries like AWS Elastic Container Registry (ECR), Google Container Registry (GCR), or Azure Container Registry (ACR).

Another key factor to consider when choosing a registry is the topology, architecture, and failure domains of your environments and clusters. You may choose to place registries in each failure domain to ensure high availability. When doing this you'll need to decide whether you want a centralized blob store or whether you want blob stores in each region and set up image replication between the registries. Replication is a feature of most registries that allows you to push an image to one of a set of registries and have that image automatically pushed to the others in the set. Even if this is not directly supported in your registry of choice, it is fairly trivial to set up basic replication by using a pipeline tool (e.g., Jenkins) and webhooks that trigger on each image push.

The decision of one versus many registries is also impacted by how much throughput you need to support. In organizations with many thousands of developers triggering code and image builds on every code commit, the number of concurrent operations (pulls and pushes) can be significant. It is important to therefore understand that an image registry, while playing only a limited role in the pipeline, is in the critical path for not only production deployments but also development activities. It is a core

component that must be monitored and maintained in the same way as other critical components to achieve a high level of service availability.

Many registries are built with the intention of being easily run *inside* a cluster or containerized environment. This approach (which we'll cover again in "Continuous Delivery" on page 439) has many advantages. Primarily, we are able to leverage all of the primitives and conventions inside Kubernetes to keep the services running, discoverable, and easily configurable. The obvious downside here is that we now have a reliance on a service *inside* the cluster to provide images to start new services inside that cluster. It's more common to see registries run on a shared services cluster and have a failover system to a backup cluster to ensure that *some* instance of the registry will always be able to service requests.

We've also commonly seen registries run *outside* of Kubernetes and treated as more of a standalone *bootstrap* component that is required by all clusters. This is usually the case where an organization is already using an existing instance of Artifactory or another registry and repurposes it for image hosting. Utilizing cloud registries is also a common pattern here, although you also need to be aware of their availability guarantees (as the same topology questions apply) and potential extra latency.

In the following subsections we'll look at some of the most common concerns when choosing and working with a registry. These concerns are all security related as securing our software supply chain revolves around our deployed artifacts (images). First we'll look at vulnerability scanning and how to ensure that our images don't contain known security flaws. Then we'll describe a commonly used quarantine flow that can be effective at bringing external/vendor images into our environments. Finally, we'll discuss image trust and signing. This is an area that many organizations are interested in but where the upstream tooling and approaches are still maturing.

Vulnerability Scanning

Scanning images for known vulnerabilities is a key competency of most image registries. Usually the scanning itself along with the database of common vulnerabilities and exposures (CVEs) are delegated to a third-party component. Clair is a popular open source choice and, in many cases, is pluggable should you have specific requirements.

Every organization will have its own requirements about what constitutes an acceptable risk when considering CVE scores. Registries will commonly expose controls that allow you to disable the pulling of images that contain CVEs over a defined score threshold. Additionally, the ability to add CVEs to an allowlist can be useful to bypass issues that are flagged but not relevant in your environment, or for those CVEs that are deemed acceptable risk and/or have no fixes released and available.

This static scanning at initial pull time can be useful to begin with, but what happens if vulnerabilities are discovered over time in the images we're already using in the environment? Scans can be scheduled to detect these changes, but then we need to have a plan for updating and replacing the images. It can be tempting to automatically remediate (patch) and push out updated images, and there are solutions that will always try to keep images up-to-date. However, this can be problematic as image updates can introduce changes that may be incompatible and/or break the running application. These automated image update systems may also work outside of your designated deployment change process and could be hard to audit in the environment. Even the blocking of image pulls (described previously) can cause issues. If a core application's image has a new CVE discovered and pulls are suddenly prohibited, this could cause availability issues in the application if those workloads are scheduled to new nodes and images are unavailable for pulling. As we've discussed many times in this book, it's imperative to understand the trade-offs that you encounter when implementing each solution (in this case, security versus availability) and make informed, well-documented decisions.

A more common model than the automatic remediation described briefly is to alert and/or monitor on image vulnerability scans and bubble these up to operations and security teams. The implementation of this alerting may differ depending on the capabilities offered by your choice of registry. Some registries can be configured to trigger a webhook call on completion of a scan with the payload including details of the vulnerable images and discovered CVEs. Others may expose a scrapeable set of metrics with image and CVE details that can be alerted on using standard tooling (take a look at Chapter 9 for more details on metrics and alerting tools). While this method requires more manual intervention, it allows good visibility into the security state of images in your environment while also affording more control over how and when they are patched.

Once we have CVE information for an image, then decisions about whether or not to patch the image (and when) can be made based on the impact of the vulnerability. If we need to patch and update the image we can trigger the update, testing, and deployment via our regular deployment pipelines. This ensures that we have full transparency and auditability and that those changes all go through our regular processes. We will discuss CI/CD and deployment models in more detail later in this chapter.

While the static image vulnerability scanning covered in this subsection is a commonly implemented part of an organization's software supply chain, it is only one layer of what should be a *defense in depth* strategy to container security. Images may download malicious content post-deployment or containerized applications may be compromised/hijacked at runtime. It's essential therefore to implement some type of runtime scanning. In a more naive form, this could take the form of periodic filesystem scanning on running containers to ensure that no vulnerable binaries and/or

libraries are introduced post-deployment. However, for more robust protection it's necessary to limit the *actions* and *behaviors* that a container is capable of performing. This eliminates the inevitable game of whack-a-mole that can occur with CVEs being discovered and patched and instead focuses on the capabilities a containerized application should possess. Runtime scanning is a larger topic that we don't have the space to fully cover here, but you should look into tools like Falco and the Aqua Security suite.

Quarantine Workflow

As mentioned, most registries provide a mechanism to scan images for known vulnerabilities and restrict image pulls. However, there may be additional requirements that images must satisfy before they can be used. We have also encountered the scenario where developers are unable to directly pull images from the public internet and must use an internal registry. Both of these use cases can be solved by using a multiregistry setup with a quarantine workflow pipeline described next.

First, we can provide developers with a self-service portal to request images. Something like ServiceNow or a Jenkins job works fine here, and we've seen this many times. Chatbots can also offer a more seamless integration for developers and are gaining popularity. Once an image is requested, it is automatically pulled to a *quarantine* registry where checks can be run on the image and the pipeline can spin up environments to pull and verify the image meets specific criteria.

Once the checks pass, the image can be signed (this is optional, see "Image Signing" on page 438 for more information) and pushed to an approved registry. The developer can also be notified (either via the chatbot, or an updated ticket/job, etc.) that the image has been approved (or rejected, and the reasoning). The whole flow can be seen in Figure 15-2.

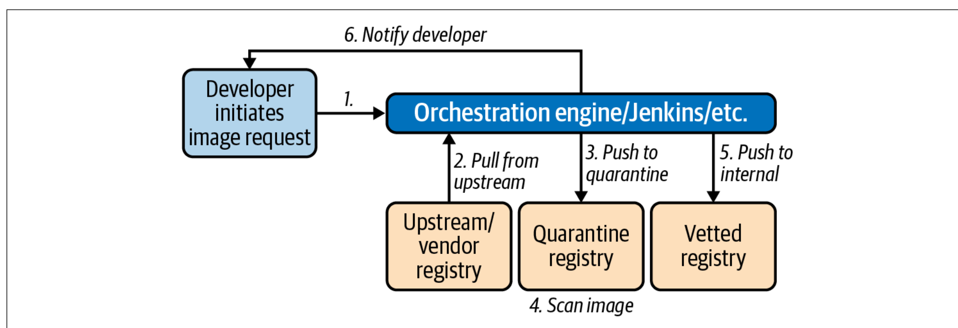


Figure 15-2. Quarantine flow.

This flow can be combined with admission controllers to ensure that only images that are signed, or those that come from a specific registry, are allowed to be run on a cluster.

Image Signing

The issue of supply chain security is becoming more prevalent as applications come to rely on an increasing number of external dependencies, be those code libraries or container images.

One of the security features often mentioned when discussing images is the notion of signing. Very simply, the concept with signing is that an image publisher can cryptographically sign an image by generating a hash of the image and associating their identity with it before pushing it to a registry. Users are then able to verify the authenticity of an image by validating the signed hash against the publisher's public key.

This workflow is attractive because it means we can create an image at the start of our software supply chain and sign it after each stage of the pipeline. Perhaps we can sign it after testing has been completed, and again after it has been approved for deployment by a release management team. Then at deploy time we are able to gate the deployment of the image into production based on whether it has been signed by the various parties that we specify. Not only do we ensure that it has passed those approvals, but we ensure that it is exactly the same image that is now being promoted to a production environment. This high-level flow is shown in Figure 15-3.

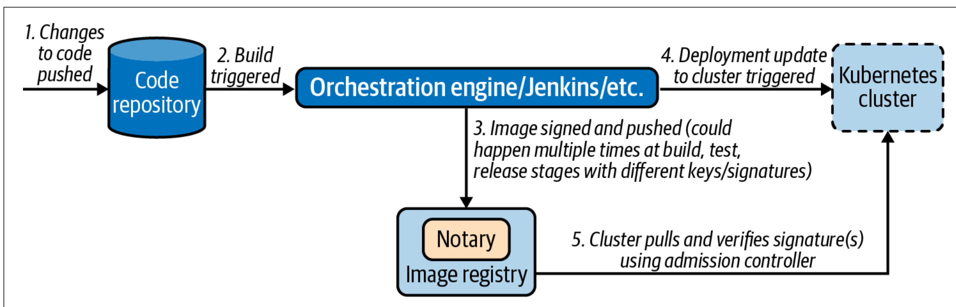


Figure 15-3. Signing flow.

The primary project in this area is Notary, which was originally developed by Docker and is built upon The Update Framework (TUF), a system designed to facilitate the secure distribution of software updates.

Despite its benefits, we have not encountered much adoption of image signing in the field for a couple of reasons. First, Notary has several components including a server and multiple databases. These are additional components that need to be installed, configured, and maintained. Not only that, but because the ability to sign and verify

images is usually in the critical path for software deployment, the Notary system must be configured for high availability and resilience.

Secondly, Notary requires each image be identified with a Globally Unique Name (GUN), which includes the registry URL as part of the name. This makes signing more problematic if you have multiple registries (e.g., caches, edge locations) as the signatures are tied to the registry and cannot be moved/copied.

Finally, Notary and TUF require different sets of key pairs to be used across the signing process. Each of the keys have different security requirements and can be challenging to rotate in the case of a security breach. While it provides an academically well-designed solution, the current Notary/TUF implementation posed too high of a barrier to entry for many organizations that were only just getting comfortable with some of the base technologies they were using. Thus, many weren't ready to trade more convenience and knowledge for the additional security benefits that the signing workflow provided.

At the time of writing, there are efforts underway to develop and release a second version of Notary. This updated version should improve the user experience by solving many of the issues just discussed, like reducing the complexity of key management and eliminating the constraint that signatures are not transferable by bundling them with the OCI images themselves.

There are already several existing projects that implement an admission webhook that will check images to ensure they have been signed before allowing them to be run in a Kubernetes cluster. Once the issues are addressed, we anticipate signing becoming a more oft-implemented property of the software supply chain and these signing admission webhooks to mature even further.

Continuous Delivery

In the previous sections we've discussed in detail the process of converting source code into a container image. We also looked at where images are stored and the architectural and procedural decisions we need to make around choosing and deploying image registries. In this final section we'll turn to examining the entire pipeline that ties these early steps up with the actual deployment of the image to potentially multiple Kubernetes clusters across many environments (test, staging, production).

We'll cover how to integrate the build process into the automated pipeline before looking at imperative, push-driven pipelines that many folks will already be familiar with. Lastly, we'll take a look at some of the principles and tooling emerging in the field of GitOps, a relatively new approach to deployments that leverages version control repositories as the source of truth for the assets that should be deployed to our environments.

It's worth noting that continuous delivery is a huge area and is the sole subject of many books. In this section we're assuming some knowledge of CD principles, and we'll focus on how to implement those principles within Kubernetes and associated tooling.

Integrating Builds into a Pipeline

For local development and testing phases, developers may build images with Docker locally. However, anything beyond those early phases and organizations will want to have builds performed as part of an automated pipeline triggered by the committing of code into a central version control repository. We'll talk later in this chapter about more advanced patterns around the actual *deployment* of images into an environment, but in this section want to focus purely on how the build phases can also be run in cluster using cloud native pipeline automated tooling.

We typically want new image builds to be triggered with a code commit. Some pipeline tools will intermittently poll a set of configured repositories and trigger a task run when changes are detected. In other cases it might be possible to trigger a process to begin by firing a webhook from the version control system. We'll use a few examples from Tekton, a popular open source pipeline tool that is designed to run on Kubernetes to illustrate some concepts in this section. Tekton (and many other Kubernetes native tools) utilize CRDs to describe components in the pipeline. In the following code, we can see a (heavily edited) instance of a `Task` CRD that can be reused across multiple pipelines. Tekton maintains a catalog of common actions (such as cloning a git repository, as shown in the following snippet) that can be utilized in your own pipelines:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: git-clone
spec:
  workspaces:
    - name: output
      description: "The git repo will be cloned onto the \
      volume backing this workspace"
  params:
    - name: url
      description: git url to clone
      type: string
    - name: revision
      description: git revision to checkout (branch, tag, sha, ref…)
      type: string
      default: master
    <...snip...>
  results:
    - name: commit
      description: The precise commit SHA that was fetched by this Task
```

```
  steps:
    - name: clone
      image: "gcr.io/tekton-releases/github.com/tektoncd/\
      pipeline/cmd/git-init:v0.12.1"
      script: |
        CHECKOUT_DIR="$(workspaces.output.path)/$(params.subdirectory)"
        <...snip...>
        /ko-app/git-init \
          -url "$(params.url)" \
          -revision "$(params.revision)" \
          -refspec "$(params.refspec)" \
          -path "$CHECKOUT_DIR" \
          -sslVerify="$(params.sslVerify)" \
          -submodules="$(params.submodules)" \
          -depth "$(params.depth)"
        cd "$CHECKOUT_DIR"
        RESULT_SHA="$(git rev-parse HEAD | tr -d '\n')"
        EXIT_CODE="$?"
        if [ "$EXIT_CODE" != 0 ]
        then
          exit $EXIT_CODE
        fi
        # Make sure we don't add a trailing newline to the result!
        echo -n "$RESULT_SHA" > $(results.commit.path)
```

As mentioned in previous sections, there are many different ways of building OCI images. Some of these require a Dockerfile, and some do not. You may also need to perform additional actions as part of a build. Almost all pipeline tools expose the notion of stages, steps, or tasks that allow users to configure discrete pieces of functionality that can be chained together. The following code snippet shows an example `Task` definition that uses Cloud Native Buildpacks to build an image:

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: buildpacks-phases
  labels:
    app.kubernetes.io/version: "0.1"
  annotations:
    tekton.dev/pipelines.minVersion: "0.12.1"
    tekton.dev/tags: image-build
    tekton.dev/displayName: "buildpacks-phases"
spec:
  params:
    - name: BUILDER_IMAGE
      description: "The image on which builds will run \
      (must include lifecycle and compatible buildpacks)."
    - name: PLATFORM_DIR
      description: The name of the platform directory.
      default: empty-dir
    - name: SOURCE_SUBPATH
      description: "A subpath within the `source` input \
```

```
        where the source to build is located."
        default: ""

  resources:
    outputs:
      - name: image
        type: image

  workspaces:
    - name: source

  steps:
    <...snip...>
    - name: build
      image: $(params.BUILDER_IMAGE)
      imagePullPolicy: Always
      command: ["/cnb/lifecycle/builder"]
      args:
        - "-app=$(workspaces.source.path)/$(params.SOURCE_SUBPATH)"
        - "-layers=/layers"
        - "-group=/layers/group.toml"
        - "-plan=/layers/plan.toml"
      volumeMounts:
        - name: layers-dir
          mountPath: /layers
        - name: $(params.PLATFORM_DIR)
          mountPath: /platform
        - name: empty-dir
          mountPath: /tekton/home
    <...snip...>
```

We can then tie together this task (and others) with our input repository as part of a `Pipeline` (not shown here). This involves mapping the workspace we cloned our git repository into earlier with the workspace that our buildpack builder will use as a source. We can also specify that the image be pushed to a registry at the end of the process.

The flexibility of this approach (configurable task blocks) means that pipelines become very powerful tools for defining a build flow on Kubernetes. We could add testing and/or linting stages to the build, or some kind of static code analysis. We could also easily add a signing step to our image (as described in "Image Signing" on page 438) if desired. We can also define our own tasks to run other build tools such as Kaniko or BuildKit (if not utilizing buildpacks as in this example).

Push-Based Deployments

In the previous section we saw how to automate builds in a pipeline. In this section we'll see how this can be extended to actually perform the deployment to a cluster and some of the patterns you'll want to implement to make these types of automated delivery pipelines easier.

Because of the flexibility of the task/step-based approach that we saw previously (which is present in almost every tool), it is trivial to create a step at the end of the pipeline that reads the tag of the newly created (and pushed) image and updates the image for the Deployment. This *could* be achieved by updating the Deployment directly in the cluster using `kubectl set image`, and several articles/tutorials still demonstrate this approach. A better alternative is to have our pipeline write the image tag change back into the YAML file describing the Deployment and then committing this change *back into version control*. We can then trigger a `kubectl apply` over the new version of the repository to enact the change. The latter approach is preferred as we can keep our YAML as the approximate source of truth for our cluster in that case (we'll discuss more on this in "GitOps" on page 446) but the former is an acceptable iterative step when migrating to this type of Kubernetes-native automated pipeline.

Image Tagging and Metadata

The naming and/or versioning of images is a topic that comes up periodically with clients, and we have seen some common patterns that work in most cases. When building images out of development, using the git hash as a tag works well. These git hashes are not human-readable, but with automation pipelines in place that's not usually a blocker and the hashes provide a nice trail back to the commit.

You will want to add more descriptive versioning tags (e.g., SemVer) as the pipeline progresses so it becomes easier to see at a glance what versions are deployed in your environments. Also consider using immutable tags (which most registries support) to avoid writing over existing tags and ensure some consistency when pulling images with the same tag to an environment.

It can also be really useful to add metadata (labels) to your images with contact data for the image owner and perhaps information about the build. This metadata can be reported on later on or utilized in other tooling, e.g., a policy tool that might allow some image access based on specific labels.

When deploying applications to Kubernetes we have two distinct types of artifacts to consider: the code and configuration required for the application and how to *build* it, and the configuration for how to *deploy* it. We are often asked to weigh in on how best to organize these artifacts, with some folks preferring to keep *everything* related to an application together in a single tree, and others preferring to keep them separate.

Our advice is usually to choose the latter route for the following reasons:

- Each concern is usually the responsibility of a separate domain or team in the organization. While developers should be aware of how their applications will be deployed and will have input into the process, the configuration around sizing, environments, the injection of secrets, etc. mostly sit as responsibilities of the platform or operations team.
- Security permissions and audit requirements are likely different for code repositories versus those containing deployment pipeline artifacts, secrets, and environment configurations.

Once we have our deployment configurations in a separate repository, it's easy to follow how the deployment pipeline might first checkout this repository, then run an update of the image tag (using `sed` or something similar), and finally *check the change back in to git* to ensure that is our source of truth. We can then run `kubectl apply -f` over the changed manifests. This imperative (or *push-based*) model provides great auditability as we can leverage the built-in reporting and logging capabilities provided by our version control system and easily see the changes flow through our pipeline, as shown in Figure 15-4.

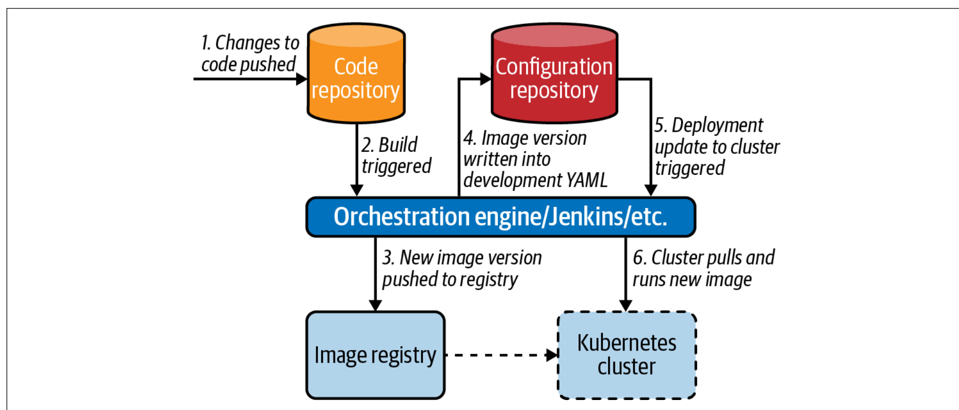


Figure 15-4. Push-based deployments.

Depending on the level of automation within your organization, you may want to have promotions between environments handled by your pipelines, and even deployments executed against different Kubernetes clusters. There are certainly ways to achieve this with most tools, and some will have better native support for this than others. However, this is an area where the imperative pipeline model described in this subsection can be more challenging to implement because we have to keep an inventory (and credentials) for each of the clusters we want to use as a target.

Another challenge of this imperative (where we have a centralized tool pushing changes out to our environments) is that if the pipeline is interrupted for some reason, we need to ensure that it is restarted or reconciled back to a healthy state. We also need to maintain monitoring and alerting on our deployment pipelines (however they are implemented) to make sure that we're aware of deployment issues if they arise.

Rollout Patterns

We mentioned briefly at the end of the previous section the need to monitor pipelines to ensure that they successfully complete. However, when deploying new versions of applications we also need a way to monitor their health and decide whether we need to resolve issues or roll back to a previous working state.

There are several patterns that organizations may want to implement. There are entire books dedicated to these patterns, but we'll cover them briefly here to show how you might implement them in Kubernetes:

Canary
: Canary releases are where a new version of an application is deployed to the cluster and a small subset of traffic (based on metadata, users, or some other attribute) is directed to the new version. This can be monitored closely to ensure that the new version (Canary) behaves the same way as the previous version, or at least does not result in an error scenario. The percentage of traffic can slowly be increased over time as confidence increases.

Blue/green
: This approach is similar to the Canary but involves more of a big bang cutover of traffic. This could be achieved over multiple clusters (one on the old version, *blue*, and one of the new version, *green*) or could be achieved in the same cluster. The idea here is that we can test that deployment of the service works as intended and perform some tests on the environment which is not user facing before cutting traffic across to the new version. If we see elevated errors, we can cut the traffic back. There is additional nuance in this approach, of course, as your applications may need to gracefully handle state, sessions, and other concerns.

A/B testing
: Again similar to the Canary, we can roll out a version of the application that may contain some different behavior that targets a subset of consumers. We can gather metrics and analysis on the usage patterns of the new version to make decisions about whether to roll back or forward, or expand the experiment.

These patterns move us toward the desired state of being able to decouple the *deployment* of our applications from their *release* to consumers by giving us control about when features and/or new versions are enabled. These practices are great at derisking the deployment of changes into our environments.

Most of these patterns are implemented via some kind of network traffic shifting. In Kubernetes, we have some very rich networking primitives and capabilities that make the implementation of these patterns possible. One open source tool that enables these patterns (on top of various service mesh solutions) is Flagger. Flagger runs as a controller within the Kubernetes cluster and watches for changes to the image field of Deployment resources. It exposes many tweakable options to enable the preceding patterns by programmatically configuring an underlying service mesh to appropriately shift traffic. It also adds the ability to monitor the health of newly rolled-out versions and either continue or halt and revert the rollout process.

We definitely see the value of looking into Flagger and other solutions in this space. However, we see these approaches more typically broached as part of a second or third phase of an organization's Kubernetes journey, given the additional complexity they both depend on (service mesh is required to enable most patterns) and introduce.

GitOps

So far we have looked at adding a push-based deployment stage into your delivery pipelines for Kubernetes. An emerging alternative model in the deployment space is GitOps. Rather than imperatively push out changes to the cluster, the GitOps model features a controller that constantly reconciles the contents of a git repository with resources running in the cluster, as shown in Figure 15-5. This model brings it closely in line with the control-loop reconciliation experience that we get with Kubernetes itself. The two primary tools in the GitOps space are ArgoCD and Flux, and both teams are working together on a common engine to underpin their respective tools.

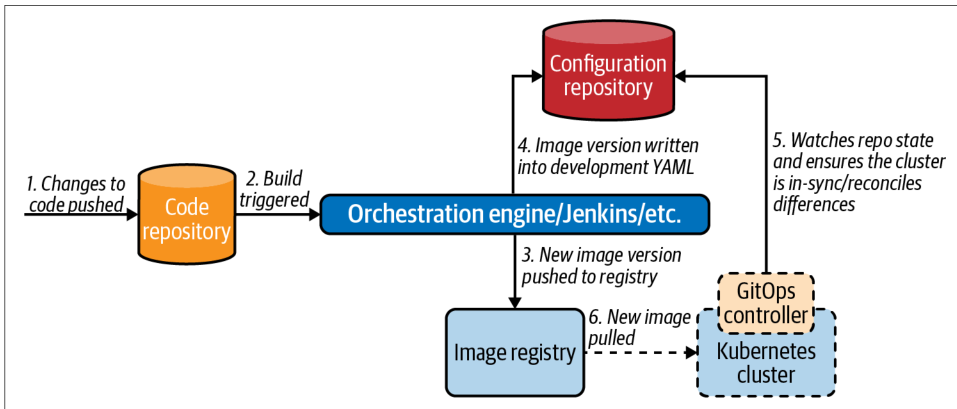


Figure 15-5. GitOps flow.

There are a couple of major benefits to this model:

- It is declarative in nature, so that any issues with the deployment itself (tooling going down, etc.) or deployments being deleted ad hoc will result in (attempted) reconciliation to a good state.
- Git becomes our single source of truth, and we can leverage existing expertise and familiarity with the tooling, in addition to getting a strong audit log of changes by default. We can use a pull request workflow as a gate for changes to our clusters and integrate with external tooling as required through the extension points that most version control systems expose (webhooks, workflows, actions, etc.).

However, this model is not without downsides. For those organizations that truly want to use git as their *single* source of truth, this means keeping secret data in version control. Several projects have emerged over recent years to solve this problem, with the most well-known of these being Bitnami's Sealed Secrets. That project allows encrypted versions of Secrets to be committed to a repository and then decrypted once applied to the cluster (so as to be available to applications). We discuss this approach more in Chapter 5.

We also need to ensure that we are monitoring the current health of state synchronization. If the pipeline was push-based and fails, we'll see a failure in the pipeline. However, because the GitOps approach is declarative, we need to make sure we get alerted if the observed state (in cluster) and the declared state (in git) remain diverged for an extended period of time.

Increased embracing of GitOps is a trend we see in the field, although it's definitely a paradigm shift from more traditional push-based models. Not all applications deploy cleanly as-is with a flat application of YAML resources, and it may be necessary to build in ordering and some scripting initially as organizations transition to this approach.

It's also important to be cognizant of tools that may create, modify, or delete resources as part of their life cycle as these sometimes require some massaging to fit into the GitOps model. An example of this would be a controller that runs in the cluster and watches for a specific CRD and then creates multiple other resources directly via the Kubernetes API. If running in *strict* mode, GitOps tooling may delete those dynamically created resources as they are not in the single source of truth (the git repo). This deletion of *unknown* resources may, of course, be desirable in most cases and is one of the *positive* attributes of GitOps. However, you should absolutely be mindful of situations where changes may intentionally occur out of band from the git repository that may break the model and need to be worked around.

Summary

In this chapter we looked at the process of getting source code into a container and deployed onto a Kubernetes cluster. Many of the stages and principles that you will already be familiar with (build/test, CI, CD, etc.) still apply in a container/Kubernetes environment but with different tooling. At the same time some concepts (like GitOps) will likely be new, building on concepts that are present in Kubernetes itself to enhance reliability and security within existing deployment patterns.

There are many tools in this space that can enable many different flows and patterns. However, one of the key takeaways from this chapter should be the importance of deciding how much (or little) to expose each part of this pipeline to different groups in the organization. Maybe development teams are engaged with Kubernetes and confident enough to write build and deployment artifacts (or at least have significant input). Or maybe the desire is toward abstracting all the underlying details away from development teams to ease scaling and standardization concerns, at the expense of additional burden on the platform teams to put relevant foundations and automation in place.

CHAPTER 16

Platform Abstractions

Many times we have seen organizations adopt a *build it and they will come* approach to designing and building a Kubernetes platform. However, this philosophy is usually fraught with risk as it often fails to deliver on the key requirements of the many teams (e.g., development, information security, networking, etc.) that will be interacting with the platform, resulting in rework and additional effort. It's important to bring the other groups along on the journey and ensure that the platform constructed is fit for purpose.

In this chapter we'll cover some of the angles you should consider when designing the onboarding and usage experience for other teams (specifically developers) to your Kubernetes platform. First we'll look at some of the more philosophical aspects and ask the question, How much should developers know about Kubernetes? Then we'll move on to discuss how we can build a smooth onramp for developers to get started deploying to Kubernetes and deploying clusters themselves. Finally, we'll revisit the complexity spectrum we mentioned in Chapter 1 and look at some of the levels of abstractions we can put in place. Our objective is to strike a good balance between complexity and flexibility when offering a Kubernetes platform to development teams that have varying degrees of knowledge and desired engagement with the underlying implementation.

Many of the areas in this chapter are covered elsewhere in the book, which we'll call out where appropriate. Here we aim to cover aspects from the specific stance of increasing team collaboration and building a platform that serves the needs of everyone in the organization. Although on the surface this might seem a light topic, the issues discussed within are often some of the hardest hurdles for many companies to overcome and can make or break the successful adoption of a Kubernetes-based application platform.

Platform Exposure

We've talked many times in this book about the need to evaluate your individual requirements and ask questions in different areas when designing and implementing a Kubernetes platform. One major question that will inform many choices is deciding how much exposure you want development teams to have to the underlying Kubernetes systems and resources. There are several factors that will impact this decision.

Kubernetes is a relatively new technology. In some cases the drive to adopt Kubernetes will come from the infrastructure side of the house, to simplify infrastructure usage and efficiency, or standardize workloads. In other cases the drive may be from development teams that are keen to implement a new technology that they feel can accommodate and accelerate their development and deployment of cloud native applications. Wherever the drive is coming from, impact will be felt in the other teams, whether it be adapting to a new paradigm, learning new tools, or just a change in user experience when interacting with a new platform.

In some organizations, there is a strong requirement that development teams should not be exposed to the underlying platform. The driver for this is the belief that developers should focus on delivering business value and not be distracted by the implementation details of the platform being developed. There is some value to this approach, but in our experience, we don't always completely agree with it. For example, it's necessary for developers to have at least *some* understanding of the base platform in order to effectively develop applications that target it. This doesn't mean increasing the coupling of the application and the platform, but purely understanding how to maximize the platform capabilities. Chapter 14 covers this application and platform relationship in more detail.

For the approach of *no developer exposure* to be successful, there must be sufficient capacity in the platform team. Firstly because they will be solely responsible for maintaining and supporting the environments, and secondly, that team will also be responsible for building the necessary abstractions required for developers to seamlessly interact with the platform. This point is important, as even when developers are not directly exposed to Kubernetes, they will still need ways of analyzing application performance, debugging issues, and troubleshooting. If giving developers `kubectl` access to a cluster exposes too much underlying detail, there needs to be an intermediate layer that allows developers to *own* the application into production while simultaneously not overwhelming them with implementation details. In Chapter 9 we cover many of the main ways to expose debugging tools to development teams in an effective way.

Simply streamlining the troubleshooting experience for developers may not be enough in some organizations. Deploying applications to Kubernetes can also be complex, potentially requiring many components. Maybe an application needs a

StatefulSet, PersistentVolumeClaim, Service, and ConfigMap to be successfully deployed. When exposing these concepts to developers is not desirable, platform teams may go a step further and create abstractions in this area. This could be achieved through self-service pipelines or building custom resources (this is covered in Chapter 11) to more simply encapsulate the required pieces. We'll cover both of these approaches later in this chapter.

A limiting factor when deciding how much of the platform to expose is the skillset and experience of the teams creating the abstractions. For example, platform teams will need to have some development skills and knowledge of Kubernetes API best practices if they want to go down the custom resource route. If they don't, you may be limited in what abstractions you can build and as such have to expose more of the platform internals to development teams.

In the next section we'll look at some of the ways we've seen teams offer a self-service model to developers (and other end users) in order to streamline and standardize the deployment of both clusters and applications.

Self-Service Onboarding

Early on in an organization's Kubernetes journey it's likely that platform teams will be responsible for the provisioning and configuration of clusters for all the teams that need them. They'll probably also be responsible for at least helping with the deployment of applications to those clusters. Depending on the tenancy model adopted (read more about workload tenancy in Chapter 12), there will be different requirements in this setup process. In a single-tenant model, cluster provisioning and configuration may be more straightforward, with a set of common permissions, core services (logging, monitoring, ingress, etc.), and access (e.g., Single Sign-On) setup. However, in a multitenant cluster we may need to create multiple additional components (e.g., Namespaces, Network Policies, Quotas, etc.) for each team and application onboarded.

However, as the organization begins to scale, provisioning and configuring manually is not sustainable. It represents repetitive toil for the platform team and blocks the development teams waiting for manual tasks to complete. Once a base level of maturity has been reached, we usually see teams start to offer some kind of self-service offering to their internal users. An effective way of providing this is through an existing CI/CD tool or process like Jenkins or GitLab. Both tools allow the easy creation of pipelines and provide the capability for additional customized inputs at execution time.

The maturity of tools like `kubeadm` and Cluster API make the automation of cluster creation relatively straightforward and consistent. Teams can expose tweakable parameters like cluster name and sizing, for example, and the pipeline can invoke

those tools to provision clusters with sensible defaults before outputting appropriate credentials or access to the requesting team. Like many things, this automation can be as sophisticated as you choose to make it. We have seen pipelines that create load balancers and DNS automatically based on the requesting user's LDAP information, including automatically tagging the underlying infrastructure with the relevant cost centers. Sizing can be open to the user but constrained with certain ranges depending on the team, environment, or project. We can even choose whether to provision in the private or public cloud depending on the classification or security profile of the application. There is a wide array of possibilities for platform teams to create a flexible yet powerful automated provisioning process for development teams.

For multitenant scenarios we'll not be creating clusters but rather creating Namespaces and all of the associated objects that allow us to provide a soft-isolation environment for the new application. Again, we can use a similiar pipeline approach, but this time allow development teams to choose the cluster (or clusters) where their application will be deployed to. At a base level we'll want to generate the following:

Namespace
: For the application to reside in and to provide the logical isolation for our other components to build on.

RBAC
: To ensure that only the appropriate authorized groups can access resources in their own application's Namespace.

Network policies
: To ensure that the application is only allowed to communicate with itself or other shared cluster services, but *not* other applications on the same cluster (if required).

Quotas
: To limit the amount of resources that one Namespace or application may consume in the cluster, reducing the potential for a *noisy neighbor* situation to arise.

Limit ranges
: To set sensible defaults for specific objects created in the Namespace.

Pod Security Policies
: To ensure workloads conform to sensible default security settings, such as not running as a root user.

Not all of these are required or necessary in every scenario, although combined they allow cluster administrators and platform teams to create a seamless onboarding experience and deployment environment for new development teams without requiring manual intervention.

As organizations mature in their usage and knowledge of Kubernetes, these pipelines can be implemented in a Kubernetes native way using operators. For example, we might define a `Team` resource in the following structure:

```yaml
apiVersion: examples.namespace-operator.io/v1
kind: Team
metadata:
  name: team-a
spec:
  owner: Alice
  resourceQuotas:
    pods: "50"
    storage: "300Gi"
```

In this example we might define a specific `Team` that we want to be onboarded with an owner (user) and some resource quotas. Our controller that lives in the cluster would be responsible for reading this object and creating the relevant Namespace, RBAC resources, and quotas and tying them together. This approach can be powerful because it allows us to tie in closely with the Kubernetes API and expose a native way of managing and reconciling resources. For instance, if a role were to be accidentally deleted or a quota were to get modified, the controller would be able to automatically remediate the situation. These higher-level types of resources (like a `Team` or `Application`) can be great for bootstrapping a cluster also, but just adding several team objects and our controller we're able to automate all of the relevant configuration ready for use.

We can definitely dig deeper down this rabbit hole to produce a sophisticated automation setup. For instance, let's think about some of the observability tooling that might need to be configured for new applications. Perhaps we could have our team controller generate and submit customized dashboards for a new team or application and have Grafana automatically reload them. We might dynamically add new alert targets in Alertmanager for new teams or Namespaces. We can create very powerful functionality behind these simpler, more user-friendly onboarding abstractions.

The Spectrum of Abstraction

In Chapter 1 we introduced the idea of a *spectrum of abstraction*. In Figure 16-1 we have expanded on that original concept and added some concrete levels of abstraction along the spectrum.

In the preceding sections we talked about some of the philosophical decisions and organization constraints that might influence where on this spectrum you might land. In this section we'll walk through this more detailed spectrum from left (no abstractions) to right (fully abstracted platform) and discuss some of the options and tradeoffs as we go.

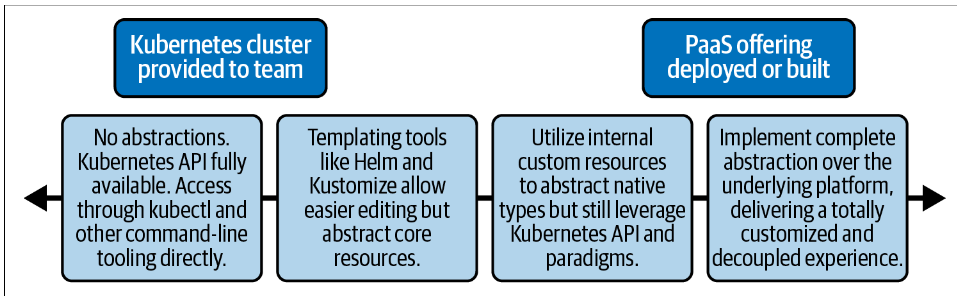


Figure 16-1. Spectrum of abstraction.

Command-Line Tooling

By exposing the Kubernetes API through native command-line tooling we are at the far-left end of the spectrum with no abstractions in place. In some organizations `kubectl` will be the primary point of entry to Kubernetes for developers. This might be due to constraints (lack of available support from the platform teams) or choice (familiarity with and desire to work directly on Kubernetes from the developers). There may still be some automation or guardrails in place on the cluster, but developers will interact with it using native tooling.

There are some downsides to this approach (even if your development teams *are* a little familiar with Kubernetes):

- The manual overhead of having to set up and configure the authentication methods for potentially multiple clusters can be cumbersome. This includes switching contexts between multiple clusters and ensuring that individuals are always targeting the intended cluster.
- The output format of `kubectl` commands can be cumbersome to view and work with. By default we are given tabular output, but this can be marshaled into different formats and piped to external tooling like `jq` to more concisely display the information. However, this requires that developers know about `kubectl`'s options and how to use them (in addition to the external tooling).
- Raw `kubectl` opens all the tweaks and knobs of Kubernetes to the user without abstraction/mediation. As such we need to ensure not only that there are appropriate RBAC rules in place to restrict unauthorized access, but also that there is a layer of admission control vetting all the requests coming into the API server.

Various tools can enhance this experience. There are many `kubectl` plug-ins that can provide a better user experience in the local shell, such as `kubens` and `kubectx`, that provide better usability and visibility into Namespaces and contexts, respectively. There are plug-ins that will aggregate logs from multiple Pods, or provide a terminal UI for application health. While not advanced tools, they can remove common pain

points and eliminate the need for developers to know the intricacies of some of the underlying implementation details. These additional tools are useful helpers, but we're still exposing the Kubernetes API directly with barely any abstraction on top.

There are also plug-ins that tie into external auth systems to streamline the authentication flow to abstract the complexity of kubeconfigs, certificates, tokens, etc., away from the user. This is an area where we regularly see some augmentation of the vanilla tooling, as enabling developers to have secure access to multiple clusters (especially those that may come and go dynamically) can be challenging. In noncritical environments, access may be based on key pairs (which must be generated, managed, and distributed), whereas in more stable environments access is likely to be linked to a Single Sign-On system. We have developed command-line utilities for several clients to pull credentials from a central cluster registry based on a local user's login credentials.

Additionally, you may decide to go down the route of Airbnb. In a recent QCon talk, Melanie Cebula shared Airbnb's approach of building out more advanced toolsets both as standalone binaries and `kubectl` plug-ins to interact with its clusters, hook into image building, deployment, and more.

There is an additional class of tooling that allows developers a graphical interface to interact with the cluster. Recent popular choices here are Octant and Lens. Rather than sitting in the cluster as the Kubernetes dashboard does, these tools run locally on a workstation and utilize a `kubeconfig` to access the cluster. These tools can be a great onramp for developers newer to the platform who want to see a visual representation of the cluster and their applications. Enhancing the client-side experience is the first step that organizations can take to simplify developer interactions with Kubernetes.

Abstraction Through Templating

Deploying a single application to Kubernetes can require the creation of multiple Kubernetes objects. For example, a *simple* Wordpress application may need the following:

Deployment
: For describing the image, commands, and properties of the Wordpress instance.

StatefulSet
: For deploying MySQL as a datastore for Wordpress.

Services
: To provide discovery and load balancing for both Wordpress and MySQL.

PVC
: To dynamically create a Volume for the Wordpress data.

ConfigMaps
: To hold configuration for both Wordpress and MySQL.

Secrets
: To hold admin credentials for both Wordpress and MySQL.

In this list we have nearly 10 different objects all to support an extremely small application. Not only that, but there are nuances and expert knowledge needed to configure them. For example, when using a StatefulSet we need to create a special *headless* Service to front it. We want our developers to be able to deploy their application to the cluster *without* having to know how to create and configure all of these different Kubernetes object types themselves.

One of the ways we can ease the user experience when deploying these applications is by only exposing a small set of inputs and generating the rest of the boilerplate behind the scenes. This approach doesn't require developers to know about all the fields in all the objects, but it still exposes some of the underlying objects and uses tools that are a level up from pure kubectl. The tools that have some maturity in this area are templating tools like Helm and Kustomize.

Helm

Helm has become a popular tool in the Kubernetes ecosystem over the past couple of years. We realize that it has capabilities over and above just templating, but in our experience have found the templating use case (followed by editing and application of manifests) more compelling over some of its life cycle management features.

Following is a snippet from the Wordpress Helm chart (package describing an application) describing a Service:

```
ports:
  - name: http
    port: {{ .Values.service.port }}
    targetPort: http
```

This template is not directly exposed to developers but is a template that will use an injected or defined value from elsewhere. In the case of Helm, this can be passed on the command line or more commonly through a values file:

```
## Kubernetes configuration
## For minikube, set this to NodePort, elsewhere use LoadBalancer or ClusterIP
##
service:
  type: LoadBalancer
  ## HTTP Port
  ##
  port: 80
```

Charts contain a default *Values.yaml* file with sensible settings, but developers can provide an override where they modify only the settings they need. This allows powerful customization via the templating without needing in-depth knowledge. Rather than purely templating values, Helm also contains functionality for basic logic operations, allowing a single tweak in the values file to generate or modify large sections in the underlying templates.

For instance, in the example values file just shown, there is a `type: Loadbalancer` declaration. This is injected directly into the template in several places, but it is also responsible for triggering more complex templating through the use of conditionals and built-in functions, as shown in the following code snippet:

```
spec:
  type: {{ .Values.service.type }}
  {{- if (or (eq .Values.service.type "LoadBalancer")
    (eq .Values.service.type "NodePort")) }}
  externalTrafficPolicy: {{ .Values.service.externalTrafficPolicy | quote }}
  {{- end }}
  {{- if (and (eq .Values.service.type "LoadBalancer")
    .Values.service.loadBalancerSourceRanges) }}
  loadBalancerSourceRanges:
  {{- with .Values.service.loadBalancerSourceRanges }}
{{ toYaml . | indent 4 }}
  {{- end }}
  {{- end }}
```

This inline logic may look complex and certainly has its detractors. However, the complexity is owned by the *creators* of the charts, and not the end-user development teams. The construction of the complex YAML structures in the template is keyed from a single `type` key in the values file, which is the interface for the developer to modify the configuration. The values file can be specified at runtime so different files (with different configurations) can be used for different clusters, teams, or environments.

Implementing Helm for configuring and deploying both third-party and internal applications can be an effective first step to abstracting some of the underlying platform from developers and allowing them to focus more closely on only the options they need. However, there are still some disadvantages. The interface (*Values.yaml*) is still YAML and can be an unfriendly user experience if developers need to explore the templates to understand the impact of a change (although good documentation can mitigate this).

For those who want to go a step further, we've seen tools developed that will abstract these tweakable items to a user interface. This allows a more native approach under the hood, but the user experience can be customized depending on the requirements of the audience. For example, workflows can be built into an existing deployment tool (like Jenkins) or a ticketing type of service, but the underlying output can still be

Kubernetes manifests that are then applied to a cluster. While powerful, these models can get complex to maintain, and the abstractions can eventually leak through to the user.

An interesting take on this model that has recently appeared is the K8s Initializer by Ambassador Labs. Using a browser-based UI workflow, the user is asked multiple questions about the type of service they want to deploy and the target platform. The site then outputs a downloadable package for the user to apply to the cluster with all the customizations applied.

All the templating approaches have many of the same strengths and weaknesses. We are still dealing with Kubernetes native objects that are applied to the cluster. For example, when outputting our Helm files with completed values we're still exposed to Services, StatefulSets, and more. This isn't a complete abstraction of the platform, so developers are still required to have *some level* of underlying knowledge. However, on the flip side, that's also an *advantage* of this approach (either with Helm, or the more abstracted approach from K8s Initializer). If upstream Helm charts or the Initializer do not output exactly what we need, we still have the full flexibility to modify the results before applying to the cluster.

Kustomize

Kustomize is a flexible tool that can be used standalone or as part of `kubectl` to apply abitrary additions, deletions, and modifications to fields in any Kubernetes YAML objects. It is not a templating tool but is useful when leveraged on a set of manifests that have been templated by Helm as a method of modifying fields that Helm does not otherwise expose.

For the reasons previously discussed, we have seen Helm as a templating tool piped into something like Kustomize for additional customizations as a very powerful abstraction that also allows full flexibility. This approach sits somewhere in the middle of the spectrum and is often a sweet spot for organizations. In the next section we'll move further to the right on the abstraction spectrum and see how we can start encapsulating underlying objects with custom resources tailored specifically to each organization/use case.

Abstracting Kubernetes Primitives

As we've spoken about many times in this book, Kubernetes provides a set of primitive objects and API patterns. In combination these allow us to build higher-level abstractions and custom resources to capture types and ideas that are not built in. In late 2019 the social media company Pinterest published an interesting blog post describing how it had created CRDs (and associated controllers) to model its internal workloads as a way of abstracting Kubernetes native building blocks away from its development teams. Pinterest summarized its rationale for this approach as such:

> On the other hand, the Kubernetes native workload model, such as deployment, jobs and daemonsets, are not enough for modeling our own workloads. Usability issues are huge blockers on the way to adopt Kubernetes. For example, we've heard service developers complaining about missing or misconfigured Ingress messing up their endpoints. We've also seen batch job users using template tools to generate hundreds of copies of the same job specification and ending up with a debugging nightmare.
>
> —Lida Li, June Liu, Rodrigo Menezes, Suli Xu, Harry Zhang, and Roberto Rodriguez Alcala; "Building a Kubernetes platform at Pinterest"

In the following code snippet, `PinterestService` is an example of Pinterest's custom internal resources. The 25-line object creates multiple Kubernetes native objects that would equate to more than 350 lines if created directly:

```
apiVersion: pinterest.com/v1
kind: PinterestService
metadata:
  name: exampleservice
  project: exampleproject
  namespace: default
spec:
  iamrole: role1
  loadbalancer:
    port: 8080
  replicas: 3
  sidecarconfig:
    sidecar1:
      deps:
      - example.dep
    sidecar2:
      log_level: info
  template:
    spec:
      initcontainers:
      - name: init
        image: gcr.io/kuar-demo/kuard-amd64:1
      containers:
      - name: init
        image: gcr.io/kuar-demo/kuard-amd64:1
```

This is an extension of the templating model we saw in the previous section where only certain inputs are exposed to the end user. However, in this case we can construct an input object that makes sense in the context of the application (rather than a relatively unstructured *Values.yaml* file) and be more intuitively understood by the developers. While it's still possible for leaky abstractions to occur with this approach, it's less likely as the platform team (creating the CRDs/operator) have full control of how to create and modify the underlying resources rather than having to work within the constraints of the existing objects as with the Helm approach. They also have the ability (with the controller) to craft much more sophisticated logic through a general-purpose programming language instead of being limited by Helm's built-in functions.

However, as we discussed earlier, this comes with the trade-off that platform teams must now have programming expertise. For more depth on creating platform services and operators take a look at Chapter 11.

By utilizing an operator, we can also call out to external APIs to integrate richer functionality into our abstracted object types. For example, one client had an internal DNS system that all applications needed to be registered with to work correctly and be exposed to external clients. The incumbent flow would have developers visit a web portal and manually add the location of their service and the ports they needed forwarded from their assigned DNS name. We have a couple of options to enhance the developer experience.

If we're utilizing native Kubernetes objects (like Ingress in this case), we can create an operator that will read a special annotation on the applied Ingress and automatically register the application with the DNS service. This might look like the following:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app
  annotations:
    company.ingress.required: true
spec:
  rules:
  - host: "my-app"
    http:
      paths:
      - path: /
        backend:
          service:
            name: my-app
            port:
              number: 8000
```

Our controller would read the `company.ingress.required: true` annotation, and depending on the name of the application, the Namespace or some other metadata could go and register the appropriate DNS records, as well as potentially modifying the host field depending on certain rules. While it reduces a lot of the manual work (of creating the records) required by the developer, it still requires some knowledge/creation of Kubernetes objects (in this case, the Ingress). In that way, it is more in line with the level of abstraction described in the previous section.

Another option is to use a custom resource like the `PinterestService`. We have all the information we need encapsulated there, and we can create the Ingress via our operator, as well as configuring external services like the DNS system. We haven't leaked any of the underlying abstraction through to the developer and have full flexibility with our implementation.

Supporting Differing Levels of Abstraction

When deciding on the right level of abstractions to offer with your organization, it's important to think about how you'll document and support them. For instance, when using something like Helm (or other templating tools) or an approach that distills down to raw Kubernetes objects (Pods, PVCs, ConfigMaps, etc.) you are able to leverage a huge amount of community resources when troubleshooting.

An example of this can be a status condition on a PVC describing why storage couldn't be bound, or perhaps an error message coming from Helm when trying to install a Chart with a misconfigured template. Both of these occurrences can be easily searched online due to the mature and extensive documentation and shared community experience with those commonly utilized tools and objects.

When fully abstracting away those objects (either behind pipelines where developers don't have raw debugging acccess or behind custom resource types like `PinterestService`), it can be a lot more difficult for development teams to tap into those shared community sources if the scope of usage is very narrow (potentially just internal to your organization). In these cases good documentation is essential, but also providing a *window* to those underlying internals can be useful in break-glass situations.

The break-glass approach is especially powerful as it allows end users to experience a default high level of abstraction, minimizing day-to-day friction, but be able to dive in and choose to consume a different level of abstraction if required or desirable (based on individual skillsets or knowledge). In our experience this model should be the ideal model when designing platform abstractions.

Another relevant aspect here is the *transferability* of skills. As upstream Kubernetes skills become more commonplace, it will be easier for those people to interact with and troubleshoot the platform if the underlying internals are accessible or exposed. It also may be more advantageous to have a more vanilla platform (or at least one where the layers are accessible) to attract talent who may not want to get deeply ingrained in any specific downstream platform/distribution for fear of narrowing their skillset.

Even with the custom resource and operator approach we've discussed in this section, we are still exposing some of the core mechanics of the platform to development teams. We need to specify valid Kubernetes metadata, API versions, and resource types. We also expose YAML (unless we're also providing a pipeline, wrapper, or UI to build it) and the quirks associated with it. In the next (and final) section we'll move fully to the right on our abstraction spectrum and talk about some of the options that allow developers to go directly from their application code to the platform and not even necessarily know about Kubernetes at all.

Making Kubernetes Invisible

In previous sections we've been moving from left to right on the spectrum of abstraction, starting with the least abstracted (raw `kubectl` access) and now ending with a fully abstracted platform. In this section we'll talk about cases and tooling where developers don't even know they're using Kubernetes and whose only interface to the platform (more or less) is committing/pushing code, allowing them to retain a fairly narrow (and deep) focus without being exposed to platform nuances.

SaaS providers like Heroku and tools like Cloud Foundry popularized the developer-focused *push* experience over 10 years ago. The concept is that the tooling (once configured) would provide a platform as a service (PaaS, now a nebulous term) that contained all of the necessary complementary components for an application to function (observability stacks, some form of routing/traffic management, software catalogs, etc.) and would allow developers to *simply* push code to a source repository. Specialized components within the platform would set appropriate resource limits, provision (if necessary) environments for the code to run, and plumb together the standard PaaS components to enable a streamlined end-user experience.

You might be thinking that there is some crossover with Kubernetes here, which also provides us some primitives to enable some similar functionality. When the original PaaS platforms were built, Docker and Kubernetes didn't exist and the prevalence of more rudimentary containerized workloads was very limited. Hence, these tools were built from the ground up for a virtual machine–based environment. We are now increasingly seeing these tools (and other new ones) be ported to or rewritten for Kubernetes for exactly the reason we identified earlier. Kubernetes provides very strong mechanical foundations, API conventions, and raw primitives to *build* these higher-level platforms atop of it.

One of the criticisms that is often leveled at Kubernetes is that it introduces a non-trivial amount of additional complexity into an environment, both for operations and development teams (on top of the paradigm shift to containers that must also be negotiated). However, this perspective misses one of the primary aims of Kubernetes which (as cofounder Joe Beda has articulated many times) is to be a platform *for building platforms*. Complexity will always exist somewhere, but through its architectural decisions and primitives, Kubernetes allows us to abstract the complexity to platform developers, vendors, and the open source community for them to build seamless development and deployment experiences *upon* Kubernetes.

We already mentioned Cloud Foundry, which is probably the most popular and successful open source PaaS (now ported to Kubernetes), and there are other fairly mature options like Google App Engine (and some other serverless technologies) and parts of RedHat OpenShift. In addition to these we're seeing more platforms appear as the space matures. One such popular platform is Backstage, originally created by Spotify. Now a CNCF Sandbox project, it is a platform for building portals that

provide tailored abstractions for developers to deploy and manage applications. Even as we write this chapter, HashiCorp (developer of many cloud native OSS tools like Vault and Consul) has just announced Project Waypoint, a new tool to separate end users from the underlying deployment platforms and provide a high-level abstraction for development teams. In their announcement blog post they wrote:

> We built Waypoint for one simple reason: developers just want to deploy.
>
> —Mitchell Hashimoto, "Announcing HashiCorp Waypoint"

Waypoint aims to encapsulate the build, deploy, and release stages of software development. With Waypoint the developers still have to create (or have help creating) a configuration file that describes their process, akin to a Dockerfile except it describes the full set of stages in a minimal way, soliciting only the essential inputs. An example of this configuration is as follows:

```
project = "example-nodejs"

app "example-nodejs" {
  labels = {
      "service" = "example-nodejs",
      "env" = "dev"
  }

  build {
    use "pack" {}
    registry {
        use "docker" {
          image = "example-nodejs"
          tag = "1"
          local = true
        }
    }
 }

  deploy {
    use "kubernetes" {
    probe_path = "/"
    }
  }

  release {
    use "kubernetes" {
    }
  }
}
```

Note that Waypoint's approach is still to push *some* complexity onto the developer (writing this file); however, they have abstracted a huge number of decisions away. Abstracting the platform doesn't always mean that all complexity is removed from the process, or that no one has to learn anything new. Instead, as in this case, we can

introduce a new, simplified interface *at the right level* of abstraction that hits the sweet spot of speed and flexibility. In Waypoint's case even the underlying platform can be switched out in the deploy and release stages to use something like Hashicorp's own Nomad, or some other orchestration engine. All of the underlying details and logic are abstracted by the platform. As Kubernetes and these other platforms evolve and become more stable and *boring* (some would argue we are nearly there), then the real innovation will continue in the development of higher-level platforms to better enable development teams to deliver more rapid business value.

Summary

In this chapter we've discussed the different layers of abstraction that platform teams can offer to their users (usually development teams) and the common tools and patterns we've seen used to implement them. Probably more than any other area, this is where organizational culture, history, tooling, skillsets, and more will all inform any decisions and trade-offs that you choose, and almost every client we've worked with has chosen to solve the issues described in this chapter in slightly different ways.

It's also important to note that while we often have espoused the value of having development teams not having to concern themselves too much with the underlying deployment platform, that is *not* to say that this is always the right choice or that developers should never understand where and how their applications are running. This information is key to being able to leverage specific features of the platform, for instance, or being able to debug issues with their software. As always, maintaining a strong balance is often the most successful way forward.

Index

A

A/B testing, 446
abstractions, 449
  (see also platform abstractions)
  considerations in application platform built on Kubernetes, 15
  developer, application platform on Kubernetes, 20
  spectrum of, 453
access control
  configuring access to storage provider, 90
  service mesh features, 169
  webhooks
    design considerations, 227
access modes for storage, 80
add-ons (cluster), 50-52
admin role, 359
admission control, 20, 219-241
  admission webhooks, 345
  centralized policy systems, 234-241
  extensions, 316
  in-tree admission controllers, 222
  Kubernetes admission chain, 220-221
  webhooks, 223-228, 361
    configuring webhook admission controllers, 225-227
  writing a mutating webhook, 228-234
    controller runtime, 231-234
    plain HTTPS handler, 229-231
AdmissionReview objects, 224
Alertmanager, 266
alerts
  from logs, 250
  from metrics, 255-257
  when alerting and metrics systems go down, 256
allowed-ingress-pattern annotation, 238
Amazon Elastic File System (EFS), 81
Amazon Elastic Kubernetes Service (EKS), 305
Amazon S3, 82
Antrea, 126
API Priority and Fairness feature, 362
API server, 3, 356
  access to resources through, 20
  admission control and, 219, 454
    in-tree admission controllers, 222
    Kubernetes admission chain, 220
    webhooks, 223
  and different types of machines in a cluster, 42
  application configurations
    ConfigMaps and Secrets, 400
    getting from the server, 402
  application deployments and, 398
  authentication to, 274
    AWS IRSA, 303
    Service Account for workloads, 296
    using PSATs, 297
    using shared secrets, 275
  and Client API consumption of secrets, 195
  controllers and, 318
  custom resources and, 319
  Endpoints resource updates, 137
  horizontal Pod autoscaling, 381
  Ingress controllers and, 157
  installation of cluster add-ons, 51
  IP address management for Services, 129

Kubebuilder facilitating interactions with, 326
in larger clusters, 40
load balancing, 44
multitenancy and
API Priority and Fairness, 362
enforcing policy using admission webhooks, 361
preventing resource modification by tenants, 358
operator extensions and, 316
resource deletions and, 346
scheduling policy updates, 348
secret data moving through network, 190
service discovery through DNS, 148
submitting nonencoded string data to, 192
support for encrypting secrets at rest, 198
webhook extensions and, 315
app-specific metrics, 416
application architecture, autoscaling and, 379
application encryption, 190
application forwarding of logs, 245
application instrumentation for tracing, 272
application platforms
building on Kubernetes, 12-20
abstraction spectrum, 15
building blocks, 17-20
determining platform services, 16
starting from the bottom, 13
building platform services (see platform services, building)
defining, 7-12
aligning your organizational needs, 10
approaches to, 8
disadvantages of prebuilt platforms, 8
application-specific operators, 324
application/workload identity, 288-311
network identity, 289-293
platform mediated Node identity, 299-311
AWS platform authentication methods/tools, 300-305
cross-platform identity with SPIFFE and SPIRE, 305-311
Projected Service Account tokens, 297-299
Service Account tokens, 293-296
shared secrets, 289
applications on Kubernetes, 397-423
application logs, 415-416
considerations, 397
deploying applications to Kubernetes, 398-399
exposing metrics, 416-419
app-specific metrics, 419
four golden signals, 419
instrumenting applications, 417-418
RED method, 419
USE method, 419
handling rescheduling events, 404-408
graceful container shutdown, 405
pre-stop container life cycle hook, 404
satisfying availability requirements, 407
ingesting configuration and secrets, 400-404
configuration from external systems, 403
instrumenting for distributed tracing, 420-423
creating spans, 421
initializing the tracer, 420
propagating context, 422
Pod resource requests and limits, 413-415
state probes, 408-413
implementing, 412
liveness probes, 409
readiness probes, 410
startup probes, 411
architecture and topology, 28-35
cluster federation, 32-35
federated software deployment, 35
management clusters, 33
observability, 34
cluster tiers, 29
etcd deployment models, 28
node pools, 31
attack vectors, new, understanding, 200
audit logs, 247-249
audit policy, 247
forwarding to a backend, 249
auditing secret interaction, 215
authentication, 273
(see also identity)
extensions, 316
methods of, 275-285
OpenID Connect, 283
public key infrastructure, 277-283
shared secrets, 275
methods provided by Vault, 207
requests to API server, 315
authorization, 20, 273
establishing identity before, 274

requests for secrets in Vault, 207
automation
in deployments, 26-27
infrastructure, 44-46
of infrastructure management, 36
integration of automated components, 313
triggering mechanisms for, 60
autoscaling, 377-395
application architecture, 379
for applications fluctuating in load and traffic, 378
cluster, 33, 377, 389-395
benefits and concerns with, 392
Cluster Autoscaler, 389-392
cluster overprovisioning, 393
of DNS server deployment, 152
primary motivations for, 378
types of scaling, 378
workload, 377, 380-389
Cluster Proportional Autoscaler, 388
custom autoscaling, 389
Horizontal Pod Autoscaler, 380-384
using custom metrics, 387
Vertical Pod Autoscaler, 384-387
autoscaling/v2beta2 API, 382
availability requirements, 407
availability zones (AZs), 43
AWS (Amazon Web Services), 299
Cluster Autoscaler Deployment manifest targeting, 391
disk encryption, 189
Kubernetes built-in controller for, 132
SPIRE integration with, 311
VPC CNI, 108, 123
AWS Elastic Block Storage (EBS), 80, 89
encryption of, 189
offering EBS volumes to Pods dynamically, 86
AWS platform authentication methods/tooling, 300-305
IAM Roles for Service Accounts, 229, 303
kiam, 301
kube2iam, 300
Azure Disk Storage, 80
Azure File Share, 81
Azure, Kubernetes built-in controller for, 132

B

backup and recovery, storage considerations, 81
bare metal versus virtualized machines, 36
base images
choosing an image, 429
golden base image antipattern, 428
base64 encoded secret data, 192
Bearer tokens, 276
BGP (Border Gateway Protocol), 105, 108
MetalLB using BGP peer with network routers, 133
use by Calico, 117
VXLAN and, 118
BGPPeer CRD, 119
Bitnami-labs/sealed-secrets, 211
block devices, 82
blue/green rollout pattern, 445
bootstrap utilities, 48
bootstrapping Kubernetes control plane, 48
Border Gateway Protocol (see BGP)
build container image versus runtime image, 431
build tools for container images, 69
building blocks, application platform on Kubernetes, 17-20
authorization/admission control, 20
container networking, 18
container runtime, 18
developer abstractions, 20
IAAS/datacenter and Kubernetes, 18
identity, 19
observability, 20
secret management, 19
service routing, 19
software supply chain, 20
storage integration, 19
Buildpacks, 433

C

C#, 327
CA (see Cluster Autoscaler)
Calico, 112
absorbing kube-proxy responsibilities, 147
BGP routing protocol, 105
GlobalNetworkPolicy CRD, 369
identity, establishing, 290
network policy enforcement, 110
overview, 117
Pod CIDR, IP range values, 103
routing packets inside the cluster, 118
canary releases, 445

capacity management, 378
centralized policy systems for admission control, 234-241
cert-manager controller, 166-168, 323
certificate management
  Certificate Authority for TLS certificates, 190
  in Istio, 175
  in service meshes, 169, 184
certificates
  authentication based on for robot accounts, 285
  Certificate Signing Requests, 279-281
  client-certificate-data, 278
  provisioned through Kubernetes CSR flow, 281
  required submission by webhooks, 224
  root CA certificates, 429
  workload, required by Istio and Envoy, 290
  x509, issues with, 282
CertificateSigningRequest object, 280
CFS (Completely Fair Scheduler) bandwidth control, 366
characteristic-based node pools, 31
chargeback, 257, 260
CI/CD (continuous integration/continuous delivery), 20
  add-on operators and, 51
  continuous delivery, 439-448
    GitOps, 446
    integrating builds into a pipeline, 440-442
    push-based deployments, 443-445
    rollout patterns, 445-446
  providing self-service onboarding, 451
  using pipeline to install cluster add-ons, 51
CIDR (see Classless Inter-Domain Routing)
Cilium
  absorbing kube-proxy responsibilities, 147
  deploying to cluster, 115
  identity, establishing, 292
  overview, 120
  Pod CIDR, IP range values, 103
CiliumClusterwideNetworkPolicy, 122, 292
CiliumNetworkPolicy, 122, 292
CiliumNode CRD, 121
Citadel, 175
Classless Inter-Domain Routing (CIDR)
  in Cilium CNI plug-in, 121
  in CNI Calico plug-in, 117
  Service CIDR block, 129
  setting up Pod CIDR, 103
Client API consumption of secrets, 195
client libraries, supported programming languages, 327
CLIs (command-line interfaces)
  Cilium CLI, 122
  containerd CLI (ctr), using to inspect containers, 74, 75
  crictl, using to inspect containers, 74
  using crictl with CRI-O, 76
  using Docker CLI to inspect containers, 73
cloud computing
  application/workload and network identity, 289
  cloud provider registries, 434
  difficulties with native routing to workload IPs, 106
  Kubernetes cloud provider controllers, 132
  object stores, 82
  security measures to protect hardware, 188
Cloud Foundry, 8, 462
Cloud Native Buildpacks (CNB), 432, 441
cloud provider integration, 6, 131
Cloud Provider Interface (CPI), 6
CloudFormation for AWS, 44
Cluster API, 33, 355, 451
  Kubernetes operators, 45
  using Cluster Autoscaler with, 391
Cluster Autoscaler (CA), 389-395
  configurable scaling behavior, 391
  Deployment manifest for AWS, 391
Cluster Discovery Service (CDS), 174
Cluster external traffic policy setting, 142
cluster federation, 32-35
  federated software deployment, 35
  management clusters, 33
  observability, 34
Cluster Proportional Autoscaler (CPA), 152, 388
cluster tiers, 29
ClusterConfiguration, 199
ClusterFirstWithHostNet DNS policy, 164
ClusterIP Service, 128, 130
  implementation details, 139
  kube-proxy in IPVS mode, 146
  Service definition exposing NGINX on, 129
ClusterIssuer resource, 167

ClusterRole, 286, 358
ClusterRoleBinding, 286, 358
clusters
  autoscaling, 33, 389-395
    cluster overprovisioning, 393
  deploying service mesh to, 181
  host/node network, Pod CIDR and, 103
  large deployments, scalability issues with Endpoints resource, 137
  multicluster service mesh, 184
  multitenant, 355
  provisioning and configuration for teams, 451
  replacement of, 55
  single-tenant, 354
  size of in etcd deployments, 28
  sizing, 39
  triggering mechanisms for builds, scaling, and upgrades, 60
command-line interfaces (see CLIs)
command-line tooling, 454
common vulnerabilities and exposures (CVEs), 435
communication protocols supported by Services, 134
compute infrastructure, 41
compute-optimized nodes, 42
ConfigMaps, 195, 319, 320
  consuming in applications, 400-402
    downsides, 402
    mounting as files in Pod filesystem, 400
    using environment variables, 401
  example defining a scheduling policy, 349
configuration management tools, 46
conmon process, 75
connection tracking (conntrack), 143
ConstraintTemplate CRDs, 236, 237, 240
container images
  building, 426-433
    build versus runtime image, 431
    choosing a base image, 429
    Cloud Native Buildpacks, 432
    golden base image antipattern, 428
    pinning package versions, 430
    runtime user, 430
  OCI image specification, 67-69
container log processing, 244-247
container networking, 18
Container Networking Interface (CNI), 6, 112, 315
  binary and configuration, 113
  CNI chaining, 148
  CNI providers implementing network policy, 110
  identity for clusters, 290-293
  installation, 114
  plug-ins, 116-126
    absorbing kube-proxy responsibilities, 147
    AWS VPC CNI, 123
    Calico, 117
    Cilium, 120
    leveraging plug-in-specific CRDS for network policy, 369
    Multus, 125
    other, 126
  plug-ins for Pod IPAM, 103
  plug-ins offering network policy APIs, 111
Container Runtime Interface (CRI), 5, 6, 69-71, 315
  containerd plug-in, 74
container runtimes, 18, 47, 63-78
  choosing a runtime, 72-78
    containerd, 74
    CRI-O, 75
    Docker, 73
    Kata Containers, 76
    Virtual Kubelet, using to surface alternative runtimes, 77
  configuring and running multiple, 76
  OCI runtime specification, 65-67
container storage (see storage)
Container Storage Interface (CSI), 6, 87-89, 315
  CSI Controller service, 88
  integration of secret store, 208-210
  Node plug-in, 89
containerd, 74
  CRI plug-in, 70, 72
  handling interaction between kubelet and Kata Containers, 76
  use by Docker, 73
  using ctr CLI to inspect containers, 74
containers
  containerized versus on host etcd deployments, 29
  graceful shutdown, 405
  history of, 64

out-of-memory killed (OOMKilled), 365
pre-stop container life cycle hook, 404
runtime, 47
contextual information in logs, 416
Contour, 5
Contour Ingress controller, 155
container pre-stop hook, 404
proxying of TLS encrypted TCP connections, 160
control groups (cgroups), 64
control plane, 356
components of, 49
Ingress controllers, 156
Istio interactions with, 175
service mesh upgrades, 182
control plane nodes, 42
in-place upgrades, 60
replacing, 58
controller manager, 3
controller runtime for mutating webhook, 231-234
controllers
Ingress (see Ingress; Ingress controllers)
LoadBalancer Service, 131
CoreDNS servers
autoscaling the deployment, 152
DNS-based service discovery, 148
effects of DNS cache on each node, 151
cost management, 378
CPU consumption
using as metric for workload autoscaling, 380
when not to use as workload autoscaling metric, 383
CPU requests and limits, 363, 365
Linux kernel bug affecting CPU limits, 368
CRI (see Container Runtime Interface)
CRI-O, 75
crictl command-line tool, 74
using to inspect containers, 75
using with CRI-O, 76
cross-platform identity with SPIFFE and SPIRE, 305-311
CrossSubnet IP-in-IP mode, 118
CSI Controller, 88
CSIDriver object, 91
CSINode objects, 90
CSRs (Certificate Signing Requests), 279-281
custom automation, 23
building to install and manage Kubernetes, 27
custom resource definitions (CRDs), 318-322
Calico's BGPPeer, 119
CNI plug-in-spacific, leveraging for network policy, 369
IPPools in Calico CNI plug-in, 117
for snapshots, 97
Cyberark, 203

D

data model design for operators, 329-331
data plane
Ingress controllers, 156
service mesh architecture, 179
service mesh upgrades, 183
database operators, 324
Dead Man's Snitch, 256
declarative model, secrets in, 210-215
multicluster deployment of sealed secrets, 215
renewing sealed secret keys, 214
sealed secrets controller, 211-214
dependencies
external dependencies on webhooks, 228
software, federation strategies and, 32
deployment models, 23-61
add-ons, 50-52
architecture and topology, 28-35
cluster federation, 32-35
cluster tiers, 29
etcd deployments, 28
node pools, 31
automation in, 26-27
custom automation, 27
prebuilt installers, 26
containerized components, 49
infrastructure, 35-46
bare metal versus virtualized machines, 36
cluster sizing, 39
compute infrastructure, 41
networking, 42
machine installations, 46-49
managed services versus roll your own, 24-26
deciding between, 25
managed Kubernetes services, 24
roll your own, 24

triggering mechanisms, 60
upgrades, 52-60
cluster replacement, 55
in-place, 59
node replacement, 57
strategies for, 55
Deployment resource, 381
deployments
deploying applications to Kubenetes, 398-399
common questions about, 398
packaging applications for Kubernetes, 399
templating deployment manifests, 398
push-based, 443-445
separating from releases, 446
descheduler, 391
Destination NAT (DNAT), 139
DestinationRule resource, 178
developer abstractions, 20
development clusters, 30
Dikastes, 290
disk encryption, 189
disk storage, 92
distributed system race, 145
distributed tracing, 269-272
instrumenting services for, 420-423
creating spans, 421
initializing the tracer, 420
propagating context, 422
OpenTracing and OpenTelemetry, 269
tracing components, 270
DNS (Domain Name System)
and role in Ingress, 165-166
Kubernetes and DNS integration, 166
wildcard DNS record, 165
auto-scaling DNS server deployment, 152
core-dns Pod not starting due to CNI issues, 114
DNS policy of Ingress controller, 164
service discovery over, 148
service performance, 151
DNS cache on each node, 151
Docker, 64
alternatives to, 5
Open Container Initiative, 65
using as container runtime, 72, 73
Dockerfiles, 426
dockershim, 73
Downward API, 402
dry runs, 224, 234
dual-stack, 109
dynamic provisioning, 81
dynamic routing, 104

E

eBPF (extended Berkeley Packet Filter), 147
eBPF data plane option in Calico, 120
maps, 120
use by Cilium, 120
edit role, 359
elastic network interface (ENI), 123
EmptyDir, 86
encapsulation and tunneling, 106-107
in Calico CNI plug-in, 118
in Cilium CNI plug-in, 122
encryption
application, 190
data within etcd, 197
disk, 189
encryption at rest, 189
envelope, 201
public and private keys, 211
static key, 198
workload traffic, 109
EncryptionConfiguration, 198
endpoints, 135-138
Cilium calculating identity for, 292
Endpoints controller, 136, 145
Endpoints Discovery Service (EDS) in Envoy, 174
Endpoints resource, 135
EndpointSlices resource, 137
envelope encryption, 201
environment variables
consuming ConfigMaps and Secrets via, 401
for access to secrets, preferring volumes over, 216
providing workload metadata via, 402
secret data injected into, 193
using for service discovery, 150
Envoy proxy, 173-175, 290
dynamic configuration via xDS APIs, 174
SDS API for publishing certificates, 309
use with Istio service mesh, 175-177
ephemeral data, storage of, 83
etcd, 49
deployment models, 28

containerized versus on host, 29
dedicated versus colocated, 28
network considerations, 28
etcd machines, 41
Events storage in, 249
in-place upgrades of nodes, 60
node replacements in dedicated cluster, 58
Secrets stored in, 196-197
events
Kubernetes Events, 249
namespace Events retrieved directly, 250
using to debug storage interaction with CSI, 96
Exec probing mechanism, 409
exporters, 254, 417
eXpress Data Path (XDP), 120
external traffic policy, 165
setting on NodePort and LoadBalancer, 141
external-dns controller, 166
ExternalName Service, 133

F

failures
failure modes for admission webhooks, 227
increasing application tolerance for, 407
fault tolerance for applications, 407
federated software deployment, 35
federation
cluster, 32-35
of metrics system, 254
of Prometheus instances, 267
file storage, 82
expanding the filesystem, 96
finalizers, 346
flannel, 112, 126
Flatcar Container Linux, 47
Fluent Bit, 246, 374
Fluentd, 246, 374
four golden signals, 419

G

Galley, 175
Gatekeeper, 235-240
ConstraintTemplate, 237
official documentation, 236
strengths and limitations, 240
Geneve tunneling protocol, 106
support by Cilium, 122
GitOps, 210, 446
GlobalNetworkPolicy, 120, 290
Go language, 220, 228, 327
golden base images antipattern, 428
Google Cloud, Kubernetes built-in controller for, 132
Grafana, 267

H

hard multitenancy, 355
hardware nodes, specialized, 42
Hashicorp, Vault, 203
Haskell, 327
Headless Service, 133
Helm, 399, 456
Heroku, 7, 462
Hierarchical Namespace Controller, 360
Horizontal Pod Autoscaler (HPA), 380-384
avoiding faulty metrics for autoscaling, 383
HorizontalPodAutoscaler resource, 381
not all workloads scale horizontally, 383
replica count increase as load increases, 390
horizontal scaling, 379
Host header, routing based on, 153, 165
host network, binding Ingress controller to, 164
hostPath, 86
HTTP Authorization header, Bearer token in, 276
HTTP Basic Authorization header, 276
HTTP headers, injecting context into, 422
HTTP proxying, 157
with TLS, 158
HTTP requests, Ingress routing of, 153
HTTP, use as probing mechanism, 409
liveness probe, 409
readiness probe, 410
startup probe, 411
HTTPProxy custom resources, 155, 160
HTTPRouteGroup CRD, 171
HTTPS handler for mutating webhook, 229-231
hubble, 123
hypervisors, 196, 363
compute running atop in the cloud, 37
Kata Containers support for, 77
multitenancy security and, 39

I

IAAS/datacenter and Kubernetes, 18
IAM (see Identity and Access Management)

identity, 19, 273-311
  application/workload, 288-311
    network identity, 289-293
    platform mediated Node identity, 299-311
    Projected Service Account tokens, 297-299
    Service Account tokens, 293-296
    shared secrets, 289
  establishing for authentication, 273
  identity service providing IAM permissions, 90
  service mesh features, 169
  tenants, binding ClusterRoles, 358
  user, 274-288
    authentication methods, 275-285
    least privilege permissions for users, 285-288
Identity and Access Management (IAM), 300
  IAM Roles for Service Accounts (IRSA), 303
  kiam, 301
  kube2iam, 300
image index, 68
image registries, 434-439
  cloud provider registries, 434
  image signing, 438
  quarantine workflow, 437
  vulnerability scanning, 435
image tagging and metadata, 443
ImagePullSecrets, 220
ImageService, 70
impersonation, 286
in-place upgrades, 59
infrastructure, 35-46
  automation strategies for, 44-46
    infra management tools, 44
    Kubernetes operators, 45
  bare metal versus virtualized machines, 36
  cluster sizing, 39
  compute, 41
  networking, 42
Ingress, 4, 127, 152-168
  allowed-ingress-pattern annotation, 239
  case for, 153
  DNS and its role in, 165-166
    Kubernetes and DNS integration, 166
    wildcard DNS record, 165
  handling TLS certificates, 166-168
  Ingress API, 154
    configuration collisions, avoiding, 155
  Ingress controller deployment considerations, 162-165
    binding to the host network, 164
    dedicated Ingress nodes, 162
    external traffic policy, 165
    spreading controllers across failure domains, 165
  Ingress controllers, 19, 156
    choosing a controller, 161
  LimitNamespaceIngress, 238
  not overwriting existing Ingress resources, 236
  traffic patterns, 157
    HTTP proxying, 157
    HTTP proxying with TLS, 158
    layer 3/4 proxying, 159
ingress-nginx controller, 5
injection integration, external secret store, 204-208
installers, prebuilt, 26
integration testing
  of platform upgrades, 54
  use of development tiers, 30
interfaces (Kubernetes), 5
IP address management (IPAM), 18, 102-104
  Calico CNI plug-in, 117
  CNI plug-in IPAM, IPv4 and IPv6, 109
  in AWS VPC CNI, 123
  in Cilium CNI plug-in, 121
  Service, 129
IP addresses
  masquerading and, 144
  ready and not ready addresses for Pods, 136
  virtual IP address (VIP), 128, 130
IPPools, 117
  ipipMode set to CrossSubnet, 118
iptables
  rules for NodeProxy and LoadBalancer Services, 141
  rules, ClusterIP Service example, 140
  use by Calico, 120
iptables mode (kube-proxy), 139
  performance concerns, 144
IPv4 and IPv6, 109
IPVS (IP Virtual Server) mode (kube-proxy), 145-147
IRSA (see IAM Roles for Service Accounts)
Issuer resource, 167

Istio, 175-179
network policy enforcement, 290
SPIFFE integration, 310
traffic management in, 177
istiod, 175

J

Jaeger, 420
Jaeger, initializing the tracer using, 420
Java, 327
Java applications, configuring memory limits for, 414
JavaScript, 327
JSON
AdmissionReview objects, 224
use by Open Policy Agent, 235
JSON Web Key Set (JWKS), 311
JSON Web Tokens (JWT), 293, 298, 311
JSONPatch structure, 225
calculating diffs, 234

K

Kata Containers, 76
keys
envelope encryption, 201
static-key encryption, 198
kiam, 301
kube-apiserver, 49
EncryptionConfiguration for all nodes running, 198
kube-controller-manager, 49
--cluster-cidr flag, 103
--node-cidr-mask-size-ipv4 flag, 103
--node-cidr-mask-size-ipv6 flag, 103
kube-dns, 134, 152
kube-prometheus project, 261
kube-proxy, 3, 138
Cilium replacement of, 122
configuring iptables rules for ClusterIP Service, 139
iptables mode, 139
performance concerns, 144
iptables rules for NodePort and LoadBalancer Services, 141
IPVS mode, 145-147
ClusertIP Service, 146
NodePort and LoadBalancer Services, 146
opening and holding port for NodePort Service, 141
problem with rolling updates and Service reconciliation, 145
running without, 147
Kube-Router, 105
kube-scheduler, 49
kube-state-metrics, 268
kube-system Namespace, 241
kube2iam, 300
kubeadm, 29, 48, 451
creating static manifests for control plane components, 49
installation of cluster add-ons, 50
node replacements with, 59
Kubebuilder project, 231, 326
kubeconfig file, 277, 282, 285, 287
kubectl, 454
plug-ins, 454
KubeFed, 35
kubelet, 3, 47
dockershim, 73
handling of secrets, 192
interacting with CNI to attach network, 115
interaction with containerd, 74
interaction with CRI-O using CRI APIs, 75
interaction with Kata Containers through containerd, 76
starting Pods, 70
KubeProxyReplacement setting, 122
Kubernetes
about, 1
core components, 2-3
extended functionality, 4
points of extension, 314-317
summary of, 7
Kubernetes API
aggregation layer, 381
using for service discovery, 150
Kubernetes Metrics Server, 381
Kubernetes Test Grid, 72
kubernetes-admin user details, 277
kubeseal utility, 211
kubeseal --fetch-cert command, 212
sealing a secret, 213
Kustomize, 458

L

labels

in Prometheus, 254
standardized for all API objects, 362
lambda controller, 327
large clusters, scalability issues with Endpoints, 137
larger clusters, benefits of, 40
layer 3/4 proxying, 159
layer 7 proxying, 159
layers (container image), 69
leaking secrets, 216
LimitNamespaceIngress object, 238
Linux
container runtime, 63
distributions used with Kubernetes, 47
kernel bug impacting CPU limits, 368
resource limits, 363
Linux Unified Key System (LUKS), 189
Listener Discovery Service (LDS), 174
Listeners (Envoy), 174
liveness probes, 409
failed, 412
recommended use, 413
startup probes and, 411
load balancers
external, using in front of NodePort Service, 131
multiple, Ingress removing need for, 153
load balancing, 44
by Services, 128
Services load balancing traffic across Pods, 135
LoadBalancer Service, 131
external traffic policy setting, 165
implementation details, 141
Local external traffic policy setting, 142, 165
Local Persistence Volume Static Provisioner, 86
log aggregation, centralizing, problems with, 246
logging, 20, 244-251
alerting on logs, 250
application logs, 415-416
centralized logging platform service, 374
container log processing, 244-247
application forwarding, 245
node agent forwarding, 245
sidecar processing, 245
Kubernetes audit logs, 247-249
Kubernetes Events, 249
security concerns, 251
logic implementation for operators, 331-347
admission webhooks, 345
desired state, 333
existing state, 331
finalizers, 346
implementation details, 335-345
reconciliation of existing and desired state, 334

M

MachineDeployment resource, 391
machines, 35
bare metal versus virtualized, 36
compute infrastructure, 41
installations, 46-49
machine images, 46
using config managment tools, 46
what to install, 47
managed Kubernetes services, using for deployments, 24
management clusters, 33
masquerade, 144
maximum transmission unit (MTU), 107
media types for container image layers, 69
memory consumption
memory limit for a Pod, 403
using as metric for workload autoscaling, 380
memory requests and limits, 364
memory-optimized nodes, 42
Metacontroller, 327
metadata
image, 443
injecting workload metadata, 402
MetalLB, 132
metrics, 20, 243, 251-268
alerts from, 255-257
application, 416-419
components of the system, 260-268
Alertmanager, 266
Grafana, 267
kube-state-metrics, 268
Node exporter, 268
Prometheus adapter, 268
Prometheus Operator, 261
Prometheus servers, 262
CPU or memory consumption, using for workload autoscaling, 380
custom, 253

using in workload autoscaling, 387
faulty, workload autoscaling based on, 383
image and CVE details, 436
long term storage, 253
multiple, workload autoscaling on, 382
organization and federation, 254
Prometheus, 251
pushing, 253
showback and chargeback, 257-260
Metrics Server, 381
collecting resource usage metrics for containers, 381
microservices, 169
autoscaling and, 379
Microsoft Azure (see Azure)
mTLS (mutual TLS), 16, 169, 184
Certificate Authority for, in a service mesh, 184
use in Istio, 177
MultiNamespace resource quotas, 362
multitenancy, 353-376, 452
degrees of isolation for tenants, 354-357
multitenant clusters, 355
single-tenant clusters, 354
Kubernetes capabilities for, 358-375
admission webhooks, 361
multitenant platform services, 374-375
network policies, 368-370
Pod Security Policies, 370-373
resource quotas, 360
resource requests and limits, 363
role-based access control, 358-360
Namespace boundary, 357
Multus CNI plug-in, 125
mutating webhooks, 222, 225
called before validating webhooks, 227
writing, 228-234
controller runtime, 231-234
plain HTTPS handler, 229-231
MutatingWebhook (Vault), 204
MutatingWebhookConfiguration, 225

N

Namespace operator, 324
Namespace-scoped NetworkPolicy, 110
Namespace-scoped secrets, 192
NamespaceLifecycle controller, 221
Namespaces, 64, 452
multitenancy and, 357
resource quotas and, 360
targeting, admission controller webhooks, 227
native routing, 105, 107, 118, 122
network address translation (NAT)
conntrack table and, 143
Destination NAT (DNAT), 139
network file system (NFS), 80
network function virtualizations (NFVs), 125
network policies, 368-370, 452
networking
considerations, 102-112
encapsulation and tunneling, 106
encrypted workload traffic, 109
IP address management, 102-104
IPv4 and IPv6, 109
network policy, 110
summary of key concerns, 112
workload routability, 108
NetworkPolicy API, 4
Calico implementation of, 120
Cilium enforcement of, 122
Cilium implementation of, 292
leveraging to implement deny-all policy, 368
NetworkPolicy objects, 110, 290, 370
configuring Ingress/Egress rules, 110
default deny-all, 369
tenants specifying ingress and egress rules in, 370
use cases, 111
networks
considerations in etcd deployment, 28
container network interface plug-in, 50
Container Networking Interface, 315
(see also Container Networking Interface)
identity, 289-293
making Pod IPs routable to larger networks, 105
networking infrastructure, 42
load balancing, 44
redundancy, 43
routability, 43
showback and chargeback for, 260
NGINX Ingress controller, 161
NGNIX
Endpoints resource for nginx Service, 135
Service definition exposing NGINX on ClusterIP, 129

node agent forwarding of logs, 245
Node exporter, 254, 268
node orchestration, 306
Node plug-in (CSI), 89
node pools, 31
  smaller clusters as alternative to, 41
node proxy, 180
Node Special Interest Group, 72
NodeLocal DNSCache, 151
NodePort Service, 130
  external traffic policy setting, 165
  implementation details, 141
nodes
  CiliumNode object, 121
  CSINode objects, 90
  dedicated to running Ingress controller, 162
  in-place upgrades, 59
  replacement of, 57
Notary, 438

O

object storage, 82
observability, 20, 243-272
  in cluster federation, 34
  components of, 243
  distributed tracing, 269-272
    OpenTracing and OpenTelemetry, 269
    tracing components, 270
  metrics, 251-268
    components of the system, 260-268
    custom metrics, 253
    long term storage, 253
    organization and federation, 254
    Prometheus, 251
    pushing, 253
    showback and chargeback, 257-260
  service mesh features, 169
  services in service mesh, 178
  tooling for new applications, 453
OCI (Open Container Initiative), 65
  image specification, 67-69, 426
  runtime specification, 7, 65-67
onboarding, self-service, 451-453
Open Policy Agent (OPA), 155, 235
  Rego language, 235
Open vSwitch, 126
OpenID Connect (OIDC), 283
OpenShift, 8
OpenTelemetry, 269
OpenTracing, 269, 420
  helper functions inject context into HTTP headers, 422
Operator Framework, 328
operator pattern, 317-322
  custom resources, 318-322
operators (Kubernetes), 45, 453
  developing, 325-347
    data model design, 329-331
    logic implementation, 331-347
    tools for, 325-329
  extensions, 316
  Prometheus Operator, 261
  use cases, 323-325
    application-specific operators, 324
    general-purpose workload operators, 324
    platform utilities, 323
  using for triggering mechanisms, 60
  using to define cluster add-ons, 51
  WebApp Operator, 320
organizational needs, aligning to application platform, 10

P

package versions, pinning, 430
packaging applications for Kubernetes, 399
patch sets, generating, 230
patchOperation, 230
PatchResponseFromRaw, 234
performance
  dedicating nodes to Ingress, 164
  disk read/write, prioritization by etcd machines, 41
  DNS service, concerns with, 151
  etcd deployments and, 28
  high-performance SSD, 94
  impact of node proxy on services, 180
  impacts of multicluster service meshes, 184
  improvements in Services with eBPF, 147
  iptables mode in kube-proxy, concerns with, 144
  IPVS mode in kube-proxy, 145
  tracing impacts of system components, 270
  use of Service to expose Ingress controller, 165
  virtualization and, 37
  webhook admission controllers, 228
permissions and privileges

application permissions, 296
implementing least privilege permissions for users, 285-288
privilege elevation attacks, 286
PersistentClaimVolume objects, resizing, 96
PersistentVolume API, 83
PersistentVolumeClaim API, 83
allowing developers to create PVCs, 94
physical layer, security at, 188
Pilot, 175
ping, 134
PKI (see public key infrastructure)
platform abstractions, 449-464
platform exposure, 450
spectrum of abstractions, 453
abstracting Kubernetes primitives, 458-460
command-line tooling, 454
making Kubernetes invisible, 462-464
templating, 455-458
supporting different levels of abstraction, 461
platform mediated Node identity, 299-311
AWS platform authentication methods/tools, 300-305
IAM Roles for Service Accounts, 303
kiam, 301
kube2iam, 300
cross-platform identity with SPIFFE and SPIRE, 305-311
platform services, building, 313-351
developing operators, 325-347
data model design, 329-331
tools for, 325-329
extending the scheduler, 347-351
custom scheduler, 350
multiple schedulers, 350
predicates and priorities, 348
scheduling policies, 348
scheduling profiles, 350
operator pattern, 317-322
custom resources, 318-322
operator use cases, 323-325
application-specific operators, 324
platform utilities, 323
points of extension in Kubernetes, 314-317
admission control, 316
authentication, 316
operator, 316
plug-ins, 314
webhook, 315
plug-ins
extensions to Kubernetes, 314
interface/plug-in relationship, 5
Pod Security Policies (PSPs), 370-373, 452
PodSecurityPolicy API, 373
PSP controller, 222
PodMetrics, 381
Pods
anti-affinity rules, 165
environment variables enhancing service discovery, 150
Events, 249
evictions, 364
Ingress controllers routing traffic directly to, 157
inspecting using Docker CLI, 74
kubeadm and static Pods, 48
memory requests and limits, 364
networking, 101-126
CNI installation, 114
CNI plug-ins, 116-126
Container Networking Interface, 112
encapsulation and tunneling, 106
encrypted workload traffic, 109
IP address management, 102-104
IPv4 and IPv6, 109
network policy, 110
routing protocols, 104-106
workload routability, 108
Pod resource and VerticalPodAutoscaler, 385
Quality of Service (QoS) class, 363
readiness and readiness probes, 136
resource requests and limits, 413-415
Service backend pool, 128
Service Pod selector, 130
invalid selectors, 135
spreading across failure domains
anti-affinity rules, 407
Topology Spread Constraints, 408
vault-agent injection, 205
Virtual Kubelet running Pods, 77
policy engines, 362, 373
policy systems, centralized, for admission control, 234-241
prebuilt installers, 26
predicates, 348

priorities, 348
privilege elevation attacks, 286
production clusters, 31
programming languages, 327
Projected Service Account tokens (PSAT), 288, 297-299
Prometheus, 34, 251-268
  autoscaling with custom metrics from, 387
  components of metrics stack, 260
    Alertmanager, 266
    Grafana, 267
    kube-state-metrics, 268
    Node exporter, 268
    Prometheus adapter, 268
    Prometheus servers, 262
  critical metrics functions, 252
  custom metrics, 253
  Event exporters, 250
  instrumenting applications for, 417-418
  organization and federation of metrics, 254
  Pushgateway, 253
  tenants deploying multiple instances of, 374
Prometheus Adapter, 387
Prometheus Operator, 261, 323, 374
PrometheusRule resource, 265
proxies
  data plane proxy in service mesh, 173-175
  HTTP proxying by Ingress, 157
  layer 3/4 proxying (TCP/UDP traffic), 159
  node proxy in service mesh, 180
  service mesh, 169
  service mesh communication via, 127
  sidecar proxy in service mesh, 180
proxy technologies, 127
PSPs (see Pod Security Policies)
public key infrastructure (PKI), 277-283
public/private key cryptography, 211-215
Python, 327

Q

Quality of Service (QoS) class (Pods), 363

R

RBAC (role-based access control), 192, 273, 452
  multitenancy and, 358-360
  PSP authorization, 372
  RoleBindings, 276
readiness probes, 137, 410, 412
Recommender (VPA), 384
  recommendations from, 386
reconciliation, 334, 345
RED (Rate, Errors, Duration) method, 416, 419
RedHat OpenShift, 8
redundancy in networking infrastructure, 43
Rego language, 235
rescheduling events, handling, 404-408
  graceful container shutdown, 405
  pre-stop container life cycle hook, 404
resizing PersistentClaimVolume objects, 96
resource overhead for service mesh, 183
resource-intensive webhooks, 228
resources
  custom, 318-322
  overhead for single-tenant clusters, 354
  quotas on, 360, 452
  requests and limits, 363, 413-415
  required to run apps, manifests for, 398
reverse proxies, Ingress controllers paired with, 156
role-based access control (see RBAC)
role-based node pools, 32
RoleBindings, 276, 358, 372
Roles, 372
roll your own deployments, 24
rollout patterns, 445-446
Rook operator, 323
routability, 43
Route Discovery Service (RDS), 174
route reflectors, 119
routing
  in AWS VPC CNI, 123
  in Calico CNI plug-in, 117
  in Cilium CNI plug-in, 120
  limited capabilities of Services, 153
  service (see service routing)
  service mesh features, 169
  workload routability, 108
routing protocols, 104-106
RPCs (remote procedure calls), interface commands issued as, 5
runc, 66
runtime image vulnerability scanning, 437
runtime user of a container, 430
RuntimeClass API, 76
RuntimeService, 70

S

sandbox (Pod), creating, 70

scalability
issues with Endpoints resource in large cluster deployments, 137
issues with iptables mode, 144
scaling
autoscaling, 377
(see also autoscaling)
less tuning for scale with smaller clusters, 40
types of, 378
scheduler, 3
extending, 347-351
custom scheduler, 350
multiple schedulers, 350
predicates and priorities, 348
scheduling policies, 348
scheduling profiles, 350
schemas
CRDs, use of Open API v3 schema, 319
for requests/responses between API server and webhook server, 224
validation on submission of objects, 221
SCTP, support by Kubernetes services, 134
sealed-secret-controller, 211-214
SealedSecret object, 213
misunderstanding of scope, 215
Namespace used during encryption, 215
secret management, 19
secret store providers, making unknown to application, 216
secretbox, 198, 200
SecretProviderClass, 209
secrets, 187
adding credentials in secret mounted into CSI driver, 90
best practices, 215-217
consuming Kubernetes Secrets in applications, 400-404
downsides, 402
mounting as files in Pod filesystem, 400
using environment variables, 401
in the declarative model, 210-215
multicluster deployment of sealed secrets, 215
renewing sealed secret keys, 214
sealed secrets controller, 211-214
sealing secrets, 211
external providers of secret management, 203-210
CSI integration of secret store, 208-210
Cyberark, 203
injection integration, 204-208
Vault, 203
obtaining for applications via external systems, 403
operational concerns, 187
protection of, 188-191
application encryption, 190
disk encryption, 189
transport security, 190
scope of, 192
Secret API, 191-203
consumption models for Secrets, 193-196
envelope encryption, 201
secret data in etcd, 196-197
Secret object, 191
static-key encryption, 198
Service Account Secret, 294
shared, 275, 289
targeting in webhook-enabled Namespace, 227
secrets-store-csi-driver, 208-210
security, 187
(see also secrets)
datacenter, Google's approach to, 188
logs, 251
vulnerability scanning of images, 435
Server Name Indication (SNI) TLS extension, 160
Service Account controller, 220
Service Account tokens (SAT), 288, 293-296
Service Accounts, 195
Calico's network policy enforcement based on, 290
Cilium using to restrict Services access, 292
for robot users, 285
IAM Roles for, 303
using for PSP authorization, 372
Service API, 135
plug-ins implementing, 4
service discovery, 148-152
using DNS, 148
using environment variables, 150
using Kubernetes API, 150
service meshes, 127, 169-184
adopting a service mesh, 181-184
Certificate Authority for mTLS, 184
deploying to new or existing cluster, 181

handling upgrades, 182
multicluster service mesh, 184
prioritizing one of the pillars, 181
resource overhead, 183
data plane architecture, 179
data plane proxy, 173-175
deciding when to use, 169
features provided by, 169
introducing, pros and cons of, 16
in Kubernetes ecosystem, 179
on Kubernetes, 175-179
Service Mesh Interface, 6, 170-173
SPIFFE and SPIRE integration with Istio, 310
tracing and, 272
service routing, 19, 127-185
Ingress, 152-168
case for, 153
choosing an Ingress controller, 161
DNS and its role in, 165-166
handling TLS certificates, 166-168
Ingress API, 154
Ingress controller deployment considerations, 162-165
Ingress controllers, how they work, 156
Ingress traffic patterns, 157-161
Kubernetes Services, 128-152
service meshes, 169-184
ServiceMonitor, 263
services
determining for application platform, 16
multitenant platform services, 356, 374-375
Services, 127, 128-152
Cilium capabilities with, 122
communication protocols supported by, 134
endpoints, 135-138
Endpoints controller, 136
Endpoints resource, 135
EndpointSlices resource, 137
Pod readiness and readiness probes, 137
implementation details, 138-148
ClusterIP, 139
connection tracking (conntrack), 143
kube-proxy, 138
masquerade, 144
performance concerns with iptables mode, 144
IP address management, 129
kube-proxy in IPVS mode, 145-147
ClusterIP, 146
NodePort and LoadBalancer, 146
limitations and downsides, 153
multiple ports and protocols, 134
problem with rolling updates and Service reconciliation, 145
running without kube-proxy, 147
Service abstraction, 128
Service resource, 129
Service types, 130-133
ClusterIP, 130
ExternalName, 133
Headless, 133
LoadBalancer, 131
NodePort, 130
troubleshooting, 134
services, platform (see platform services, building)
showback and chargeback, 257-260
chargeback, 260
network and storage, 260
showback by consumption, 258
showback by requests, 257
side effects, webhooks with, 228
sidecar proxy, 180
sidecars
log processing, 245
Vault injection architecture, 204
SIGTERM signal, application handling of, 406
single-tenant clusters, 354
smaller clusters, benefits of, 40
SMI (see service meshes, Service Mesh Interface)
snapshots of storage volumes, 97
security for, 189
soft multitenancy, 355
software supply chain, 20, 425-448
building container images, 426-433
build versus runtime image, 431
choosing a base image, 429
Cloud Native Buildpacks, 432
golden base image antipattern, 428
pinning package versions, 430
runtime user, 430
continuous delivery, 439-448
GitOps, 446
integrating builds into a pipeline, 440-442
push-based deployments, 443-445

rollout patterns, 445-446
image registries, 434-439
image signing, 438
quarantine workflow, 437
vulnerability scanning, 435
software-defined networks (SDNs), 101
SPIFFE and SPIRE, 305-311
architecture and concepts, 305
direct application access, 308
integration with AWS, 311
integration with secrets store (Vault), 310
integration with service mesh (Istio), 310
integration with sidecar proxy, 309
other application integration methods, 310
Spring Cloud Kubernetes, 195, 402
ssh access to server running etcd, attack on secret data via, 196
staging clusters, 30
startup probes, 411
state
desired state for a system, 333
existing state of the sysem, 331
reconciliation of existing state to desired state, 334, 345
state probes, 408-413
implementing, 412
liveness probes, 409
probing mechanisms common to, 409
readiness probes, 410
startup probes, 411
static provisioning, 81, 85
static routes, 104
static vulnerability scanning of images, 436
static-key encryption, 198
storage, 79-99
considerations, 80-83
access modes, 80
backup and recovery, 81
block devices, file, and object storage, 82
choosing a storage provider, 83
ephemeral data, 83
volume expansion, 81
volume provisioning, 81
CSI (Container Storage Interface), 87-89, 315
databases for tracing, 270
encryption of stored data, 189
implementing storage as a service, 89-99
consuming storage, 94
exposing storage options, 92
installation of the integration/driver, 90
resizing, 96
snapshots, 97
Kubernetes storage primitives, 83-87
persistent volumes and claims, 83
storage classes, 86
long term, for metrics, 253
persistent storage availability in cluster replacement, 56
showback and chargeback for, 260
storage integration, 19
storage providers, 87
StorageClass API, 86
creating storage classes for storage options, 93
subdomain-based routing, 154
SVIDs (SPIFFE Verifiable Identity Documents), 305

T

Task CRD, 440, 441
TCP
DNS cache query upgraded to, 151
proxying by Ingress controllers, 159
support by Kubernetes Services, 134
using as probing mechanism, 409
TCPRoute CRD, 171
Tekton, 440
telnet, 134
templating, abstraction through, 455-458
tenancy models, 451
tenants, 353
(see also multitenancy)
degrees of isolation, 354
multitenant clusters, 355
single-tenant clusters, 354
listing all Namespaces on the cluster, 359
Terraform, 44
testing
cluster tier for, 29
conformance testing of container runtime, 72
early prerelease versions of platform, 27
integration testing of platform upgrades, 54
tiers (cluster), 29
TLS
API server calling webhooks over, 224
encryption over, 190

handling TLS certificates, 166-168
HTTP proxying with, 158
mTLS (mutual TLS), 177
proxying of TLS encrypted TCP connections, 160
TLS passthrough, 161
TokenReview API, 203, 207, 298
tokens
for users and groups, 275
Projected Service Account, 297-299
Service Account, 293-296
tracing, 20, 243
distributed (see distributed tracing)
Traffic Access Control APIs, 171, 177
Traffic Metrics API, 172
Traffic Specs API, 171
Traffic Split API, 171
TrafficMetrics resource, 172
TrafficTarget, 171
transferability of skills, 461
transport security, 190
(see also TLS)
triggering mechanisms for automated installations and management, 51, 60
tunneling protocols, 106

U

UDP
proxying by Ingress controllers, 159
support by Kubernetes Services, 134
Unix socket
CRI-O exposing the CRI over, 75
CSI Controller service exposing APIs over, 88
dockershim, using crictl with, 74
for containerd CRI plug-in, 70
Updater (VPA), 384
updates
Endpoints resources, 137
problem with rolling application updates and Service reconciliation, 145
to iptables rules, 144
(see also iptables)
upgrades, 52-60
benefits of smaller clusters for, 41
handling for service mesh, 182
in-place, 59
integration testing, 54
planning for failures, 53
platform versioning, 52
strategies for, 55
cluster replacement, 55
node replacement, 57
upstream repository, 231
USE (Utilization, Saturation, Errors) method, 416, 419
user identity, 274-288
authentication methods, 275-285
OpenID Connect, 283
public key infrastructure, 277-283
shared secrets, 275
implementing least privilege permissions for users, 285-288
user interface for tracing service, 271
username/password combinations, 276
utilities (platform), 323

V

validating admission controllers, 221
validating webhooks, 222, 224
marking required fields in resources, 362
ordering of calls to, 227
ValidatingWebhookConfiguration, 225
Vault, 203, 403
enterprise secret store, 289
injection integration of secret store, 204-208
SPIFFE and SPIRE integration, 310
vault-provider installation on hosts, 208
Velero, 82
versioning (platform), 52
Vertical Pod Autoscaler (VPA), 384-387
components, 384
configuring vertical autoscaling, 385
recommendations, 386
vertical scaling, 379
view role, 359
virtual IP address (VIP), 128, 130
Virtual Kubelet, 77
virtual machines (VMs), 6
use by Kata Containers, 76
VM-based runtimes, 72
VM-level isolation for workloads, 73
Virtual Private Clouds (VPCs), 56
AWS VPC CNI, 123
virtualization
bare metal versus virtualized machines, 36
trade-offs with, 37
VirtualService resource, 177

VolumeContentSnapshot, 97
volumes
expansion of, 81, 96
loss of data, 98
persistent volumes and claims, 83
preferring over environment variables for access to secrets, 216
provisioning, 81
secrets consumed via, 194
VolumeSnapshot, 97
VPA (see Vertical Pod Autoscaler)
VSphereMachine objects, 232
vulnerability scanning by image registries, 435
VXLAN tunneling protocol, 106
support by Cilium, 122
use by Calico, 118

W

Weave, 126
WebApp CRD, 319-322
Kubernetes resources in, 319
manifest, 320
webhooks, 222, 223-228
admission, 345, 361
configuring webhook admission controllers, 225-227
design considerations for admission controllers, 227
extensions to Kubernetes, 315
sending logs to, 249
writing a mutating webhook, 228-234
wildcard DNS records, 165
worker nodes, 42
in-place upgrades, 60
replacing, 59
workflows, 457
workload orchestration, 307
workload plane, 356
workloads
autoscaling, 380-389
Cluster Proportional Autoscaler, 388
custom autoscaling, 389
Horizontal Pod Autoscaler, 380-384
using custom metrics, 387
Vertical Pod Autoscaler, 384-387
general-purpose workload operators, 324
identity, 288
(see also application/workload identity)
injecting workload metadata, 402
isolation guarantees, 73
isolation requirements, 356
memory limits on, 365
migrating between clusters, 56
routability, 108

X

x509 certificates, 278, 282
XDP (eXpress Data Path), 120
xDS APIs (Envoy), 173

Z

Zipkin, 420

About the Authors

Josh Rosso has been working with organizations to adopt Kubernetes since version 1.2 (2016). During this time he's worked as an engineer and architect at CoreOS (RedHat), Heptio, and now VMware. He's been involved in architecture and engineering to help build compute platforms in financial institutions, establish edge compute to support 5G, and much more. He's worked on environments ranging from enterprise-managed bare metal to cloud-provider managed virtual machines.

Rich Lander was an early adopter of Docker and began running production workloads using containers in 2015. He learned the value of container orchestration the hard way and was running production applications on Kubernetes by version 1.3. Rich took that experience and subsequently worked at CoreOS (RedHat), Heptio, and VMware as a field engineer helping enterprises in manufacturing, retail, and various other industries adopt Kubernetes and cloud native technologies.

Alexander Brand started working with Kubernetes in 2016, when he helped build one of the first open source Kubernetes installers at Apprenda. Since then, Alexander has worked at Heptio and VMware, designing and building Kubernetes-based platforms for organizations across multiple industry verticals, including finance, healthcare, consumer, and more. As a software engineer at heart, Alexander has also contributed to Kubernetes and other open source projects in the cloud native ecosystem.

John Harris has been working with Docker since 2014, consulting with many of the top Fortune 50 companies to help them successfully adopt container technologies and patterns. He brings experience in cloud native architecture, engineering, and DevOps practices to help companies of all sizes build robust Kubernetes platforms and applications. Prior to working at VMware (via Heptio), he was an architect at Docker advising some of its most strategic customers.

Colophon

The animal on the cover of *Production Kubernetes* is a common beaked whale, a name given to a group of 22 whale species having a shared characteristic of a dolphin-like beak. Little is known about most of these species due to their scarcity and tendency to live in deep-ocean habitats off continental shelves.

Cuvier's beaked whale, or goose-beaked whale, is the species most commonly encountered by humans. Like the whale on the cover, male goose-beaked whales are a dark gray with a lighter head; females tend to be orange-brown in color. These beaked whales have a curved dorsal fin more closely resembling that of dophins than fellow whales as well as a stunted beak. Males develop tusks that are used to ward off predators and possibly compete for mates, which may be the reason male Cuvier's beaked whales in particular have distinguishing scars along their sides.

Beaked whales regularly dive to depths over 1,600 feet for more than an hour at a time, using echolocation and their unique suction feeding technique, made possible by specialized pairs of throat grooves, to hunt prey. Scientists have theorized that the relatively larger spleens and livers in beaked whales could be adaptations to assist with oxygen processing when deep diving. These deep foraging dives are typically followed by multiple shallow dives and extended periods of surface breathing.

Although little is known about the conservation status of most species of beaked whales, four are classified by the IUCN as "lower risk, convervation dependant" due to anthropogenic factors such as deep-sea fishing, biocontamination, and sonar. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Quadrupeds*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning

©2019 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. | 175