O'REILLY®

Compliments of vmware® DELL Technologies

Overcoming Infrastructure Obstacles When Deploying Production-Ready Kubernetes

Nathan LeClaire

REPORT

DELL Technologies

VMware Tanzu
on Dell EMC VxRail

VxRAIL

Kubernetes at Cloud Speed

Accelerate Adoption

Automate Kubernetes infrastructure deployment and provisioning to accelerate developer productivity.

Automated deployment of curated VxRail clusters with vSphere with Tanzu seamlessly delivers developer ready infrastructure at cloud speed.

Kubernetes Your Way

Choice of infrastructure delivery options across edge, core and cloud that align to your organization's operating model.

Choose from validated architecture, native Kubernetes integration, or fully integrated Dell Tech Cloud Platform.

Rapid Kubernetes Evolution

Continuously, confidently and predictably take advantage of evolving Kubernetes technology on integrated, automated infrastructure.

Commitment to synchronous release cadence with vSphere with Tanzu, with automated, non-disruptive full stack lifecycle management across every deployment option.

https://www.DellTechnologies.com/Tanzu

Overcoming Infrastructure Obstacles When Deploying Production-Ready Kubernetes

Nathan LeClaire

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

Overcoming Infrastructure Obstacles When Deploying Production-Ready Kubernetes
by Nathan LeClaire

Copyright © 2021 O'Reilly Media. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Mary Preap
Development Editor: Angela Rufino
Production Editor: Daniel Elfanbaum
Copyeditor: Stephanie English
Proofreader: O'Reilly Media
Interior Designer: David Futato
Cover Designer: Susan Thompson
Illustrator: Kate Dullea

October 2020: First Edition

Revision History for the First Edition
2020-10-29: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Overcoming Infrastructure Obstacles When Deploying Production-Ready Kubernetes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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

This work is part of a collaboration between O'Reilly and Dell Technologies/VMware. See our statement of editorial independence.

978-1-492-09240-7

[LSI]

Table of Contents

1. What Problems Are You Solving and Are You Ready? **1**
Is the Business Ready? 1
Are the People Ready? 3
Are Your Systems Ready? 4
Modern Applications Initiative 5
Cloud Native Applications Are About "How" 6
Identifying the Low-Hanging Fruit of Apps 7

2. Tackling the Code **9**
Separation of Concerns 9
Containerization and Artifact Storage 12
Prepare for Running On Kubernetes 13

3. Tackling the Infrastructure **17**
Where and How to Run Kubernetes 17
Bare Metal, Cloud Provider, or Bring-Your-Own? 18
Number of Clusters and Size 19
Networking 20
Storage 22

4. Operations and Security **25**
Authentication and Authorization 25
Monitoring and Tracing 26
Log Aggregation 28
Scaling 30
Upgrades 32
Business Continuity and Disaster Recovery 33

5. People and Process. 35
Adopting a DevOps Culture 35
Immutable Infrastructure 36
Everything As Code 38
Updates and Patches Across the Stack 38
Fleet Management 40
Continuous Integration, Delivery, and Deployment 42
Wrap-Up 43

CHAPTER 1

What Problems Are You Solving and Are You Ready?

Modern businesses have to adapt to a changing infrastructure landscape to achieve their growth goals. Part of that transition has been the transformation in the enterprise to use tools such as containers and Kubernetes. These new tools, while offering many benefits, come with a high degree of uncertainty too. The pressure to use new technology to stay competitive is constantly vying with other needs of the business, such as operational reliability. A new deployment system and infrastructure has to solve more problems than it introduces.

How can an enterprise harmonize Kubernetes adoption with the need to protect itself, to preserve existing value, and to ensure compliance? In this piece, we'll discuss what you need to prepare for adopting Kubernetes in your business.

Is the Business Ready?

Why would you be interested in using Kubernetes in the first place? The constructs Kubernetes has allow us to move faster, deploy more software, and quickly roll back changes in the event of a buggy build. This workflow is beneficial to developers because it is can free them up from laborious infrastructure tasks and allow them to focus on shipping more code.

Some folks on your team might need some convincing before jumping in to Kubernetes with both feet. After all, there's a whirlwind of buzzwords and hype out there, and even for experts, it's hard to keep up. When articulating the value of adopting Kubernetes, especially in the enterprise, you need to meet your audience where their needs are. That includes addressing topics such as security, safe migration, and organizational challenges for rolling out Kubernetes.

Questions Kubernetes opens up include:

- Who is going to handle what responsibilities in the migration and day-to-day operations?
- What about teams with special hardware or security needs?
- How and where should we deploy Kubernetes?

All of these questions and their counterparts are the source of a lot of angst. That's what makes clearly articulating the business value of a container orchestration tool like Kubernetes vital. You'll need to be prepared to address a lot of objections.

In addition to addressing objections, you also want to proactively demonstrate what's great about Kubernetes. Kubernetes can enable teams to deploy incredibly rapidly compared to how teams managed their code rollouts in the past. Not only can it allow for that rapid rollout, but it has the primitives to make that scalable and address concerns about monitoring and availability.

Instead of spending time fretting over configuration of the infrastructure, your team can define their applications using code and track them in version control, so everyone can easily follow what's going on. Kubernetes then serves as the beating heart of your deployment infrastructure, absorbing those definitions and handling many of the scheduling hassles your engineering team would otherwise be ensnared with. Painting a vivid picture of life in greener pastures will go a long way towards successfully making a case.

Finally, once your organization is in alignment about the benefits of adopting Kubernetes, you'll want to manage expectations clearly. If the business is under the impression that there'll be an easy lift-and-shift from existing code running in virtual machines (VMs) to Kubernetes resulting in instant benefit, there will be a lot of disappointment. While Kubernetes is often presented as a panacea for a

variety of problems, the truth is that your managers and engineers need to be prepared for lots of refactoring existing apps to get them working properly on Kubernetes. If a migration is handled improperly, it can have catastrophic results for the business.

Are the People Ready?

Your company might also need to be prepared to restructure some of the organization for maximum effectiveness while adopting Kubernetes. Organizations with traditional structure often have a hierarchical system in which a few key architects have veto power. Such an organization might also set policies, design principles, and formal processes in place that everyone has to follow. This kind of rigidity can make it difficult for an individual or project to make a change. Businesses can begin to operate more effectively as they adopt Kubernetes if they can make small changes from a strict and hierarchical model to a more egalitarian one with a strong culture of ownership.

One way to help promote such a change is by developing a *site reliability engineering* (SRE) team. Instead of working on one specific service or subsystem, the site reliability team in your organization can focus on developing solutions that benefit everyone and improve application reliability across the board. The development of a site reliability engineering department can enable your development teams to breathe easier when it comes to production deployments as they know someone has their back when it comes to monitoring, infrastructure deployment, and Kubernetes best practices.

As you can see in Figure 1-1, teams can start small by spinning off operators or having bridge members that help them adopt cloud native practices without having to refactor the business to an edgy SRE model right away. These folks can be responsible for deploying software around or on top of Kubernetes.

What type of software will your SRE or pseudo-SRE team be deploying? One important addition is the introduction of a core control plane as a consolidated "single pane of glass." Instead of each team rolling their own solutions, a common platform will serve as the interface point for the underlying infrastructure. While forcing standardization can cause slowdowns initially, it's building equity by freeing the team to ship more code down the line. The SRE team can

also be responsible for deployments of shared components for systems such as monitoring and tracing.

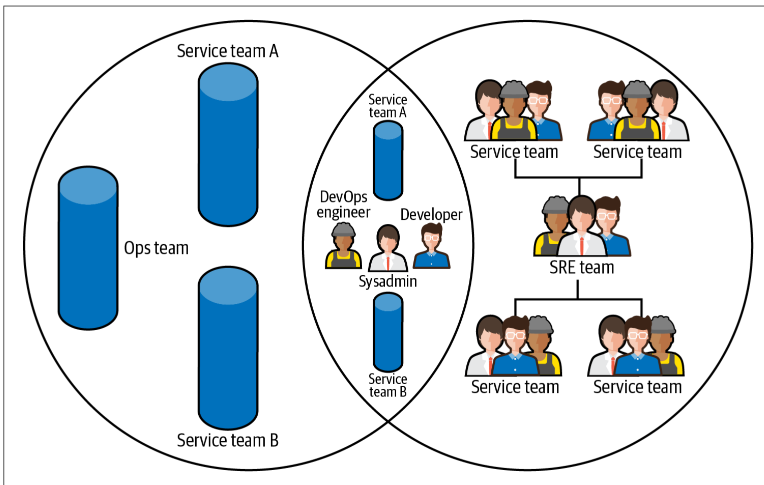

Figure 1-1. Enterprises and siloed organizations can learn from site reliability engineering practices slowly, as operators and developers from previously disparate departments work together on Kubernetes-related projects.

Once teams adapt to the addition of an SRE team and deployment standardization, they will be able to deploy more frequently and with less fear, and make improvements at bridging the gaps between the development and operations wings. Enabling a team to make smaller, more concise calls about what they need is the key to more easily scaling operations teams. Standardizing on a common underlying deployment platform (i.e., Kubernetes) as well as common tooling for interacting with it is key for success.

Are Your Systems Ready?

As we'll discuss in more detail in Chapter 2, preparing for an effective Kubernetes rollout on the systems side requires a lot of careful thought and planning to ensure it is done correctly. Getting too lax about security, for instance, could come back to haunt you.

For example, you could take special care in configuring your ingress and networking, which we will discuss in Chapter 2. You can also meticulously code review changes, audit what's happening within your Kubernetes cluster, and run a scanning system for your

container images to catch vulnerabilities in your dependencies quickly.

You can also look at existing workflows within your company to see where you can start making changes in the quest to adopt Kubernetes, and establish if your business is even ready for such a jump in the first place. Are you already automating common operational tasks, or is it tedious and manual? Are you using Git or other version control systems? Are you taking advantage of existing virtualization or cloud systems? If you aren't already taking these baby steps in the direction of cloud native, you will be in for a huge culture shock if you try to adopt Kubernetes. Take some small steps in that direction first if your systems aren't already headed that way.

Modern Applications Initiative

To successfully adopt Kubernetes, you must prepare the applications that will run within it to follow the way of cloud native. Kubernetes is a solution that can make your infrastructure sing when everything is done its way, but might make everything wail with horrible cries when it's not. Often, net new apps are the best fit to run on it, as they don't have to unlearn the old ways of the world they were originally coded for.

Kubernetes won't magically be able to take a traditional application, such as a legacy monolith depending on a huge SQL database, and transform it into a fast modern app. To get ready for your applications to run on Kubernetes, you'll have to perform a lot of prep work. While vendors might promise that lift-and-shift from an old model can be relatively straightforward, this couldn't be farther from the truth. Adopting Kubernetes requires adopting a whole new mindset. Everything from secrets management, to dependency installation, to networking will be different from what you might be used to when working with legacy systems. Of course, that can also be a good thing—many of you are probably fed up with the old way of doing things and looking for change.

Being able to use modern cloud systems and tools built on top of Kubernetes will enable engineers to be more productive. In a slow-moving environment, access to certain legacy data systems might be impractical or impossible. Adapting to the new landscape with heavy lifting done by cloud, container platforms, and trusted virtual control planes in the data center will help companies to speed up.

We'll discuss more on architecture and considerations for cloud native applications in Chapter 4.

Cloud Native Applications Are About "How"

Many folks have probably heard the term "cloud native" bandied about. It's easy to get an incomplete picture of what it means, and you might be surprised to find out that it doesn't just encompass the public cloud, but private clouds too. It's a philosophy and system of design that can be leveraged to deploy applications that can fundamentally handle resiliency, scale, and several other concerns naturally. Adopting cloud native practices can help your team learn, grow, and deliver high-quality software.

To best benefit from what cloud native applications have to offer, you have to be continually asking yourself "how." After all, it's not magic behind the scenes with tools like Kubernetes, but a system that will do exactly what you tell it to do like any other computer. So it helps to understand how it will be able to deliver on its promises —for instance, how will you scale out apps when needed? This requires an understanding of not only Kubernetes concepts, but how you will deliver the underlying hardware and resources to allow those apps breathing room. After all, the control of Kubernetes will simply not schedule our apps if the resource claims cannot be enforced.

What's another "how" for cloud native apps? One thing you can't take for granted is resiliency, i.e., how will you run Kubernetes with high availability, a stable networking configuration, and with the right trade-offs between flexibility and security? We'll discuss more on those topics specifically in Chapter 4.

Likewise, how will you properly adapt your current applications to conform to the Kubernetes way? Kubernetes enforces a new model for application architecture, which requires extra thinking about how you will define your workloads. The blessing of highly automated and flexible systems like Kubernetes comes with responsibility, too.

Identifying the Low-Hanging Fruit of Apps

As you identify ways that your business can move forward with Kubernetes and cloud native, identifying key areas to get started in will help enable your success. For instance, Kubernetes has a lot of obstacles and risks when it comes to running stateful workloads such as databases, so those types of workloads should be avoided for early picks. There's also no need to jump in the deep end of the pool by migrating tier 1 workloads, a project which would be risky for teams without much container experience under their belt.

Instead, for your first choices to deploy on Kubernetes, it's best to choose applications that are fairly simple, so they can be adapted with relative ease. There's probably a lot of low-hanging fruit on tier 3, 4, 5, and beyond that your team can pluck. In terms of identifying good candidates, there are some common qualities that will help you assess them. If they are stateless, that's a big win, because managing state on Kubernetes requires extra planning and caution.

The less pressure there is on the applications in their final environment, the better. Development use cases can excel, like allowing software engineers to run their forks on a test cluster to accelerate their workflow. Production apps that are only lightly relied upon (e.g., an internal best-effort app where no one needs to get paged if it goes down for a while) also can make for a great fit—they offer some of the excitement of deploying something to production, with low risks and an opportunity to experiment and learn what works well and what doesn't.

All in all, it's best to walk before you run—and before you know it, you'll be sprinting around the cloud native landscape like a pro.

CHAPTER 2
Tackling the Code

Kubernetes is a system for containerized apps. It allows us to decouple our apps from their scheduling layer, but it asks for discipline from us in return. A big benefit of Kubernetes is being able to declaratively specify what you want and let the "brain" of Kubernetes handle the heavy lifting of ensuring everything gets arranged correctly. The effort to port applications to the Kubernetes model eventually starts paying off quite quickly, but a Kubernetes migration can't be performed in a day—there are several hurdles your team will have to clear first.

In this section, we'll discuss the practical concerns your team will encounter as you move forward with tackling the challenges of a technical rollout of containers and how to address them.

Separation of Concerns

Traditionally, the applications you were responsible for and the manner in which you ran them in production tended to get conflated, with fuzzy separation between how the app was defined and how the app was run. For instance, vital configuration might live inside a database table instead of in a version-controlled system. Because traditional deployments often become a mishmash of code that is over-specific to the environment they're run in, porting applications from one environment to another can require a massive investment of engineering resources. When the underlying platform is not cleanly designed, the temptation is strong to make architectural compromises and reach directly into backing systems.

Kubernetes helps you separate out these concerns thanks to its clean design. It also helps you split out applications into discrete logical components, so that monoliths can be decomposed into smaller pieces for effective scaling. Reviewing the twelve-factor app methodology, originally popularized by Heroku, is a good idea for teams getting started with Kubernetes. Many of these guiding principles were key influences on the way modern container-based app architecture developed.

In Figure 2-1, you can see an example of some of the complexities that arrive at your doorstep without clean separations in what you are deploying. Processes can read and write shared local files with abandon, and upgrading or changing dependencies for one application can break others. Applications don't have any separation of concerns from their underlying infrastructure.

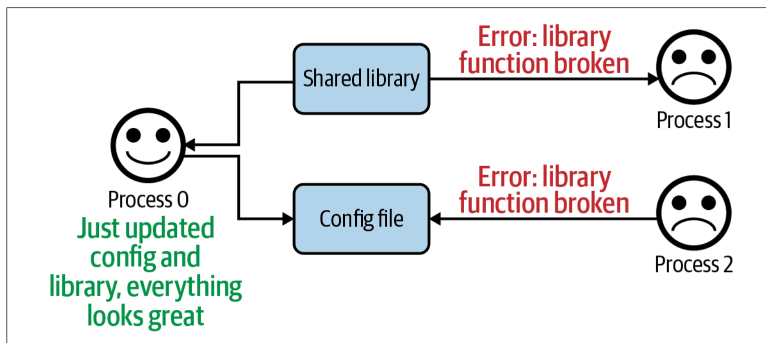

Figure 2-1. Cotenant processes traditionally share config files, libraries, and more, which can cause all sorts of chaos.

How does Kubernetes, in contrast, promote separation of concerns? It has been designed so that workloads are specified using a set of common best practices. Static pre-baked container images can help you fix the problem of libraries changing out from underneath your feet, but Kubernetes doesn't stop there. It encourages you to get even more modular.

For instance, if teams are sharing infrastructure, even making changes to something simple like an nginx reverse proxy can cause headaches. The right balance of easy-to-customize for a particular team, yet generalizable to all shared needs, might prove evasive. In Kubernetes, by contrast, you can cleanly separate the configuration

pieces into a ConfigMap, and isolate previously mixed use cases into their own deployments.

If a problem occurs in one deployment update, it won't affect the others. While it might have been too much hassle to have so many small deployments running around before, Kubernetes encourages and enables it. This is illustrated in Figure 2-2.

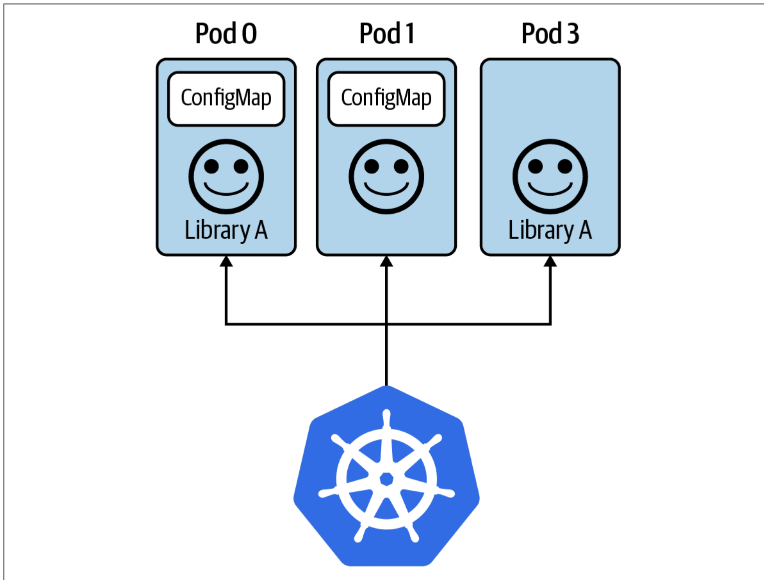

Figure 2-2. Kubernetes can help smooth over operational challenges by keeping workloads separate from one another and the underlying infrastructure.

Likewise, applications and the infrastructure code that surrounds them don't have to spend nearly as much effort fretting about *how* they are run in production. A regularly scheduled batch job is a good example. In your legacy systems, the code of the job itself might be performing operations such as querying a data store to ensure that only one job is being run at a time. That batch job might also log to an esoteric location and, generally, have a fairly cryptic (yet critical) operational lifecycle. In Kubernetes, by contrast, a lot of those concerns are handled by a standardized move to the Job resource. Users can use standard Kubernetes tools to check logs, Kubernetes itself will schedule the job appropriately according to the user-specified config, and any needed secrets or config already

provided for other workloads on Kubernetes can be reused without commingling.

Containerization and Artifact Storage

It's usually not very long into a deployment of containers that a lot of questions emerge about where exactly your images should be kept, how they should be kept, and how they should be secured. Because distribution of images was one of the things that propelled the modern container movement forward in the first place, you should hardly be surprised.

It's not uncommon for teams to start off simply by sharing Dockerfiles, which contain a simple but composable way of describing how your container images should be built. This basic technique allows developers to hit the ground running fast, because building and testing the images locally, as well as sharing them to a remote registry, is drop-dead easy. Unfortunately, what works well in the fast and loose world of local development doesn't necessarily line up with the needs you're going to have when you go to build and run images in intermediate environments and production.

It's natural for your team to then grow and evolve from that basic state as they continue to move forward. First, you might start off with a bit of shell glue and multiple Dockerfiles. You might then find yourselves in hot water as you reinvent the wheel in your own test-and-build systems, and such a system often gets brittle to maintain. That will naturally lead you to explore other options, such as Cloud Native Buildpacks. Cloud Native Buildpacks offer a prescriptive solution that helps you balance control between developers and operators, compliance requirements, and maintainability.

You will also need to decide where you'll store your container images by choosing an *image registry*. While there is a simple open source Docker registry available, there are also projects with a lot to offer on top of simple storage such as Harbor, a registry sponsored by the Cloud Native Computing Foundation (CNCF). Harbor doesn't just store and serve your container images—it also secures your builds by scanning and signing them. Your team also needs to weigh the cost of operating an open source image registry like Harbor against the option to use a vendor's solution.

Prepare for Running On Kubernetes

How do you need to modify or build your application so it can run effectively on Kubernetes? There are a lot of considerations, including:

- Is your logging ready to go?
- Do you have liveness, readiness, and metrics endpoints set up?
- What's the best way to architect the application's pieces with Kubernetes resources?

In this section, we'll discuss the nuts and bolts to consider when porting your application to run effectively on Kubernetes.

Logging

When adopting Kubernetes, you'll have to rethink logging for your applications. Kubernetes expects processes to write their logs to the console, not to an arbitrary file on the system somewhere as is often the case with legacy apps. It's no surprise, of course, that logs are an extremely popular way to gain access to information about what's happening inside an application. By adopting the best practices of container logging, we can make sure they're not just available for access, but indexed and highly searchable too.

Emitting lines of output to a log is a natural reflex for developers, and handling logging is well within the sphere of most operators' core competencies. When adopting cloud native practices, however, the traditional notion of what will happen with logs is no longer present. Any given app has a number of things that could go wrong, such as the underlying node crashing, that would make its logging output unavailable if it was simply written to the local system.

When such an event happens, Kubernetes will reschedule the process, but administrators still need to be able to access the logs to understand what has been happening with the app. So your apps will need to be changed to log to the console as Kubernetes expects. If you have a lot of control over how logging is done, this can likely be handled at the application level. This might be a simple code change to write to the console instead of to a local file. If the underlying app is more of a black box, you might have to find an alternative way to redirect the output.

Once logs are set up to make Kubernetes happy, the possibilities of what comes next are endless, such as sending to a centralized log aggregator. More discussion on this will follow in Chapter 5.

Lifecycle Endpoints

One of the ways Kubernetes expects you to conform to its model of how the world should work is by adding special endpoints to your code that expose information to the outside world about what's going on in your program. These will allow Kubernetes to make infrastructure decisions (such as whether to reschedule a Pod or when to start sending it traffic), and integrate with monitoring systems for better visibility into what's going on. Figure 2-3 shows a diagram of what some of these preparations end up looking like in practice.

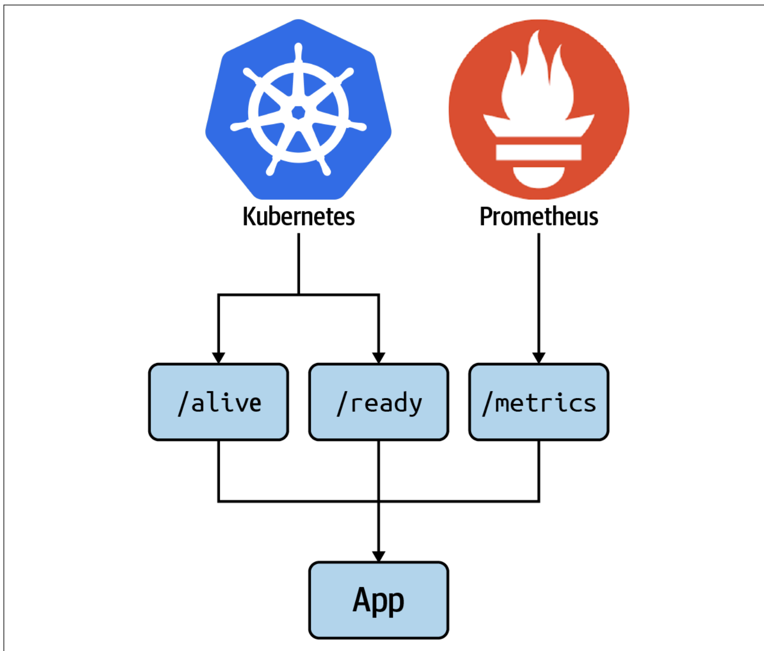

Figure 2-3. As you move your application to Kubernetes, you'll have to think about adding custom code to integrate with the "hooks" Kubernetes gives you to enable new automation and monitoring.

Some things to consider include liveness endpoints and readiness endpoints for liveness and readiness probes. Sometimes there's nothing quite like a good old-fashioned restart to fix problems that

are happening in a live application, but having to manually step in and do this is annoying. Is there a way we can communicate to Kubernetes that it should take care of that on its own? This is the function that liveness probes serve. After all, little is better equipped to report on an application's internal state than the application itself. With a liveness endpoint, we can *request* Kubernetes to restart our Pod if we hit an undesirable state.

Likewise with readiness probes, we can ensure that traffic is not sent to our app until it's ready. While it might be live in the sense that it is technically running and not in a broken state, it might have to do some setup work or have other delays before it's ready to serve traffic when it is started. Readiness probes allow you to break out this part of the deploy lifecycle with more granularity and ensure that when traffic is routed to your app, it's truly ready to serve it.

You're likely also going to want to expose metrics information for monitoring, which will allow you to track everything from how many HTTP requests your application is receiving to how many errors it has encountered. A common approach for this is to expose a */metrics* endpoint that Prometheus (an open source time series database) can scrape. You can likely even leverage existing libraries for Prometheus to get a lot of statistics without needing to do much legwork. While properly porting applications takes time and patience, it will pay off down the line in operations.

Rearchitecting for Kubernetes

There's no two ways around it—even just writing the specifications for your applications to run on Kubernetes will require some precision and elbow grease. Your team will probably soon find themselves knee-deep in the markup language used to define Kubernetes components (YAML), and though it might be intimidating at first, eventually understanding and generating sophisticated specs will be second nature. It will, however, require some planning to make sure you are conforming everything properly to the shape that Kubernetes expects.

It's not always clear what the mapping from a legacy system to a Kubernetes construct should be. Should this be a Deployment, which is mostly for stateless services? Should it be a DaemonSet that will run on every host? Or does it need to be something even more custom? Brainstorming the architecture with your team and

exploring your options will be needed to get the best result. Choosing the right boundaries for where to split things into their own containers or Pods will be something you'll have to tackle as well.

A lot of legacy monitoring systems, for instance, expect to have the ability to run some type of process side-by-side with the thing they're monitoring. It might be the best fit for those type of use cases to have a *sidecar container* that runs directly next to your app in the same Pod, so the monitoring system can access the things it needs to collect telemetry data. You'll have to make sure to integrate that in the specifications you write for Kubernetes.

Networking is one area that will probably require some whiteboard sessions before implementation. There are a lot of questions to address: What is the best way for your services to to talk to each other and to the outside world? Should you go for something fully open source or should you go for a vendor solution? Are you better off configuring everything yourself or are there ecosystem projects you can leverage? We'll discuss networking further in Chapter 3.

If you want to adopt distributed tracing (which we will discuss in Chapter 4), you'll need to update the code of the apps themselves to collect and send that data, a process called *code instrumentation*. This will allow you to visualize the flow of one request as it moves through multiple services and understand when one service is slowing down another.

These hurdles might look like fun challenges to some folks as they adopt new infrastructure, and they also might look like a risky slog to others. At any rate, it's true that we can't simply assume running applications on Kubernetes will be a straightforward lift-and-shift. It requires rethinking the way that we have done things previously. On the other side, however, is a better world where we have higher availability, applications structured according to industry best practices, and a happier engineering team.

CHAPTER 3

Tackling the Infrastructure

Given the enormous momentum and energy in the Kubernetes community, it's no surprise that end users have tinkered with deploying it on every platform from ARM, to embedded, to high-performance supercomputers. So, there are no shortage of options available for teams to evaluate when considering the important and fundamental question—once you've made a decision to run Kubernetes, *where* and *how* do you run the dang thing? Additionally, given that many teams are not Kubernetes experts, how are they supposed to evaluate the various options available? These questions will have a huge impact on your organization's success or failure in adopting Kubernetes.

Where and How to Run Kubernetes

Ultimately, most teams need to carefully and honestly evaluate their business needs, and weigh them against the various costs and benefits of possible solutions such as:

- Running your own Kubernetes cluster(s)
- Running on a cloud provider's Kubernetes service
- Running on premises using a trusted vendor's Kubernetes offering

Every team needs to decide for themselves what the right solution is, but in general, it's safest to assume that deep knowledge in Kubernetes is not and should not become a core competency of your

business. It's important to weigh the natural instincts engineers have to adopt new technology, learn more, and understand how their component systems work with a balance that encourages us to make smarter decisions about how much we can accomplish by simply investing with a knowledgeable partner to operate the systems instead.

To that end, having infrastructure-focused engineers make time for experimentation with the platform using tools such as minikube, Amazon Web Services Elastic Kubernetes Service (AWS EKS), and kubeadm during their sprints can be a good engineering investment. Using the many Kubernetes provisioning tools available can allow your team to dip their toes in the water with lower-stakes environments like development, test, and staging. It will also allow your team to get a realistic feel for the challenges of operating the platform in production. Your team might gain an appreciation for letting someone else handle the specifics, or come away with a renewed confidence in their own ability as they jump the hurdles of command line configuration, ensuring proper networking for the cluster, digging through administrative YAML, and so on. If there is buy-in from engineering to move forward with a specific Kubernetes distribution, the team can then be freed to do more productive tasks.

In these early experiments and evaluations, you will have some particular administrative concerns you want to pay attention to. In the rest of this chapter, we'll discuss the considerations of how many and what type of Kubernetes clusters you should deploy, networking issues your team should consider, and how your team should think about storage.

Bare Metal, Cloud Provider, or Bring-Your-Own?

Trade-offs you have to weigh when considering where to operate your Kubernetes cluster include operational expense, scalability, and platform independence. For instance, let's say you want to roll your own cluster. You might be tempted to glue together various ecosystem projects, but leaning heavily on open source tools has its issues. Downstream projects will often need to move at the same blistering pace as the upstream core Kubernetes if they're going to remain stable and usable. That journey may not be smooth for end users as

migrations can sometimes go haywire. The reality of open source projects is that they will sometimes prioritize making workable project decisions over a smooth upgrade process for end users.

Using a vendor's distribution of Kubernetes can alleviate many issues that crop up with trying to do it all yourself, albeit with its own concerns such as lock-in or a lack of flexibility. With Kubernetes, it's possible to deploy and manage distributed workloads without even touching the underlying machinery if someone else sets it up for you. A Kubernetes vendor will often have a blessed path for this type of setup for you to move forward with and additional goodies on top. It's often a good bet that using a managed Kubernetes platform, be it in the cloud or in the datacenter, will save you far more money than it will cost you in the long run. Not everyone, of course, has the luxury of using such a product, but for those who can, we recommend it.

Thanks to the constructs Kubernetes has to isolate workloads without the overhead of virtualization, it might be tempting to ditch virtualization entirely and go for a bare metal setup. While potentially a good fit for some use cases, this is rarely a good strategy to go all-in on. Virtualization and cloud offer so many benefits that they are generally well worth the costs they impose. After all, every cloud provider out there is using virtualization of some form to hyperscale their own Kubernetes offerings.

With VMs, the process to provision new nodes is simple. By contrast, in the physical realm you have to be prepared with unused server racks ready to go, which is costly. VMs also free you from having to worry about hardware compatibility issues, and spinning up new instances from a pre-configured template is trivial. Likewise, everything from monitoring to storage to software updates is made easier by the use of VMs due to the abundance of solutions and flexibility provided by the hypervisor.

Number of Clusters and Size

It's likely, of course, that your team is going to have more than just one monster Kubernetes cluster that every workload is run on. Some decisions about where and how to split out workloads will have to be made. Luckily, you won't necessarily need a new Kubernetes cluster for each team or project—Kubernetes namespaces, which offer a way to group related resources together, help you stay

organized and separate out workloads for different purposes. With namespaces, you can have a clean distinction in your cluster between user and system workloads, various teams, and various projects.

While namespaces are helpful, you might still find yourself in situations where you need to align clusters across different environments. Heterogenous environments such as a workload destined for on-premises and a workload destined for the cloud are some places you will need to deploy separate clusters. While theoretically feasible, the operational logistics of making a hybrid cloud or multicloud approach might be too much of a pain, and it might make more sense to split those workloads out into separate clusters entirely.

How about the number of nodes, and what type of nodes, you'll deploy in the cluster? There are limits on how large one Kubernetes cluster can grow. There's also an upper bound on the total number of Pods, containers, and Pods per node to consider. In addition to sizing the cluster appropriately, you'll want to invest some effort figuring where in the right balance lies for your organization between fewer large nodes and comparatively large swarms of small ones. You can, of course, mix node types and flexibly schedule Pods across different types as needed.

Networking

Configuring application networking can be a pain. Traditionally, processes would listen on a static or dynamically assigned port, but managing port allocations with both methods has issues. Statically assigned ports might require the operator(s) to maintain a spreadsheet or database about who is listening where, which is both tedious and error prone. Dynamically assigned ports solve some of these issues by choosing a place to listen arbitrarily, but require the downstream services to somehow find out what those locations are. Kubernetes solves these problems elegantly with its service-to-service networking model.

In Kubernetes networking, every Pod gets its own IP address, and that IP address can be assumed to have certain properties. This is highly compatible with porting applications that previously used to run on VMs to run in containers, since the networking model is similar. For any given Pod to communicate with another, it simply

has to access the IP address of the destination Pod. It's rare, however, that Pods will talk directly without using intermediate Service resources, which function as a gateway for service-to-service communication.

End users of Kubernetes can take advantage of constructs specific to their underlying infrastructure to accomplish this by making use of a plug-in conforming to the Container Networking Interface (CNI). CNI plug-ins run within a Kubernetes cluster and handle the details of underlying network machinery, i.e., they enable Pod-to-Pod communication. That opens up the door for a variety of implementations including, ones that lean on proprietary platforms and ones that should work when deployed to any type of infrastructure.

The flexibility of the CNI is a big blessing. So how do you know which one to choose? That will depend on your team's unique needs. If there's one native to the cloud platform you're deploying on, selecting that one is probably prudent. Indeed, many Kubernetes distributions have already made this choice for you, freeing you from have to worry about that in the first place. Failing that, consider what's most important to you in selecting such a critical piece of infrastructure.

Is it performance and observability? If so, Cilium might be up your alley as it has performed well in benchmarks. Is it simplicity that you're after? Flannel might be more your speed. Encryption and security? Take a look at Weave Net. Reviewing the list in the Kubernetes documentation and experimenting with a few different CNI installations to get a feel for their operational lifecycle is advised.

Load Balancing and Ingress

To have high availability for any app, whether it's an externally-facing server or an internal tool, you will need to set up and configure load balancing for your cluster. You will also need to ensure services can communicate with each other. You might be familiar with setting up and configuring a traditional load balancer by editing its configuration to indicate which backends it should direct traffic to, but things in Kubernetes work a little differently than you may be accustomed to.

For starters, groups of Pods can be exposed to other Pods within the cluster using the Kubernetes Service abstraction. By defining a Service for your deployment, a stable IP address will be assigned to the

group of Pods, and depending on configuration, a DNS entry too. This enables service-to-service communication. But once we have services successfully communicating with each other, how do we expose them to the outside world?

One possible way to do so is by defining the service as a NodePort type to forward a port from the Kubernetes host to the Pod, but that brings us back to the many problems associated with static port assignment we discussed in the previous section. Instead, we likely want to choose between a LoadBalancer type for our service or using an Ingress object. What's the difference?

A Load Balancer Service type will dynamically create an external load balancer once created using the Kubernetes API, so it's best suited for cases when that's the desired result. Usually, the platform on which you are running Kubernetes will have the "correct" type of load balancer configured by default (e.g., AWS Application Load Balancer (ALB) when running on AWS), and additional configuration options can be set through the use of additional metadata in your YAML specification. Over the lifecycle of your deployment, Kubernetes will take care of rotating Pods in and out as backends for the LoadBalancer automatically as needed.

Ingress, by contrast, is a completely separate API object type that can be used to create processes *within* the cluster that can handle load balancing, SSL termination, and name-based virtual hosting. Using Ingress is likely favorable to using LoadBalancer in many cases, since creating an external load balancer usually has extra costs. Ingress also has an appeal when it comes to portability—if you're using an open source solution like nginx, it can be picked up and used on another Kubernetes cluster with relative ease, even if it's on a different platform.

Storage

Earlier, we alluded to the fact that stateless workloads are a better choice for migrating to Kubernetes first than stateful ones. Why? What about workloads that are stateful by nature such as databases, or would require too much effort to port to be completely stateless?

To answer these questions, let's first look at the architecture of Kubernetes itself. Kubernetes is a system dedicated to helping engineers deliver resilient and scalable applications, and to accomplish

this, it has a specific set of expectations of apps that run on it. One of these is that a Pod could be terminated and restarted on another node at any time due to an eviction, a changed definition, or an infrastructure failure. That presents unique complications for stateful workloads—this operation might be destructive and unrecoverable for them if handled poorly. To address these challenges, the Kubernetes project has worked hard to make Kubernetes a friendly place to run stateful apps with a few different constructs.

One such construct is that access to more sophisticated storage capabilities than the default overlay filesystem is accessible using the API. Existing underlying storage created outside of Kubernetes can be modeled in the API and "claimed" by apps within Kubernetes using Persistent Volumes. Volumes can even be created on the fly by Kubernetes itself using dynamic provisioning. Apps that are stateful but can handle a reschedule, such as caches, can use ephemeral volumes to request temporary storage on either the underlying host filesystem or in an external volume (such as a mounted block storage device). At a higher level, abstractions such as StatefulSet then can help to tie all of these lower-level pieces together into a common operational framework.

Through all of this, Kubernetes offers a rich suite of built-in storage classes that enable support for common storage systems, such as AWS Elastic Block Store (EBS), vSphere volumes, and Ceph RADOS Block Device (RBD). You are not limited to the built-in classes either. Kubernetes has a Container Storage Interface (CSI) where drivers can be defined for arbitrary backing storage systems. While usually such customizability won't be needed thanks to the comprehensiveness of the built-in drivers, it's no small feat that the Kubernetes community has made this component pluggable and it's something that just might save your team's bacon if you end up having custom storage needs.

As you can see from the discussion in this chapter, Kubernetes offers many powerful tools in its belt but requires that the user configure everything heavily for maximum effectiveness. Doing as much homework as possible at understanding the ideal Kubernetes architecture for your own unique infrastructure needs will be required to succeed with adoption.

CHAPTER 4

Operations and Security

If you don't secure Kubernetes and its associated networking components, your business might get cryptojacked or worse. Likewise, just getting the applications onto the platform in the first place is only the first step in a long production lifetime—good monitoring is critical to make sure they keep running fast and smoothly. In this section, we'll cover operations and security challenges and possible solutions you should investigate while adopting Kubernetes.

Authentication and Authorization

Operations in Kubernetes entail a lot of challenges around client *authentication* and *authorization*. After all, how do you, the operator of a Kubernetes cluster, know and trust that the Kubernetes client attempting to connect and perform API calls is who they say they are and should be trusted? Authentication is the process of ensuring that the client making the request has a particular identity. Authorization is the process of making sure that, once authenticated, the client has permission to perform the action they're attempting.

These concepts are important for automation on top of Kubernetes as well as manual operator access because permissions should always be scoped down as much as possible. After all, cluster admin access on Kubernetes is root-level access on the whole cluster—otherwise, Kubernetes wouldn't be able to perform all the operations it needs to deliver its end results.

Kubernetes provides some constructs for managing these issues, but has limitations too. For instance, Kubernetes does not have any notion of a user account. Everything must be managed using certificates, role-based access control (RBAC), and service accounts, which are a way of defining a particular set of API permissions that a Pod running on the cluster is authorized for.

If an external client makes a request to the Kubernetes API, the API will check the certificates presented by that client to find out which user is making the request, and determine if the user has the correct permissions based on the cluster's RBAC configuration. While Kubernetes assumes that an external system manages the user accounts, it infers which user is making the request from the presented certificate, and checks whether the user has the right permission to perform the operation based on the RBAC config. For organizations that need to integrate with a form of external identity source such as Active Directory, Kubernetes also supports OpenID Connect (OIDC), a flavor of OAuth 2.0 that allows integration with such identity platforms. Just as we have discussed previously, integrating OIDC will require additional effort if you decide to roll your own cluster instead of using a pre-configured offering.

For workloads that are running within the cluster and need to perform operations using the Kubernetes API, such as updating the container image used by a pod, *service accounts* are used to grant the correct permissions. In fact, all Pods have a default service account, and it's important to check that the default one on your cluster is scoped down. While you generally want to restrict permissions, you might have other use cases where it's OK for Pods to have higher privileges. For instance, you might have a deployment bot that rolls out new versions of an app every hour, and it will need its service account configured correctly to enable this.

As with many aspects of Kubernetes, the documentation for both authentication and authorization are important to review in detail as you move forward. Those will help guide you towards ensuring that access to critical systems such as the Kubernetes API are safe.

Monitoring and Tracing

Monitoring and tracing play a key role in delivering on the promise of an optimal vertex between speed and safety. Taken together, monitoring and tracing broadly define what is known as *observability*:

the ability to ask questions of your systems and understand what is happening within them. The underlying source of a given issue might be a "good" (lots of interest in your public website) or "bad" (accidentally triggering expensive SQL commands that take down your system) root cause, but either way, you need to be able to understand what is occurring so your operators can fix it. Monitoring and tracing are necessities for the "measure everything" approach many DevOps teams take—not just for reacting to problems, but for improving the general health of your service and influencing ongoing development.

Distributed tracing is a technique that can help you achieve this goal. Tracing allows you to visualize the lifecycle of one request as it flows through different components of the system. Tracing is especially important if we're rolling out *microservices*, a popular architectural pattern separating apps into small decoupled services.

How do you want to approach monitoring and tracing in Kubernetes? Many teams start simply with infrastructure metrics monitoring. Seeing a big spike in metrics such as CPU usage, memory, or number of jobs waiting in a queue can be a big hint that something is way off in your systems. This might be handled a little differently than you're accustomed to, though, as you'll need to shift your focus away from the node level and towards the Pod and application level. This type of metrics deployment is readily achievable in many ways including by using open source systems such as Prometheus, which is a scrape-based system that gathers information from Pods every 15 seconds.

Prometheus is far from the be-all and end-all of monitoring, but it's a lovely option to have available because previously the availability of robust metrics time series databases was slim. Prometheus has heaps of solutions available for everything under the sun from use cases that need to push data instead of following the pull-based model, alerting, and creating gorgeous dashboards with Grafana.

Once you have monitoring set up, integrating a tracing system is a logical next step. Jaeger, which is depicted in Figure 4-1, is a CNCF project that enables you to send *traces* from libraries embedded within your applications based upon the idea of *span* blocks describing what's happening over time and how long it's taking. Based on this data, you can search for what's slow or having errors in your

programs, and follow along with what happened at each step in a call that spanned multiple services.

Figure 4-1. The waterfall view in a tracing system shows you what happened in your apps over time. Each span represents a timed chunk of work.

Jaeger offers a great all-in-one mode to get started, so it's an easy way to experiment and get your legs under you as you begin to explore the landscape of tools and standards such as OpenTelemetry. When it comes time to use tracing in production, you could scale up Jaeger by using a data store such as Cassandra or you could send your tracing data to a vendor such as Honeycomb. There are a lot of options available—give them a spin and see if your team will benefit from adopting tracing.

Log Aggregation

When managing production operations, dealing with logs is inevitable. Not only are they the most common way for your company's applications to output data describing what's happening within your apps, they are often your only way to figure out what's happening inside the black boxes your system relies on (such as the daemons of Kubernetes itself). Traditionally, processes might log to inconsistent places within your infrastructure, as depicted in Figure 4-2. Logs might end up in a system journal, a file in */var/log*, or some other destination altogether.

Logs in Kubernetes are a whole different ball game from what you might be accustomed to. We discussed porting your apps to log the Kubernetes way in Chapter 2. So what happens once that's set up?

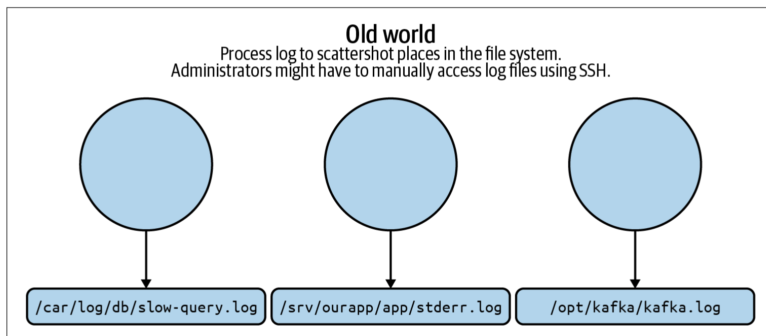

Figure 4-2. Managing logs is a perennial infrastructure pain.

Once the containers are configured properly, you need to extract those logs and make them useful. Kubernetes has some command line options available for viewing logs, but very little in the way of getting them all at a glance. Looking up logs from various parts of the system can be tedious and confusing. Given the ephemeral nature of containers, the Pod might be rescheduled and the logs gone by the time an operator gets a chance to look over what happened. Storing your logs in a centralized system is, therefore, important for operational, regulatory, and compliance reasons. *Centralized logging* is a way of importing all the logs from your Pods into one storage system for access and querying.

So what does tackling centralized logging look like? Figure 4-3 shows a diagram illustrating the basic principles.

In centralized logging, all logs (for both applications and the system level) are forwarded to a centralized storage system (such as Elasticsearch) that indexes them and makes them available for easy searching. Often, there will be intermediate or post hoc processing applied as well to add structure and make diving through the logs to find the relevant information easier for end users.

Intermediate layers like Cribl and fluentd can also be used to forward or "tee" the logs to various locations. For instance, one might want to archive all logs to system storage as they come in as well as forwarding them to an indexing system for free-form search.

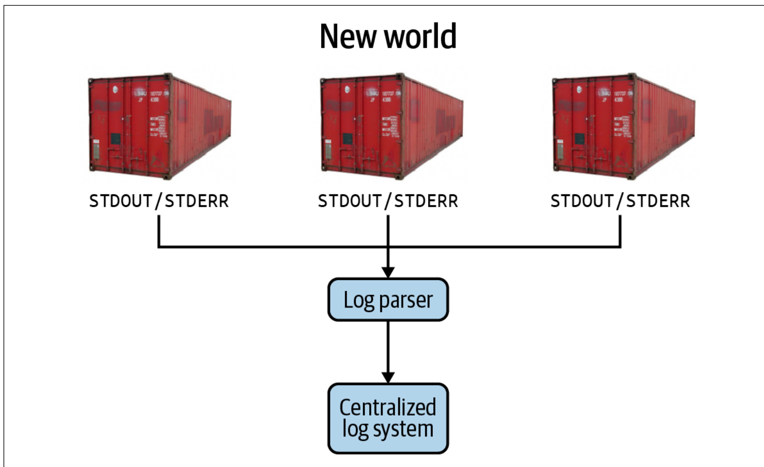

Figure 4-3. Kubernetes opens up new options for centralizing logging infrastructure.

Once the basic architecture is set up, you can tune to your heart's content. For instance, you might add more custom rules that parse structure out of otherwise flat plain-text logs. You could also set up indexes for fast searching on relevant fields in that parsed structure. Eventually, your team might even consider sophisticated techniques, such as running machine learning on logs, to identify problems before they affect uptime. The centralization of logs has many benefits and, especially since Kubernetes has and encourages a lot more moving pieces than older techniques, will help your team to troubleshoot production issues more quickly.

Scaling

A few options exist for scaling a Kubernetes cluster and the workloads running on it. Usually, you want to focus on scaling out the Pods that define your deployment. To that end, you have the option of *vertical scaling*, where you increase the amount of resources available for the underlying processes, as well as *horizontal scaling*, where you increase the number of independent replicas of the service. Horizontal scaling in particular is one of the core tenets of cloud-native applications.

Vertical scaling is the type of scaling you're likely familiar with from legacy systems. To scale vertically, first you will need to make sure that the underlying machines have resources available to handle it.

Once you have confirmed this, you can define new resource limits for the deployment in question and reapply the specification. Kubernetes will automatically take care of terminating the old underpowered Pods and rescheduling the newly beefy Pods where they can claim the resources they need. That method might well take you pretty far on its own, and if you happen to be running in the cloud, it's probably pretty straightforward to bump up the instance type to accommodate. This can, of course, be done as a rolling operation where Kubernetes reschedules workloads as you power cycle the old machines to change their type.

Unlike the scaling up performed in legacy systems, it's more likely you will find yourself scaling *out* in Kubernetes by deploying more replicas across the cluster with horizontal scaling. Using horizontal scaling, you will increase the replica count of your Pods and let the Kubernetes manager do its thing and start scheduling more Pods. Just like simple architecture practices can often end up working fine for a very long time in a web application, this basic framework can get your team very far, and early effort on a deployment is best invested elsewhere. It's not too hard to know when you might want to scale out when you have good monitoring like we mentioned previously, because your system will be reaching saturation points for metrics like CPU and memory.

Horizontal scaling is particularly fundamental for cloud native applications, and it requires a rearchitecting of applications to ensure that they will continue to perform well under this model. While it may be a bigger lift than the old "throw resources at the problem" approach, it helps improve goals such as availability, reliability, and performance. With horizontal scaling, your system will be able to handle a failure of some parts of the system better. As we have discussed previously, Kubernetes's rescheduling upon hardware failure hits much less hard in a world where you are not dependent on powerful pieces of specific hardware. Likewise, horizontal scaling can help with performance—an operation slowing the process down in one replica will have a smaller blast radius, and may even get taken out of the load balancing rotation entirely if it can't respond to its defined probes.

When it comes time to get more sophisticated, Kubernetes has some nifty options on tap. The Horizontal Pod Autoscaler will run calculations based on your system metrics to determine when Kubernetes should go off and scale up on its own without any need for manual

operator intervention. As you can imagine, this takes some fine tuning to dial in properly. Set the thresholds too tight, and you'll suddenly have a much larger fleet on your hands, burning resources that it doesn't need to. Set them too loose, and the system will be frustratingly sluggish. As with all things, experimentation will help unveil where there might be a good fit for you in autoscaling.

Kubernetes also has all sorts of clever bells and whistles for users to be as specific as possible about how important an application is. For instance, as your team gets more comfortable with resource limits, it might start looking into operations such as setting Pod Disruption Budgets. Pod Disruption Budgets limit the number of Pods of a replicated application that can be down simultaneously. This can help ensure that very critical and sensitive applications continue to run with high availability when Kubernetes's powerful but aggressive reconciliation magic is happening behind the scenes.

Upgrades

Once you have a Kubernetes cluster that you're using for real workloads, you'll eventually have to deal with the topic of upgrading the underlying cluster. Kubernetes is a fast moving project and new versions come down the pike about once a quarter. Your team will need to be prepared to handle the upgrade process, for many reasons from securing vulnerabilities to enabling the latest and greatest features to play with.

What will need to be upgraded, and how should you handle it? The answer to those questions depends on which type of Kubernetes distribution you're using. A big advantage of a managed or vendor-provided Kubernetes solution is that this can largely be taken care of for you—sometimes without your team even noticing or incurring downtime. In the cloud, providers will often perform such an upgrade for you, or make it as easy as clicking a button. A Kubernetes solution that lives on-premises likely won't upgrade automatically, but it will provide a clear pathway forward, even if that includes working directly with folks from your vendor to safely roll out the change. There are middle ground options, too, such as using an open source tool that handles some of these processes for you, such as Rancher.

If the cluster was created with a tool like kubeadm, on the other hand, you may be in for a long and multiple step process to properly

upgrade the cluster (for instance, you can take a look at kubeadm's upgrade process). You'll have to verify which operations are safe to perform in which order ("Can a manager node on this future version operate with worker nodes on previous ones?"), and make sure that workloads are properly drained and separated from the components that are actively changing. You may need to concern yourself with updating a component such as etcd, which is the shared memory system of a Kubernetes cluster. You will also need to religiously stay on top of upgrades over time because skipping a version is likely to result in problems. That's a lot of work for your team to take on!

Even with the best preparation in the world, things can sometimes go wrong. When you go to upgrade Kubernetes, and in the cluster lifecycle in general, having a clear picture of what a recovery process looks like in the event of a failure is key. We'll discuss how to think about handling that in the next section.

Business Continuity and Disaster Recovery

Your business might be accustomed to doing things a certain way when it comes to business continuity and disaster recovery, but dealing with these in Kubernetes is a totally different animal. Some on your team might claim that disaster recovery planning can be handled simply by snapshotting live VMs. Backups are certainly necessary, but not sufficient. It makes sense to have a snapshot of things like a database's state, but what happens when you need to restore it? Is your team prepared for that, and have you done a restore lately to verify that it's possible? Snapshots might have been a crutch your team could lean on before, but the concerns when operating a Kubernetes cluster will be different.

Instead of worrying about protecting individual nodes, in Kubernetes you shift your focus to a higher level up. After all, in the case of a node failure in Kubernetes, the Pods will just get rescheduled, and your service will have minimal downtime. You need to have a bigger picture perspective for recovering from disasters. For instance, how many manual steps would you have to take to restore the whole system from scratch? In the cloud, you should be able to get pretty far with the push of a proverbial button. You should be taking the time to root out manually created or administrated components and replace them with automation wherever possible.

You also need to understand what happens in the event of an outage or catastrophe in varied environments and how your team will adapt. Can the Pods from one environment be seamlessly run in another, either by rescheduling or by reapplying your spec? Or do you need to be prepared for additional steps in such a forced migration? Your team should be prepared with automation and runbooks to guide them in the case of such an event. In the next section, you'll see how following best practices such as GitOps makes this process of redeploying workloads easier, especially stateless ones. This is one of many benefits of the infrastructure as code principle, which we will also discuss.

Lastly, you don't have to tackle the challenge of Kubernetes backups and restores alone. You don't need to throw your hands up in the air and eschew backups or disaster recovery simply because your business has switched to Kubernetes and the old ways no longer apply.

Operations and security in Kubernetes, like many aspects of the platform, will reward the careful planner and punish the impatient. So take the time to make sure your Kubernetes cluster is secured, hardened, and operationally sound. It will help you and your team sleep better at night.

CHAPTER 5

People and Process

Adopting a DevOps Culture

DevOps is both a mindset and a set of practices, and your team might be all over the map in terms of making progress towards its adoption. DevOps is all about collaboration between previously disparate departments—developers working more closely with operators to ship software more quickly and deliver higher quality. The promise of such a movement has led a lot of vendors to imply that DevOps is something that can be purchased and staffers to infer that it's someone who can be hired, but it has to be a cultural transformation first and foremost.

Because DevOps is not something you can buy off the shelf, organizations must turn inwardly and be honest with themselves about how they can adopt a DevOps culture and encourage the type of behavior they're looking to get from it. So how can you do that? One step is to survey your organization and where you are today, and sketch out a piecemeal roadmap towards greater collaboration and agility. For instance, as we mentioned in Chapter 1, you can start forming some bridge teams that function as SRE-lite units. You can and should get started without dramatically shaking up the way your organization functions.

Your team needs to get in the habit of praising work that makes life better for everyone. Instead of giving all the accolades to heroic contributors who pushed user-facing changes, your team needs to shift towards valuing work that automates common tasks and makes

deployments more reliable. If leadership values and praises increased collaboration over solo acrobatics, the individual contributors will get the hint and start shifting their focus.

Likewise, we need a way to quantify and track our progress, such as agreeing on some common goals around mean time to resolution, deployment frequency, and how much downtime is acceptable. Many teams are finding value out of setting programmatically tracked service-level objectives—quantifiable performance goals for your apps that will help your team understand how things have been going lately and where they could do better. Your team will, of course, hone in on improving these metrics to the detriment of other needs if they are fretted about too aggressively, so they need to serve as guides that are open to changing rather than as holy law.

It's also important to build empathy for the folks on the other side of the fence, be they in Development or in Operations. Operations, perhaps unfairly having gained a reputation for being a buzzkill and slowing the shipment of new things down, needs to cultivate an attitude of *how* they will help developers ship new software quickly, not *why* they can't. Developers, by contrast, need to develop a shared sense of pain and ownership with the operations team. A fantastic way to build this is to make developers pick up the pager and be on call for production issues. While it won't whet their appetite to move quickly, it will help them see the challenges of doing so for reliability and learn to adapt their application code to be reliable before it hits production.

You can have great DevOps, you can have a great team, and you can have great architecture. All these things are possible in enterprises, just as they are elsewhere. Best of all, those are all just pieces of the bigger picture. You and your team can be a part of creating and leveraging the open standard of great enterprise IT in Kubernetes.

Immutable Infrastructure

Large organizations are obsessed with safety and with protecting their existing position in the market. It should come as a shock to us all, then, that it has been so commonplace for operators within those organizations to constantly log into their systems and perform manual changes to the infrastructure. Not only does this type of workflow have a lot of risks, it's also inefficient and maddening for the operators themselves. How can we migrate from these legacy

workflows to a better way? One way is by adopting the principle of *immutable infrastructure*.

Immutable infrastructure is the notion that once a system has been defined and applied, it should not be changed without also updating the definitions. Users shouldn't be SSHing into a server and making changes on their own. This can manifest in a variety of ways. One way is that teams will create "golden" VM images by snapshotting VMs that boot up with everything already installed and configured. While there might be some minor mutations on the local filesystem or state supported through an attached block device, generally it's expected that nothing new will be installed or configured. This helps to ensure the operational lifecycle remains sane and predictable.

New container technologies take the idea of a golden filesystem image to its logical extreme. As we've discussed, they enable building those images out quickly as well as easily sharing them across various environments using a registry. One of the most difficult things about production troubleshooting has always been to deduce what's different in the state between production, staging, and development environments. With golden container images, businesses can make fantastic progress at chipping away at the surface area of that problem. Stateful applications will still require some finesse, but their state can largely be confined to a small surface area and everything else can remain immutable.

The practice of immutable infrastructure is not only confined to making golden images. New trends such as the GitOps philosophy encourage users to encode everything from top to bottom in your infrastructure in a declarative way, tracked and version-controlled using a system such as Git. That way, a number of benefits are conferred by default. You can easily see who made what changes to the infrastructure and when because Git and its surrounding technologies have a built-in auditing trail. You should also be able to roll back to a previous state of the infrastructure with minimal hassle because Git makes such an operation trivial at the code level. This is directly tied to the concept of having everything as code, which we will discuss in the next section.

Everything As Code

Each Kubernetes cluster has a unique life of its own. It is birthed into this world, whether by connecting the components by hand or by a single API call. Then, it has to effectively continue to operate and run workloads. While it might be tempting to do everything in this lifecycle by hand to get going quickly, having the patience to follow an *everything as code* principle and codify it instead can result in huge benefits to you and your team. Following this principle means that everything you rely on, including your infrastructure, pipelines, test suite, and more, are tracked as version-controlled code that can be automatically redeployed at will without manual intervention.

For instance, let's say everything is going great with your team deploying and operating your Kubernetes cluster, when suddenly a team has a new requirement to deploy an entirely separate cluster in a specific geographic region. If you haven't "documented" the process you used to create your other cluster(s) by defining the infrastructure as code, you'll be in hot water when it comes to stamping out such a new cluster. On the other hand, if your team has everything committed to Git and ready to go as a templated deployment manifest, the process will be relatively straightforward and easy in comparison. Likewise, the issues don't stop once the cluster is deployed—what if something changes in the "root" cluster, such as a security patch, and that change needs to be applied to the new cluster as well? Challenges like needing to deploy a mirrored environment are more common than novice operators might think due to the complex regulatory climate we find ourselves in, and investing a few hours now to save weeks of time down the line often is a wise choice.

Updates and Patches Across the Stack

Any container-based infrastructure is a layer cake of sorts, starting with a foundation at a physical computing level and working its way up through various abstraction layers from hypervisors, to underlying orchestration components such as the ones Kubernetes combines, to the final workloads running atop the cluster. All too often, it's the case that the people and processes responsible for managing these various layers do not share responsibilities and do not naturally coordinate amongst themselves. This can cause tension when the time comes to update or modify part or all of the stack.

Even the Kubernetes layer itself has many layers that can be hairy to manage. For example, let's consider what operating a production instance of Kubernetes is like if you are running your own Kubernetes distribution. Your life is made a little bit easier by the use of tools like kubeadm, but there is no one-stop shop for all of your cluster automation needs and the choices you will have to make. Are you comfortable assuming the responsibility if an error made in managing the stack results in another team's workloads becoming inoperable? Are you making the right decisions for pluggable components we've discussed previously, such as networking, storage, and scaling? Are you going to make opinionated choices about what should be done yet another layer up with systems such as Helm, or will you decide to expose the underlying cluster directly to end users?

These decisions will need to be carefully managed while aligning key stakeholders, including the leadership and the engineers who will deal with the consequences of your decisions. You don't want to end up in a situation where the engineers who are end users of your platform are in open rebellion against the decisions you've made and work around you, building out yet another layer of shadow infrastructure and defeating goals you might be striving for such as uniformity in deployments. Likewise, you will need to understand who gets paged when something goes wrong, what the procedures will be to fix software, and what your team will do when a critical patch needs to be released for security or stability purposes.

If one team alone is responsible for managing the Kubernetes cluster(s), that will simplify things a bit, but if there are multiple teams, the division of responsibilities will have to be made clear. Of course, your team could decide to skip that particular hassle and use a turnkey vendor distribution instead. A turnkey distribution of Kubernetes will alleviate some of the burden on your team, but still entails relationships to manage. Of course, there's always the question waiting for you at the end of setting up all these underlying layers of what will happen when developers actually start rolling out code on top of Kubernetes. Do they write the manifests and deploy the code themselves? Do we make everyone use a shared framework that generates YAML for them and/or standardizes on something like buildpack-based images? Who gets paged when a deploy goes bad? These are all questions that someone will have to answer eventually, and it's best to address them in advance.

Developers are likely to have to take more responsibility than they're accustomed to. A traditional model might entail larger updates at the code level and less frequent deploys, but with fewer changes to the underlying dependencies. A container-based deployment, by contrast, practically eggs developers on to change underlying dependencies and deploy more frequently, and that entails additional responsibility that they need to be ready to bite off. Like a supercar, Kubernetes can help you go fast and have fun, but also requires planning for the cost of maintenance and the learning curve. Not everyone is comfortable or well-trained enough to drive it, and especially for developers who might be new to operational concerns, you need to think twice before handing over the keys.

Fleet Management

You're likely to end up with multiple Kubernetes clusters. This fact of life, naturally, raises the question of how exactly all these fleets of machines will be managed. Both technical and human challenges will need to be tackled. As we've discussed previously, there's always a balance between letting folks have direct access to infrastructure and operational consistency. The last thing you want is many different ways of doing the same thing deployed side by side.

First, we suggest you turn your focus to the human side of the house. Map out what environments make sense to group together, which ones have special considerations, and which people will be the owners responsible for handling each environment as your team moves forward with a rollout of Kubernetes. You want to avoid nasty shocks down the line where some subsection of infrastructure wasn't planned to be integrated in the rollout, resulting in duct-tape-style fixes, as well as infrastructure no-man's lands where the owners aren't clear.

Once this landscape has been plotted and a loose strategy has been defined, teams can begin brainstorming how they will manage the intermediate layers that are required for Kubernetes to operate effectively but aren't necessarily part of Kubernetes itself. A healthy fleet will have answers for the questions we previously discussed vis-à-vis the infrastructure layer cake. Fleet management is about managing the nodes that will become part of Kubernetes clusters as much as it is about the clusters themselves. Do you have solutions available for bare metal server instances to be provisioned? How about VM

templates? The more of this that operators within your organization can confidently reach for off the shelf, the better off your team will be.

In addition to the organizational concerns, there is open source and proprietary software available that can help faciliate the management of multiple clusters. For instance, *cluster federation* is a feature of Kubernetes being worked on by a Special Interest Group. Federation helps users with multiple distinct Kubernetes clusters to manage them more centrally. A sample of the architecture in such a federated setup is depicted in Figure 5-1.

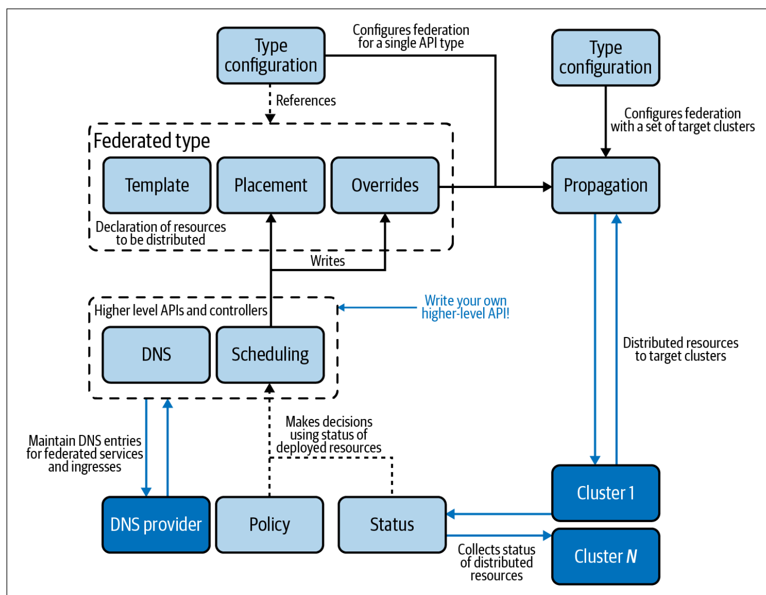

Figure 5-1. While complicated, the dream of managing multiple distinct Kubernetes clusters from one control pane is slowly becoming reality. This diagram shows the architecture for the current working proposal for cluster federation in upstream Kubernetes.

Continuous Integration, Delivery, and Deployment

When adopting a DevOps mindset and Kubernetes, it's almost inevitable that you'll find yourself crossing paths with the practices of *continuous integration*, *continuous delivery*, and *continuous deployment*. Continuous integration is the practice of merging code changes with the upstream as quickly as possible and verifying that they do not introduce any issues. This can be done by running an automated test suite constantly to check if the new code breaks any of the team's tests, which specify how the app should behave. These tests can not only be run on the main branch of your code repository, but on the individual developers' pull requests as well, ensuring that no obviously bad code ever makes it upstream.

Continuous delivery is the practice of leveraging this state of continuous integration to ensure that code making it into the version control system is available in a deployable state. Not only can we automate our test suite to merge code more safely, we can automate the whole process of building its associated artifacts, such as JAR files, container images, and deployment manifests. This practice has a natural friend in Kubernetes, whose attitude of automating as much as possible and storing deployments declaratively is a good fit to enable this practice. The end goal of continuous delivery is to be able to wave a magic wand and have a new version of software delivered to end users, fast. Thanks to the safety offered by continuous integration, this can be done more safely and with fewer operational headaches.

Continuous deployment is an even more radical step in this direction, where all changes are deployed to higher environments shortly after landing in the upstream branch and passing tests. This might imply a staging environment where operators perform final smoke checks before green lighting a production deployment, or it might even be a deployment directly to production. This encourages a fluid workflow for engineers and customers as an idea can move from experiment to change request to production rapidly. Failed builds will, of course, not be deployed until manual intervention from an operator gets everything back to normal.

While it might seem scary to deploy software constantly, and indeed might be too big of a jump for many teams to adopt immediately,

the practice of deploying continuously actually helps releases to be a lot less scary. Problems can be fixed and remediated as they happen, instead of accumulating unseen for months between releases. This might require a shift in mindset as we have discussed, where engineers have to take more production ownership, and the idea might be a hard sell at first, but often increases morale as developers can actually touch and affect their code running in production.

Like it or not, to stay competitive your organization *will* have to adopt some of these practices to stay competitive in a world where more businesses than ever are deploying quality software quickly. The good news is that these changes can be adopted incrementally, and they can be adopted alongside Kubernetes as your team moves forward with that. As you tackle challenges such as making sure each code release has its own container image built, you will likely find that continuous delivery pipelines grow organically. Doing those types of operations continuously will become second nature to your team, and your software will be better off for it—code quality will be rigorously checked by the test suite, expected to adhere to certain practices, such as infrastructure as code to fit in with the automation systems for delivery, and maybe even some day delivered just as quickly as it's written and approved.

Wrap-Up

There are a lot of challenges when deploying Kubernetes to production, but none are insurmountable. Armed with a bit of grit and the knowledge set out in this book and elsewhere, you and your team really can make effective change and have a great DevOps culture. Containers are an important part of the modern DevOps story, and your organization can benefit from their use, but you need to plan carefully. Following the Kubernetes blog can help you keep your eyes on what's happening in the community and what you need to be prepared for. And, of course, there's no substitute for lots of hands-on experience.

So, fire up those test environments, get your hands dirty with a real live Kubernetes cluster, and don't forget to have fun too! You, your company, and your infrastructure have a great future ahead of you.

Acknowledgments

Special thanks to Chip Zoller (@chipzoller) of Dell Technologies and Keith Lee (@KeithRichardLee) of VMware for their deep Kubernetes and infrastructure expertise. Your crisp vision, experience, and generosity of time to advise and edit countless revisions were invaluable.

About the Author

Nathan LeClaire is a Go programmer and author living in San Francisco, CA. He participated firsthand in the development of container technology at Docker and got to know Kubernetes and related systems well. In his free time, he likes to cook steak, drink whiskey, and listen to the Grateful Dead.