O'REILLY®

Production Kubernetes

Building Successful Application Platforms

Early Release
Raw & Unedited

Compliments of

VMware Tanzu™

Josh Rosso, Rich Lander,
Alex Brand & John Harris

vmware

VMware Tanzu™

Modernize your applications and infrastructure to deliver better software to production, continuously.

Learn more at tanzu.vmware.com

Production Kubernetes

Building Successful Application Platforms

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

Josh Rosso, Rich Lander, Alex Brand, and John Harris

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

Production Kubernetes
by Josh Rosso, Rich Lander, Alex Brand, and John Harris

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: Deborah Baker
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Cate Dullea

July 2021: First Edition

Revision History for the Early Release
2020-11-05: First Release

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

978-1-492-09230-8

[LSI]

Table of Contents

CHAPTER 1
A Path to Production

A note for Early Release readers

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

This will be Chapter 1 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *prodk8s@gmail.com*.

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. Evident 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 that 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 towards 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

The above 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 capbility, 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 Kuber-

netes 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 below, 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)  (to override, set KUBERNETES_CLIENT_OS
and/or KUBERNETES_CLIENT_ARCH)

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 from https://
dl.k8s.io/v1.18.6
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 which 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. 3 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.

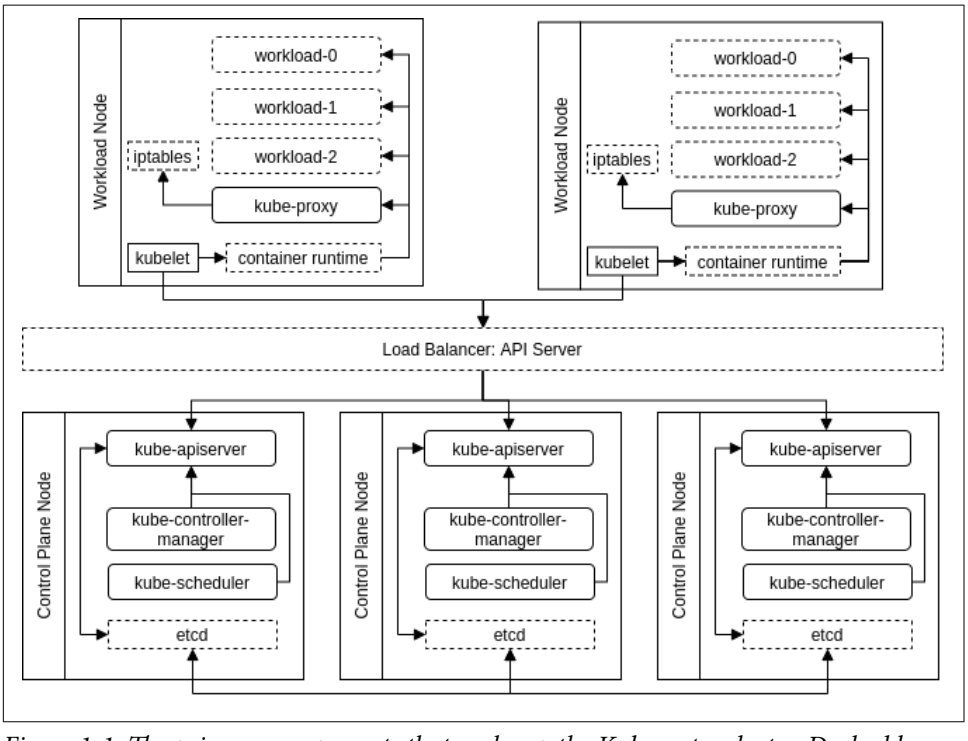

Figure 1-1. The primary components that make up the Kubernetes cluster. Dashed borders represent components that are not part of core Kubernetes.

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.

Beyond Orchestration - Extended Functionality

There are areas where Kubernetes does more than just orchestrate workloads. As mentioned above, 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

plugin to implement the service abstraction. This approach would require users to choose between a variety of plugins 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. 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 plugins that satisfy these interfaces, providing functionality such as networking.

A specific example of this interface/plugin 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.

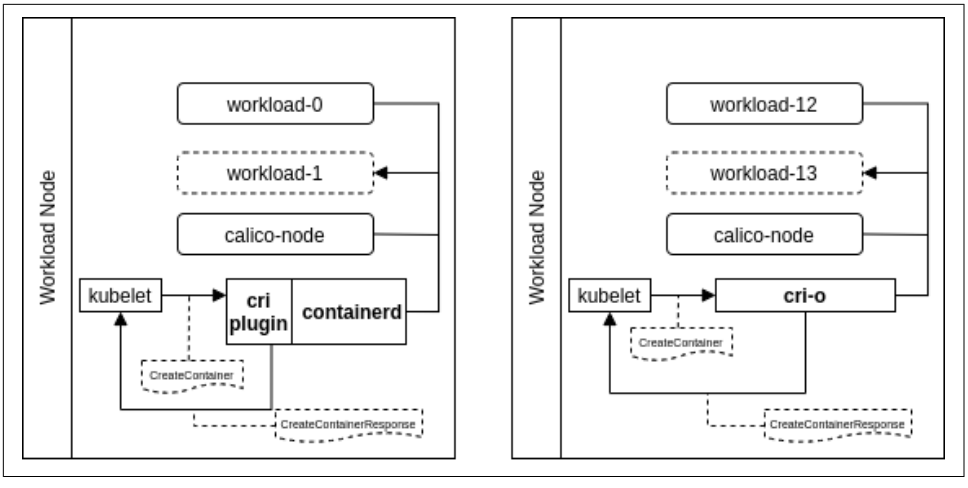

Figure 1-2. Two workloads nodes running two different container runtimes. The Kubelet sends commands defined in the CRI such as `CreateContainer` and expects the runtime to satisfy the request and respond.

In many interfaces, commands, such as `CreateContainerRequest` or `PortForwardRequest`, are issued as remote procedure calls (RPCs). In the case of CRI, communication happens over GRPC and the Kubelet expects responses such as `CreateContainerResponse` and `PortForwardResponse`. In the above, 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 plugin 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 architecture, build, and deploy Kubernetes clusters.

Over time, we've even seen some functionality removed from the core project in favor of this plugin 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 proceedures 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.

CNI: Container Networking Interface
: Enables networking providers to define how they do things from IPAM to actual packet routing.

CSI: Container Storage Interface
: Enables storage providers to satisfy intra-cluster workload requests. Commonly implemented for technologies such as ceph, vSAN, and EBS.

CRI: Container Runtime Interface
: 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.

SMI: Service Mesh Interface
: 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.

CPI: Cloud Provider Interface.
: Enables providers such as VMware, AWS, Azure, and more to write integration points for their cloud services with Kubernetes clusters.

OCI: Open Container Initiative Runtime Spec
: 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 plugins 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 it 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 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 where Kubernetes, by itself, is the perceived answer to problems. When in actuality their problems revolve around delivering applications, 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 we consider 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 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 tenant 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? Do they run their 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 pre-built application platforms is the need to conform to its 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 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 a naieve evaluation of approaches.

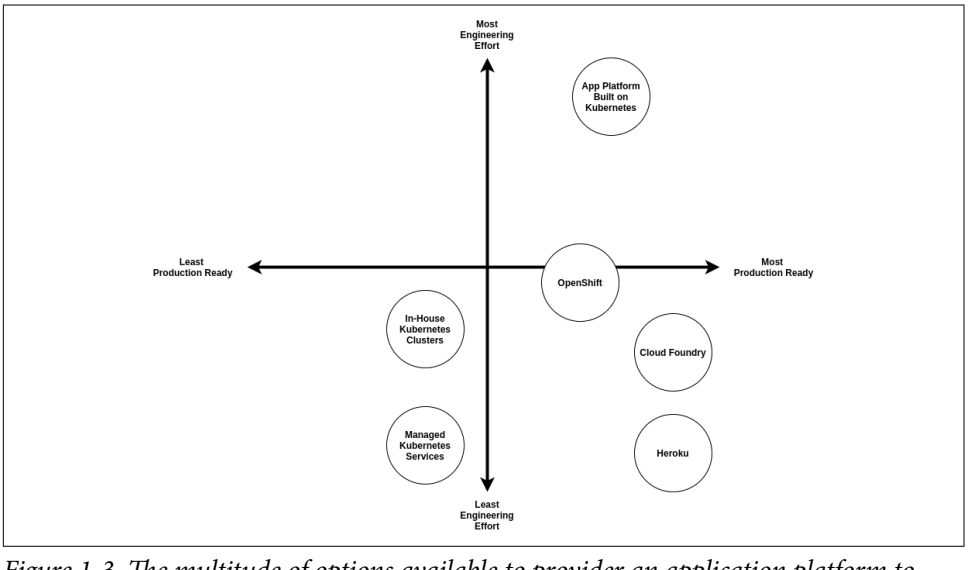

Figure 1-3. The multitude of options available to provider an application platform to developers.

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 as 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.

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 above is a third dimension, a z-axis that demonstrates how aligned the approach is with your requirements. Let's take the same liberties as above, and use Figure 1-4 to map out how this might look when considering alignment with organizational needs.

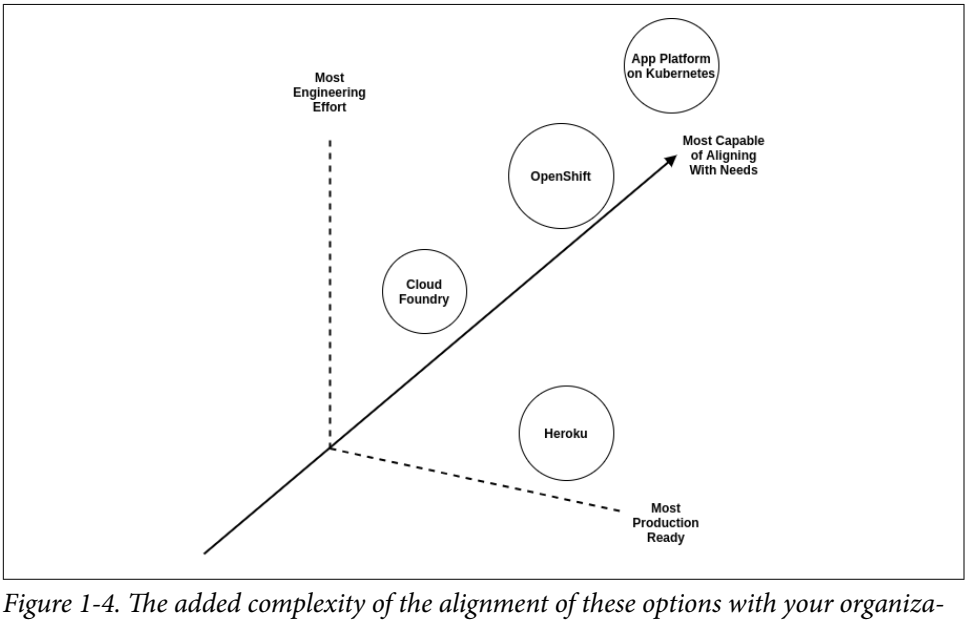

Figure 1-4. The added complexity of the alignment of these options with your organizational needs, the Z axis.

In terms of requirements, features, and behaviors you'd expect out of a platform, building a platform is almost always going to to be the most aligned. Or atleast 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 data center
- 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 as it's 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 non-trivial 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 there needs. Often we do not want to simply provide a "powerful" engine every developer must build their own platform atop of, as jokingly depicted in Figure 1-5.

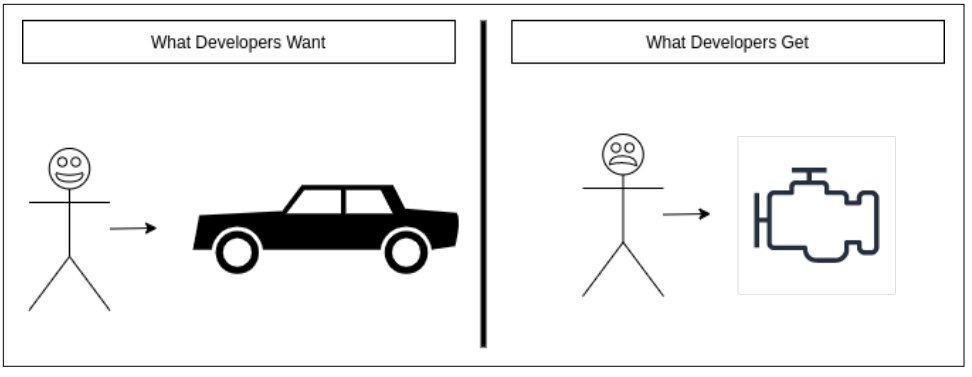

Figure 1-5. When developers desire an end-to-end experience (e.g. a driveable car), do not expect an engine without a frame, wheels, and more to suffice.

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 principal 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, as in Figure 1-6.

Figure 1-6. The foundation of a soon to be built home. Similar to Kubernetes.

A good foundation should be structurally sound, able to 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 seen in Figure 1-7.

Figure 1-7. An apartment that is move-in ready. Similar to platform as a service options like Heroku.

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 non-negotiable opinions within, we can be successful on day 1.

Circling back to Kubernetes, which we have likened to a foundation, we can now look to build that habitable home on top of, as depicted in Figure 1-8.

Figure 1-8. Building a house. Similar to establishing an application platform, which Kubernetes is foundational to.

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 our Deployment Considerations chapter.

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 1-9.

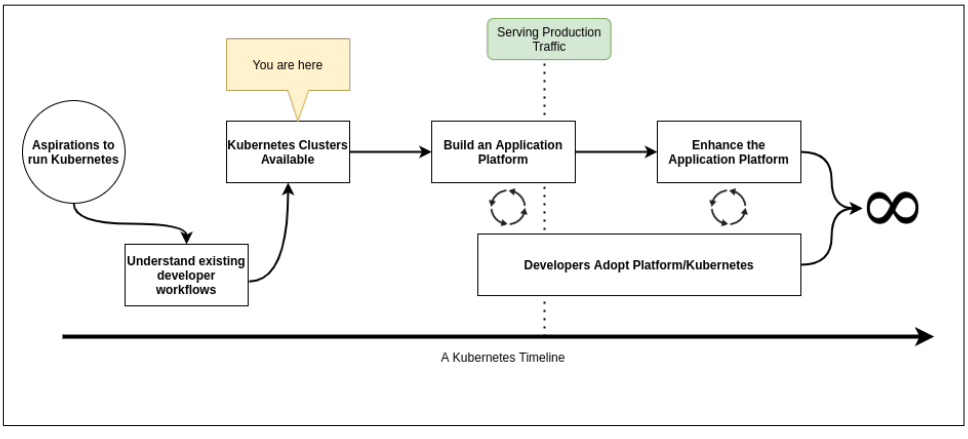

Figure 1-9. A flow our teams may go through in their path to production with Kubernetes.

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 inter-

action 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.

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

Weighing the above, 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 we will have available as you build a platform (see Figure 1-10. This includes everything from the foundational components to optional platform services you may wish to implement.

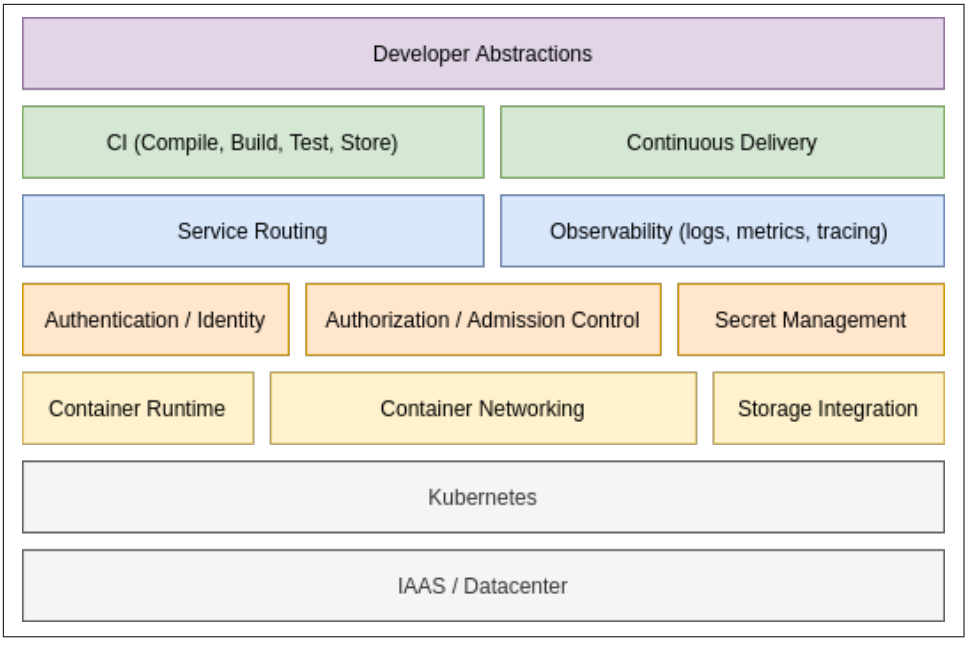

Figure 1-10. Many of the key building blocks involved in establishing an application platform.

The components in figure 1-10 have differing importance to differing audiences. Some components such as container networking and container runtime are required for every cluster, considering 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

This is the foundational layer we have called out many times in this chapter. We don't mean to trivialize this layer as 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 lifecycle 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 Inteface (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 faciliate communication. Common technology choices include Calico or Cilium, which optionality 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 plugins 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 amount of guarentee in that 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 plugin 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. 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 mutual-tls, 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 managment 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 mutual TLS 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 that 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 lifecycle of getting software in source code to runtime. This involves the common concerns around continuous integration (CI) and continuouss 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 saftey 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 your clusters. This includes at the system and application layers. Typically, we think of observability to cover 3 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 scape, 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 static. As trace data is col-

lected, 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 signifigant 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 better understanding 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

A note for Early Release readers

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

This will be Chapter 2 of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *prodk8s@gmail.com*.

The first step to using Kubernetes in production is obvious: make Kubernetes exist. This includes installing systems to provision Kubernetes clusters, and also 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 hav-

ing 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 installation as well as the infrastructure dependencies and break down the various cluster components and demystify how they fit together. We'll also look at ways to manage the addons you install to the base Kubernetes cluster as well as strategies for upgrading Kubernetes and the addon 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 lifecycle 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 AWS and already use Fargate for serverless compute, 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 you 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 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, developers will care only about the features of the platform, not the underlying infra 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 vs 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 towards 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 towards rolling your own Kubernetes if:

- the vendor-imposed restrictions make you uneasy
- if you have little or no faith in the corporate behemoths that provide your compute infrastructure
- if you are excited by the power of the platform you can build around Kubernetes
- if 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. The Addons and Triggering Mechanisms sections 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 up-front engineering investment to automate deployments using software systems will pay dividends in saved toil and troubleshooting in future.

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

Pre-Built Installer

There are now countless open-source and enterprise-supported Kubernetes installers available. 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 pre-built 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 pre-built 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 home grown automation route is likely your choice only if:

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

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

- You fit the use-case above 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. 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 the containerized components that constitute the control plane of a Kubernetes cluster.

Architecture and Topology

This section covers the architectural decisions you should have settled before you begin work on implementing the automated systems 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 give 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 vs co-located 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 Co-Located

A very common question we get about how to deploy is whether to give etcd its own dedicated machines or to co-locate 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 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 co-located etcd. Kubeadm is a popular Kubernetes bootstrapping tool that you will likely be using and 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 co-located 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 many meanings and implications. I 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 I will describe what each of these four tiers commonly mean.

Testing
: These 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. They may also used by developers when they a local cluster is inadequate for local development, or as a subsequent step to testing on a local cluster. This is more common when a 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
: These are generally "permanent" clusters without a TTL. They are multi-tenant (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 and 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 since 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 development, these 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 that need access to pre-release versions of software. They will often have a similar SLO to development clusters. They 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
: These 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 revolves around the choice between using multiple node pools within a single cluster vs 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. The subtle advantage of using distinct clusters vs node pools is that you are going to need to solve automated cluster provisioning and multi-cluster management, and you will have to tackle federated software deployments at some point in the future. So consider your options and federation plans carefully.

A characteristic-based node pools is one that consist 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 the infrastructure section of this chapter. 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.

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 multi-level 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 discuss two strategies for keeping a central registry for your cluster inventory. 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 increase, they need to leverage software systems to accomplish this. And, as you would expect, they use Kubernetes-based platforms to run this software. There is usually nothing particularly unique about these management clusters besides the fact that their workloads are used to manage other clusters and the software that is deployed there, rather than end user websites or applications. In fact, the more commonality there is between your management and workload clusters, the lower the operational overhead you will be.

Common regional services run in management clusters are:

Container Registries
: These will serve images and possibly Helm charts to the rest of your infrastructure, either regionally or globally.

CI/CD Systems
: When these are run on Kubernetes, the management cluster is a logical place to deploy these.

Cluster Provisioners and Registries
: This is a central register of all the clusters under management, usually including the underlying infrastructure.

  Federated Observability
  : This gives cluster operators the ability to look at high-level metrics and logs in one location, rather than logging into individually logging into cluster-specific dashboards. Having a federated view is very useful for getting a global view of affairs. Even if it becomes necessary to go to a cluster's local metrics dashboard to get details, having a federated view that gives you enough information to *know* you need to investigate further can be vital.

Federated Software Deployment
: Some end user applications need to be deployed to multiple clusters. This should be defined and managed in a single source of truth.

Cluster Registration

Let's talk about how you get your clusters registered centrally and brought under federated management.

One strategy involves a top down approach. With this method you will have a single gateway to producing clusters and implies that cluster provisioning systems are a part of your cluster federation strategy. The cluster's creation and registration with your system will be the same action. Quite a few large enterprises that were early adopters have used a variation of this approach to build themselves an in-house Kubernetes as a Service (KaaS) platform.

This type of system lends itself to good controls and governance that are strict requirements in many enterprises. The drawback is that the versions and features of the resulting Kubernetes clusters are limited to what is available through the KaaS platform.

And on the topic of cluster registration, the KaaS platform will need to poll for successful cluster creation to initiate cluster addon installation. That is to say that the KaaS platform will provision infrastructure that, ideally, will start machines with images that have the required software installed to start the necessary services and initialize the cluster. The cluster's control plane will generally become available some minutes after infra provisioning is first initiated. So the KaaS system will need to poll the new cluster's Kubernetes API endpoint until it comes online. Inherent to this pattern is implementing timeout systems that will decide when cluster initialization is considered failed and what the subsequent alerting or remedial actions are to be. In Figure 2-1 we see the flow of operations for provisioning a new cluster with a top-down approach.

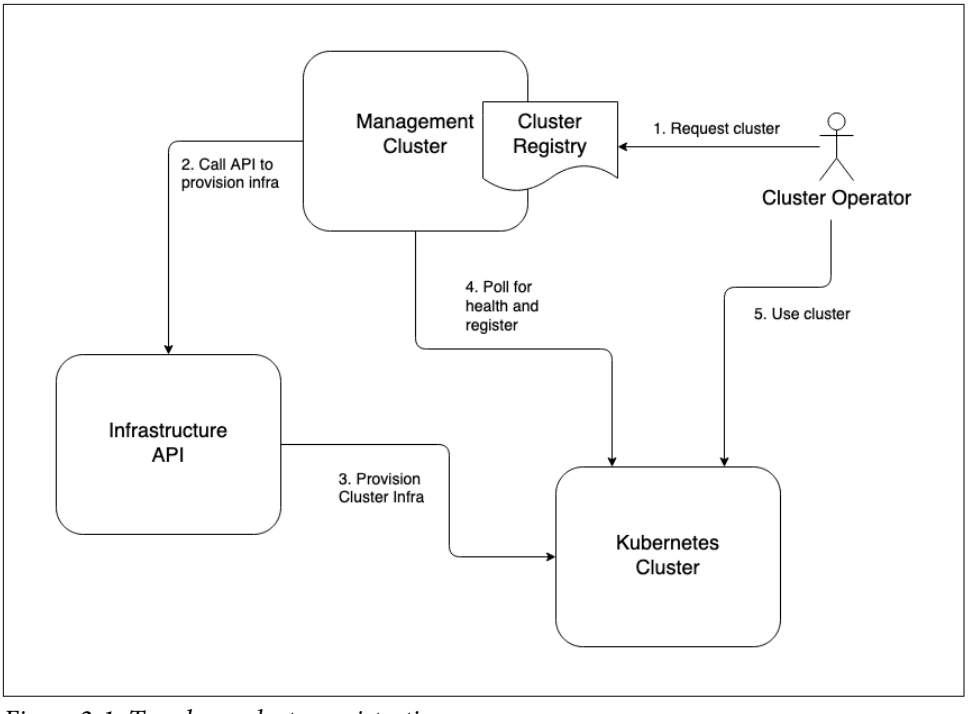

Figure 2-1. Top-down cluster registration.

The alternative is a bottom-up pattern whereby your cluster registers itself into the system when it comes online. This could be achieved, for example, with a simple static pod manifest that spins up a pod that does one thing: calls your federated cluster management system and registers the new cluster into that system. This method has the advantage of allowing teams to bring their own Kubernetes if they have specialized requirements not met by the corporate KaaS. It also alleviates the KaaS system from verifying the result of its cluster creation operation.

The downside is that you lose control over what you introduce into your cluster management system and the compatibility of said clusters. You can mitigate this by publishing the requirements and putting the onus on teams to meet them if they wish to introduce their own clusters into the system. The other complication is authenticating and authorizing registration of clusters into your system. If you have elegant methods to address this issue through a centralized identity provider such as Active Directory this may be somewhat trivial to address. Figure 2-2 illustrates the flow of operations when using this bottom-up cluster registration approach.

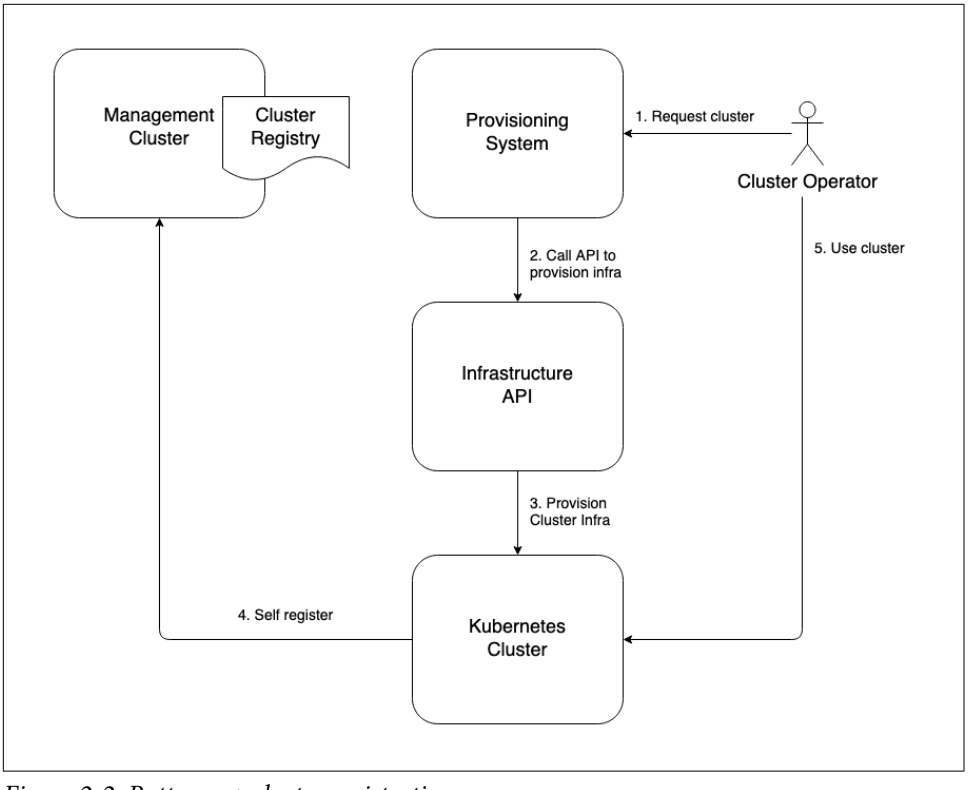

Figure 2-2. Bottom-up cluster registration.

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.

Federated Software Deployment

One more 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 make a federated version of Namespace and Deployment types and declare the spec that is to 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 because, up until now, still the most common way we see in the field is to manage this concern is with CI/CD tools that are configured to target different clusters when workloads are deployed.

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 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. Terraform from Hashicorp is a

popular tool to achieve this declarative automation and it is most successfully used when addressing the specific requirements of a particular organization's environment. It becomes unwieldy when leveraged as a part of a generalized tool for everyone to use. If you find yourself using a tool such as Terraform as a multi-cloud installer, you are likely going to find yourself trying to use its configuration language as a programming language. In that situation, you will find an actual general-purpose programming language such as Go or Python to be a better fit than tools like Terraform for automating infrastructure provisioning and management. Don't discount using cloud provider's client libraries and writing software that calls the cloud provider's API.

This automation model requires the ability to provision servers, networking and related resources on demand, as with a cloud provider like Amazon Web Services, Microsoft Azure or Google Cloud Platform, to give the common examples. 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 lifecycle 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 where will examine those models and patterns that we have seen success with.

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 2 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.

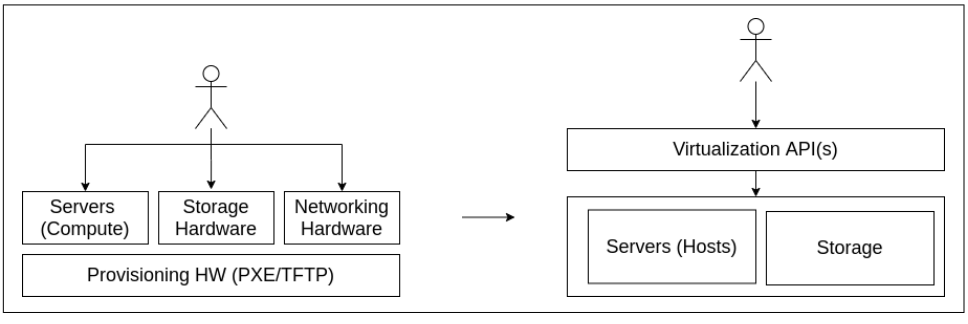

Figure 2-3. Comparison of administrator interactions when provisioning and managing bare metal compute infrastructure vs virtual machines.

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 setting without disrupting the server
- Programmatically control what hardware servers have access to, e.g. NICs
- Running 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, with this mind 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 they will be scheduled appropriately.

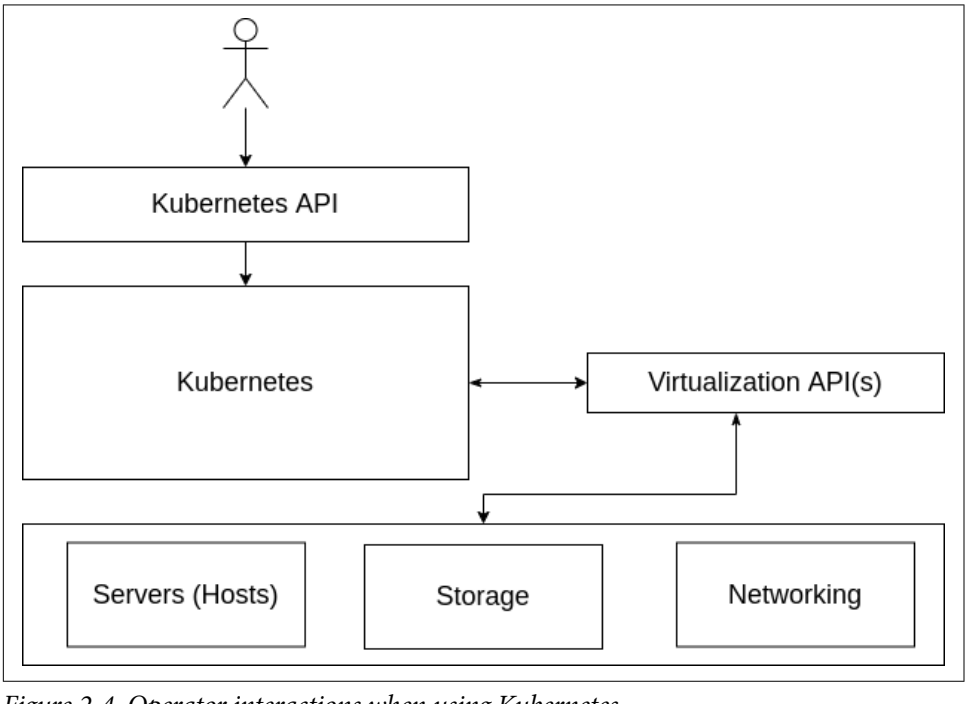

Figure 2-4. Operator interactions when using Kubernetes.

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 autoscaling 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 becoming available through major cloud providers and bare metal-exclusive IaaS services like Packet are gaining market share. Success with bare metal is not without its challenges, including:

Significantly larger nodes
: Larger nodes cause for (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 machines 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.

Multi-tenant 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 aged 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 amount of trade-offs to continue unpacking, but in essence our guidance when choosing between cloud providers (virtualization), on-prem virtualization, and bare metal are to consider what option makes the most sense based on your needs and 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 your 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 multi-tenant 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 resources 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 100+ workers, we often encounter issues of scale that need to be solved for. These issues vary case to case, but bottle-necks 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 in the Upgrades section later in this chapter, but it is an attractive 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 in the section 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 graphics processing units (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 Nodes (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 co-locating etcd on these control plane machines, the same considerations for etcd nodes 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 a 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 1vCPU: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 node such as the R5 instance types on AWS. 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 1vCPU: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 required (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 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 data center 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 the Networking Considerations under Etcd Deployment Models earlier in this chapter. 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 three categories, the first being the tools that exist today that you can leverage. Then, we'll get into how custom software development can play into this. And, lastly, 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 processes. 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 - or impossible if they cater to a single cloud provider. Data formats like json and even configuration languages are not as good at conditional statements and looping functions as general purpose programming languages. In fact, they can be downright terrible to work with when implementing more involved logic.

Kubernetes clusters require involved infrastructure provisioning. Furthermore, clusters will vary according to their use. A production cluster in one public cloud environment will look very different from, say, a development cluster on a different cloud provider, let alone an edge deployment at a warehouse or a store. Additionally, your cluster management and upgrade strategies may not allow for strict immutable infrastructure implementations whereby you replace entire clusters wholesale. In these cases, inspection and conditional logic will be required for automation that is beyond the practical capabilities of such tools.

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 developing software to solve these problems as discussed in the next sections. If you don't have software developers on your team and you are building custom automation to deploy Kubernetes clusters, strongly consider adding some. If this is not an option, we would recommend reevaluating the choice to build custom automation from the ground up. Perhaps there is an open source or commercially supported Kubernetes installer that will meet your needs.

Custom Software

In this section we'll cover the notion of using custom software generally but *exclude* custom Kubernetes operators which we will discuss in more detail in the next section. Custom Kubernetes operators are certainly custom software but are a category that warrants its own section in this context. The kind of software we're addressing here would include custom command line tools or web applications that integrate with infrastructure provider APIs.

Developing custom software is not traditionally engaged in by operations teams. Scripting repetitive tasks is one thing. But developing tested, versioned, documented, feature-complete software is another. With the client libraries offered by modern infrastructure cloud providers, the barrier to developing custom software is signifi-

cantly lowered. Building command line tools and web services with modern programming languages is quite trivial for experienced software engineers and the toil that can be alleviated by such systems is compelling.

Your custom infrastructure management software can implement sophisticated conditional logic, call out to other systems to collect information, use powerful custom libraries to parse, process and persistently store information. There is barely a comparison in feature capabilities between custom-built software compared with infrastructure management tools.

However, while advances in modern software engineering have made quality software development faster and more readily available, it is far from free. Experienced software engineers that can build reliable software like this are in high demand. And production-ready software entails overhead such as writing unit tests, managing automated build and deployment systems, feature planning and coordination. The larger the project, the more engineers involved, the greater the proportional overhead. But if these disciplines - and the overhead incurred - are not followed, software quality will suffer. And the last thing you want is instability or bugginess in the software that manages the infrastructure that supports your business' software.

It is uncommon to see enterprises developing custom software in-house to manage infrastructure on a wide scale. In part, this is due to the skill set of engineers that traditionally occupy operations teams. They often haven't had a background in, or an inclination to, software development. But that is beginning to change. Software engineering is becoming more and more important to operations and platform management teams.

Another reason this kind of independent custom software is not commonly employed to manage infrastructure for Kubernetes clusters is the emergence of using Kubernetes operators to manage Kubernetes infrastructure. Again, custom Kubernetes operators *do* constitute custom software development, but are a category that we will cover specifically in the next section.

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 for fulfillment of that 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. This is similar to the custom software option discussed in the previous section. However, instead of being an independent service or tool, this is a tight

Kubernetes integration. It involves designing and developing a custom resource definition that extends Kubernetes, and developing a Kubernetes controller that will essentially become a part of the control plane for your cluster. The topic of Kubernetes operators is covered in more depth in Chapter 13.

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. 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? It generally follows a management cluster model whereby you define your Cluster and Machine resources and create them in your management cluster. Cluster API and related cloud provider controllers then fulfill that desired state by calling the API for the cloud infrastructure provider to provision the needed infrastructure and bootstrap the new workload cluster. This incurs the overhead of a cluster dedicated to simply managing other clusters. It seems to be a convenient separation of concerns that is gaining traction.

This management cluster model does have flaws, however. It is usually prudent to strictly separate concerns between your production tier and other tiers. Often organizations will therefore 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 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 creates a dependency external to any given workload cluster that invokes more complications. What if the management cluster becomes unavailable at a busy time that your cluster needs to scale out to meet demand?

Another awkwardness introduced with management clusters and their remote Cluster resources, is that some workloads in your cluster - especially platform utilities - will want access to the attributes and metadata that are contained in that remote Cluster resource. Perhaps a platform utility needs to know if it is running in a production cluster to determine how to fulfill some request. Having that data live in a remote management cluster introduces the problem of having to duplicate and expose that information within the cluster it represents.

If the overhead of dedicated management clusters is not palatable, another option is to bootstrap a single-node Kubernetes cluster using a standalone CLI or similar tool. Then, instantiate the resources that represent its infrastructure - similar Cluster and Machine resources - in that cluster and run the controllers there. In this model, the

cluster manages its own infrastructure instead of a management cluster managing it remotely and potentially from afar.

This pattern also has the distinct advantage that if any controllers or workloads in the cluster have a need for metadata or characteristics in the local cluster, they can access them (with the appropriate permissions) by reading the resource through the 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 will already be contained in the Cluster resource that represents the cluster in which it lives. Having that Cluster resource available locally alleviates the need to separately make that configuration information available somehow.

Another advantage is if some local workload needs to exert control over these resources, having them reside locally is very useful. For example, cluster autoscaling relies on local information such as whether there are Pending pods due to an exhaustion of worker node compute resources. That information is gleaned locally. And being able to remedy that shortfall by scaling the `replicas` on a local MachineSet resource could be extremely useful. It is a far more straight-forward access control model compared with having to scale or manage that resource on a remote management cluster. Lastly, external dependencies are reduced when the reliance on a management cluster to manage infrastructure is removed.

Conclusion

If your organization's operations team has specialized requirements, deep Kubernetes expertise and software engineering experience, extending the Kubernetes API and developing custom infrastructure management software is likely to be a favorable route. If you're short on software engineering but have deep experience with infrastructure management tools, you can definitely be successful with this option, but your solutions will have less flexibility and more workarounds due to the limitations of using configuration languages rather than full-feature 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 what a machine must look like 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 require 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 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 pre-built 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. Any time 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 pre-built machine images.

What to Install

So what do you need to install on the machine?

1. 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.
2. 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.
3. 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.
4. So far we have a Linux OS, a container runtime to run containers, 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.
5. 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.

6. Lastly, you need a command line program that I will call a *bootstrap utility*. The Cluster API project uses Kubernetes custom resources and controllers for this. But a CLI program installed on the host also works well. The primary function of this utility is to call kubeadm and manage runtime configurations. It should be well tested and documented, log intelligibly and handle errors as elegantly as possible, but it isn't critical what language you use.

The Go programming language is a great choice for applications such as this but use a programming language that is familiar to your team. The important part is that it is run when the machine boots and is given arguments that 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. 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.

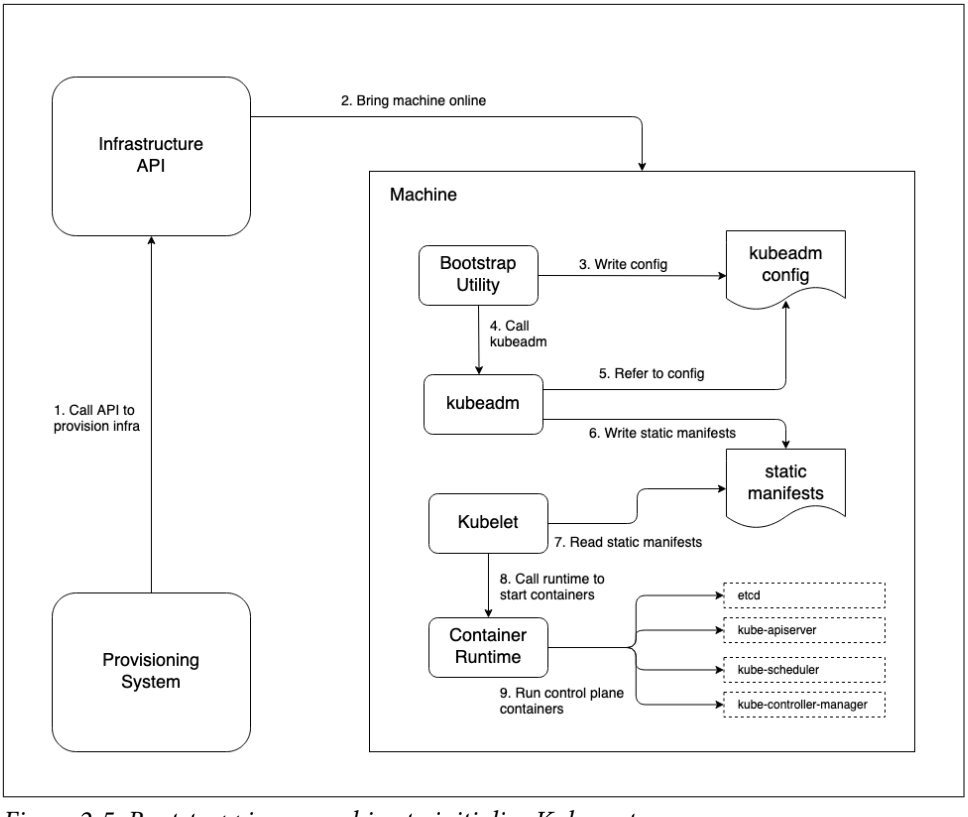

Figure 2-5. Bootstrapping a machine to initialize Kubernetes.

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 only be managed 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 only be valid 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 addons installed onto Kubernetes. These addons 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.

Addons

Cluster addons broadly cover those additions of cluster services layered onto a Kubernetes cluster. We will not cover *what* to install as a cluster addon 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 addons that you add to a cluster should be considered as a part of the deployment model. Addon installation will usually constitute the final phase of a cluster deployment. These addons should be managed and versioned in combination with the Kubernetes cluster itself. It is useful to consider Kubernetes and the addons that comprise the platform as a package that is tested and released as such since there will inevitably be version and configuration dependencies between certain platform components.

Kubeadm installs "required" addons 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 plugin. 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.

In an earlier section under Architecture and Topology, we discussed cluster federation and the registration of new clusters into that system. That is usually a precursor to addon installation since the systems and definitions for installation often live in a management cluster.

Whatever the architecture used, once registration is achieved, the installation of cluster addons can be triggered. This installation process will be a series 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 addons 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 roll-outs of software once new changes have been merged into a branch of version-controlled source code. Triggering installations of cluster addon 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 addon components for the cluster and their versions. It also involves writing the controller logic that can is able to execute the proper installation of the addon 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 addons are for a cluster. But more importantly, it offers the opportunity to programmatically manage the ongoing lifecycle of these addons. 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 addon 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 addon 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 addons.

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 addons should only be undertaken 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 addons. Now we will turn to the "day 2" concern of upgrades.

Upgrades

Cluster lifecycle 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, ongoing lifecycle needs to be solved for 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 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 and upgrade strategy will likely sit 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 machines' operating system version and all packages installed on the machine, such as the container runtime. It obviously includes the version of Kubernetes in use. And it should also include the version of each addon 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 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 applications state and Kubernetes resources without regard to order and dependencies. Brainstorm all the possible failure modes, 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 failovers 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, how you manage these versions is another. In systems as complex as a 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 plugins 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 plugins, 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 lifecycle 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 operators.

Cluster Replacement

This 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 re-routed 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 data center 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. Often times 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 have it re-deploy 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, 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 back-up 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 datastore, 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 (see Figure 2-6).

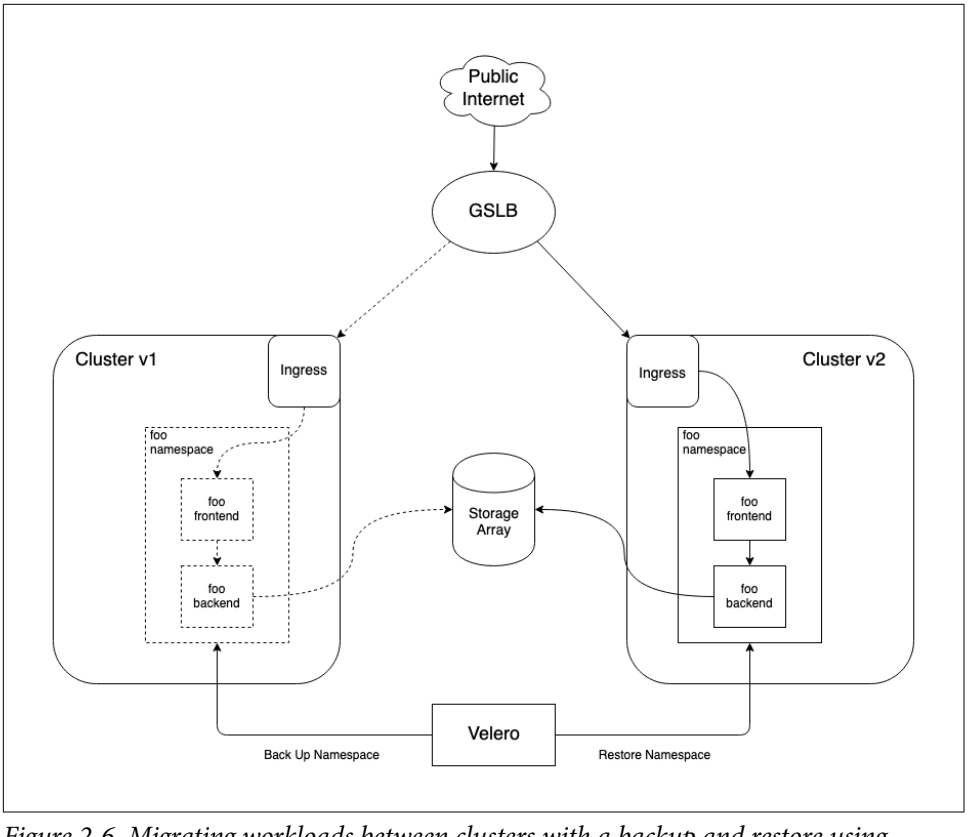

Figure 2-6. Migrating workloads between clusters with a backup and restore using Velero.

Node Replacement

This option represents a middle ground for cost and risk. It is a palatable option if you're managing larger clusters and compatibility concerns are well understood. And 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 4th member to a 3-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 co-located 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.

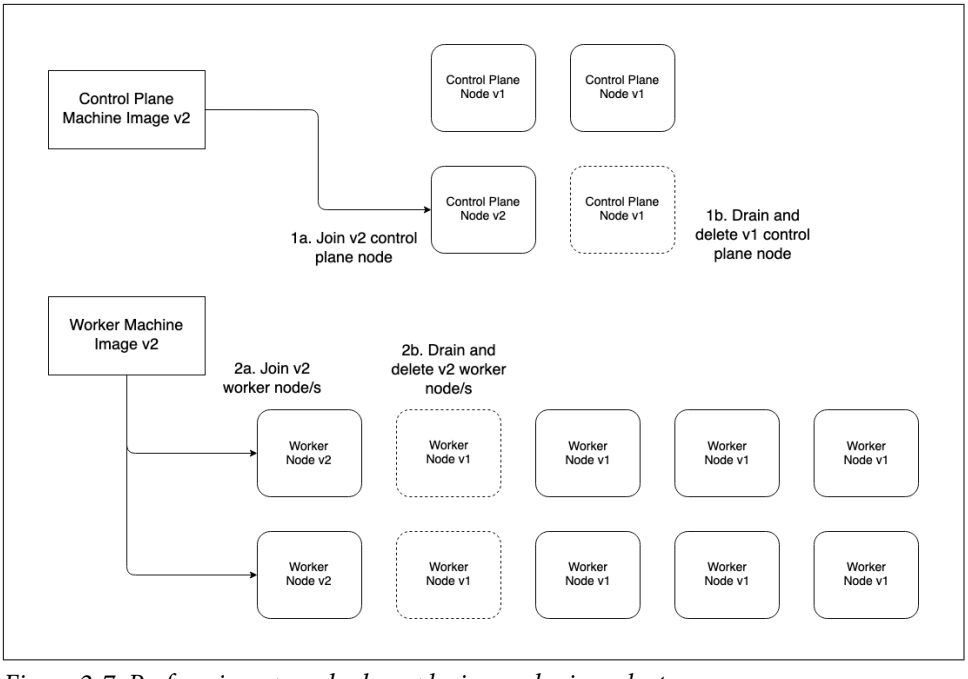

Figure 2-7. Performing upgrades by replacing nodes in a cluster.

In-Place Upgrades

In-place upgrades are appropriate in resource-constrained environments where replacing nodes is not practical. The roll back 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, Kubernetes in production configurations is a highly available system. So if in-place upgrades are done one node at a time, the risk is reduced. So if using a config management tools 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 as 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 pre-built 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 addons.

Another approach which 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 dependen-

cies 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 addons that will provide the essential features of your app platform. Version your platform and get your day-two management and upgrade paths in place before you put production workloads on your clusters.

About the Authors

Josh Rosso has been working with organizations to adopt Kubernetes since version 1.2 (2016), and during ths 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. Environments have ranged 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.

Alex Brand started working with Kubernetes in 2016, when he helped build one of the first open source Kubernetes installers at Apprenda. Since then, Alex has worked at Heptio and VMware, designing and building Kubernetes-based platforms for organizations across multiple industry verticals, including finance, healthcare, consumer, and more. A software engineer at heart, Alex 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.