O'REILLY®

Kubernetes Best Practices

Blueprints for Building Successful Applications on Kubernetes

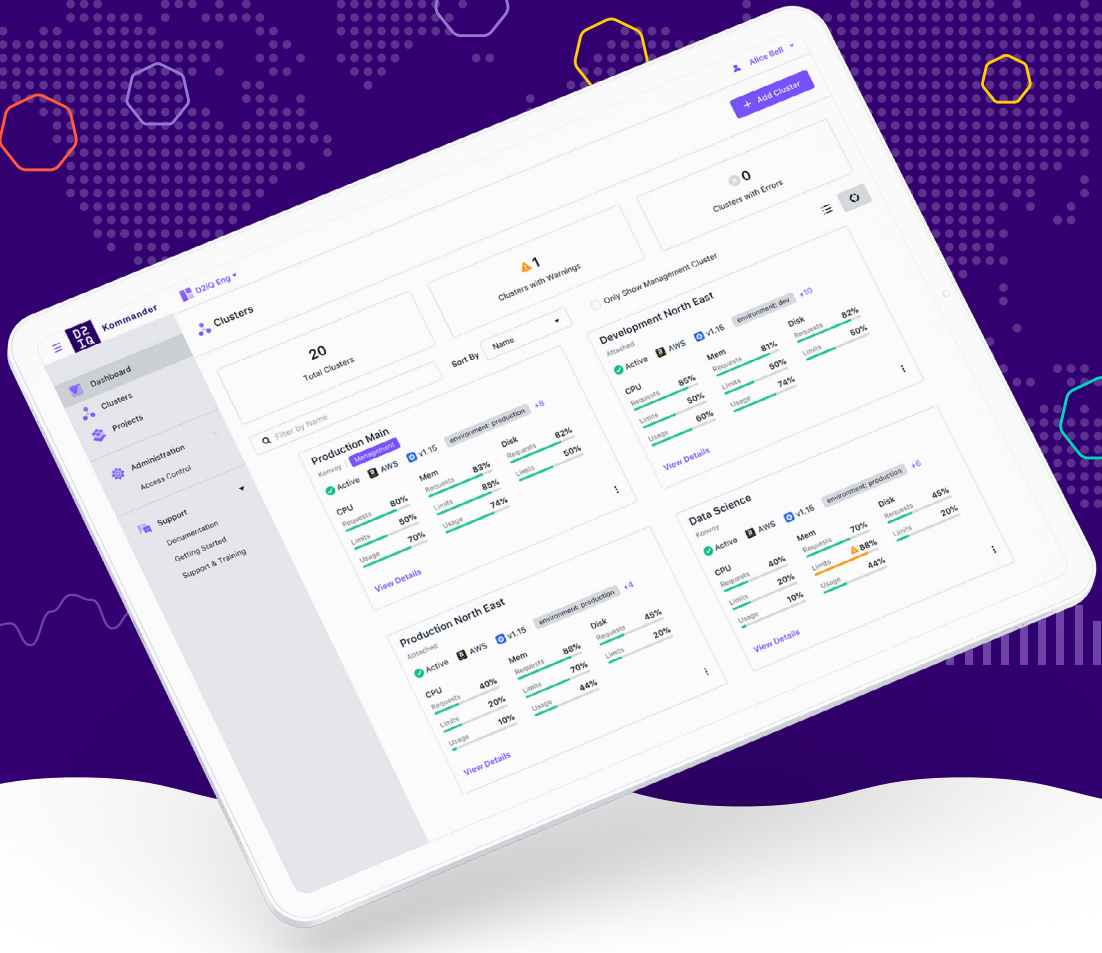
Free Chapters

compliments of

D2 IQ

Brendan Burns, Eddie Villalba,
Dave Strebel & Lachlan Evenson

Take Control of your Kubernetes Multi-Cluster Operations



D2iQ Delivers Governance and Manageability Across the Kubernetes Landscape

Try it Yourself

Kubernetes Best Practices

Blueprints for Building Successful Applications on Kubernetes

This excerpt contains Chapters 2, 11, and 12. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson

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

Kubernetes Best Practices
by Brendan Burns, Eddie Villalba, Dave Strebel, and Lachlan Evenson

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: Virginia Wilson
Production Editor: Elizabeth Kelly
Copyeditor: Charles Roumeliotis
Proofreader: Sonia Saruba

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: First Edition

Revision History for the First Release
2019-11-12: First Release
2020-07-10: Second Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=0636920273219* for release details.

978-1-492-05647-8

[LSI]

Table of Contents

Foreword

To remain competitive in today's fast-paced, digital world, organizations need to adopt technologies that enable them to innovate and scale, often at light speed. One of the most popular open source technologies for achieving this sought-after agility is Kubernetes, which has become the de facto standard for cloud native deployment and container orchestration. In fact, our recent research found that organizations expect that projects in production using Kubernetes will rise 61% in the next two years. Kubernetes remains a popular choice for container orchestration, enabling organizations to accelerate their cloud native journeys and better compete in today's digital world.

Kubernetes has many features (such as declarative APIs, autoscaling, RBAC, and network policy) that provide availability, scalability, and security, and make it the platform of choice for container orchestration. However, to use these features most efficiently, you need to follow a set of best practices around monitoring, logging, deploying, and upgrading to automate application onboarding and streamline operations. Adopting best practices not only allows organizations to establish standardization early in the process of Kubernetes deployment, but also enables them to manage the complexity of running Kubernetes in production and to avoid any pitfalls and resultant slowdown in transitioning to Kubernetes.

As organizations realize the enormous benefits of Kubernetes and their workloads grow substantially, they often have to make a choice between a single large cluster or multiple clusters. While cost is a major factor, other factors that could influence the choice are regulatory and legal requirements, physical infrastructure constraints, fault-tolerance against infrastructure failures, or maintaining isolation among independent business units. Managing multiple clusters puts more pressure on your IT teams, requiring massive time and resource investments. Chapter 12, *Managing Multiple Clusters* has a lot of great information about these issues and how to address them.

Further, as the number of Kubernetes clusters grow exponentially, it can be increasingly challenging to manage and create consistency across an organization, leading to

ungoverned cluster sprawl. Without centralized governance, observability, and standardization around where and how resources are used, an enterprise is exposing itself to potential cybersecurity risks, significant overhead costs, and redundant efforts—not to mention magnifying the overall complexity of its IT infrastructure. Chapter 11, *Policy and Governance for Your Cluster*, provides a detailed discussion of the governance challenges rapid growth can create and some of the steps you can take in response to them.

D2iQ is excited to partner with O'Reilly to offer this excerpt of *Kubernetes Best Practices* because it provides the guidance and tools you need to successfully build and manage applications on Kubernetes, as well as empowering IT administrators and developers to standardize deployments, increase the availability of their applications, and streamline maintenance operations.

— Deepak Goel
Chief Technology Officer, D2iQ

CHAPTER 2

Developer Workflows

Kubernetes was built for reliably operating software. It simplifies deploying and managing applications with an application-oriented API, self-healing properties, and useful tools like Deployments for zero downtime rollout of software. Although all of these tools are useful, they don't do much to make it easier to develop applications for Kubernetes. Furthermore, even though many clusters are designed to run production applications and thus are rarely accessed by developer workflows, it is also critical to enable development workflows to target Kubernetes, and this typically means having a cluster or at least part of a cluster that is intended for development. Setting up such a cluster to facilitate easy development of applications for Kubernetes is a critical part of ensuring success with Kubernetes. Clearly if there is no code being built for your cluster, the cluster itself isn't accomplishing much.

Goals

Before we describe the best practices for building out development clusters, it is worth stating our goals for such clusters. Obviously, the ultimate goal is to enable developers to rapidly and easily build applications on Kubernetes, but what does that really mean in practice and how is that reflected in practical features of the development cluster?

It is useful to identify phases of developer interaction with the cluster.

The first phase is *onboarding*. This is when a new developer joins the team. This phase includes giving the user a login to the cluster as well as getting them oriented to their first deployment. The goal for this phase is to get a developer's feet wet in a minimal amount of time. You should set a key performance indicator (KPI) goal for this process. A reasonable goal would be that a user could go from nothing to the current

application at HEAD running in less than half an hour. Every time someone is new to the team, test how you are doing against this goal.

The second phase is *developing*. This is the day-to-day activity of the developer. The goal for this phase is to ensure rapid iteration and debugging. Developers need to quickly and repeatedly push code to the cluster. They also need to be able to easily test their code and debug it when it isn't operating properly. The KPI for this phase is more challenging to measure, but you can estimate it by measuring the time to get a pull request (PR) or change up and running in the cluster, or with surveys of the user's perceived productivity, or both. You will also be able to measure this in the overall productivity of your teams.

The third phase is *testing*. This phase is interleaved with developing and is used to validate the code before submission and merging. The goals for this phase are two-fold. First, the developer should be able to run all tests for their environment before a PR is submitted. Second, all tests should automatically run before code is merged into the repository. In addition to these goals you should also set a KPI for the length of time the tests take to run. As your project becomes more complex, it's natural for more and more tests to take a longer time. As this happens, it might become valuable to identify a smaller set of smoke tests that a developer can use for initial validation before submitting a PR. You should also have a very strict KPI around *test flakiness*. A flaky test is one that occasionally (or not so occasionally) fails. In any reasonably active project, a flakiness rate of more than one failure per one thousand runs will lead to developer friction. You need to ensure that your cluster environment does not lead to flaky tests. Whereas sometimes flaky tests occur due to problems in the code, they can also occur because of interference in the development environment (e.g., running out of resources and noisy neighbors). You should ensure that your development environment is free of such issues by measuring test flakiness and acting quickly to fix it.

Building a Development Cluster

When people begin to think about developing on Kubernetes, one of the first choices that occurs is whether to build a single large development cluster or to have one cluster per developer. Note that this choice only makes sense in an environment in which dynamic cluster creation is easy, such as the public cloud. In physical environments, its possible that one large cluster is the only choice.

If you do have a choice you should consider the pros and cons of each option. If you choose to have a development cluster per user, the significant downside of this approach is that it will be more expensive and less efficient, and you will have a large number of different development clusters to manage. The extra costs come from the fact that each cluster is likely to be heavily underutilized. Also, with developers creating different clusters, it becomes more difficult to track and garbage-collect resources

that are no longer in use. The advantage of the cluster-per-user approach is simplicity: each developer can self-service manage their own cluster, and from isolation, it's much more difficult for different developers to step on one another's toes.

On the other hand, a single development cluster will be significantly more efficient; you can likely sustain the same number of developers on a shared cluster for one-third the price (or less). Plus, it's much easier for you to install shared cluster services, for example, monitoring and logging, which makes it significantly easier to produce a developer-friendly cluster. The downside of a shared development cluster is the process of user management and potential interference between developers. Because the process of adding new users and namespaces to the Kubernetes cluster isn't currently streamlined, you will need to activate a process to onboard new developers. Although Kubernetes resource management and Role-Based Access Control (RBAC) can reduce the probability that two developers conflict, it is always possible that a user will *brick* the development cluster by consuming too many resources so that other applications and developers won't schedule. Additionally, you will still need to ensure that developers don't leak and forget about resources they've created. This is somewhat easier, though, than the approach in which developers each create their own clusters.

Even though both approaches are feasible, generally, our recommendation is to have a single large cluster for all developers. Although there are challenges in interference between developers, they can be managed and ultimately the cost efficiency and ability to easily add organization-wide capabilities to the cluster outweigh the risks of interference. But you will need to invest in a process for onboarding developers, resource management, and garbage collection. Our recommendation would be to try a single large cluster as a first option. As your organization grows (or if it is already large), you might consider having a cluster per team or group (10 to 20 people) rather than a giant cluster for hundreds of users. This can make both billing and management easier.

Setting Up a Shared Cluster for Multiple Developers

When setting up a large cluster, the primary goal is to ensure that multiple users can simultaneously use the cluster without stepping on one another's toes. The obvious way to separate your different developers is with Kubernetes namespaces. Namespaces can serve as scopes for the deployment of services so that one user's frontend service doesn't interfere with another user's frontend service. Namespaces are also scopes for RBAC, ensuring that one developer cannot accidentally delete another developer's work. Thus, in a shared cluster it makes sense to use a namespace as a developer's workspace. The processes for onboarding users and creating and securing a namespace are described in the following sections.

Onboarding Users

Before you can assign a user to a namespace, you have to onboard that user to the Kubernetes cluster itself. To achieve this, there are two options. You can use certificate-based authentication to create a new certificate for the user and give them a *kubeconfig* file that they can use to log in, or you can configure your cluster to use an external identity system (for example, Microsoft Azure Active Directory or AWS Identity and Access Management [IAM]) for cluster access.

In general, using an external identity system is a best practice because it doesn't require that you maintain two different sources of identity, but in some cases this isn't possible and you need to use certificates. Fortunately, you can use the Kubernetes certificate API for creating and managing such certificates. Here's the process for adding a new user to an existing cluster.

First, you need to generate a certificate signing request to generate a new certificate. Here is a simple Go program to do this:

```
package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "crypto/x509/pkix"
        "encoding/asn1"
        "encoding/pem"
        "os"
)

func main() {
        name := os.Args[1]
        user := os.Args[2]

        key, err := rsa.GenerateKey(rand.Reader, 1024)
        if err != nil {
                panic(err)
        }
        keyDer := x509.MarshalPKCS1PrivateKey(key)
        keyBlock := pem.Block{
                Type:  "RSA PRIVATE KEY",
                Bytes: keyDer,
        }
        keyFile, err := os.Create(name + "-key.pem")
        if err != nil {
                panic(err)
        }
        pem.Encode(keyFile, &keyBlock)
        keyFile.Close()
```

```
        commonName := user
        // You may want to update these too
        emailAddress := "someone@myco.com"

        org := "My Co, Inc."
        orgUnit := "Widget Farmers"
        city := "Seattle"
        state := "WA"
        country := "US"

        subject := pkix.Name{
                CommonName:         commonName,
                Country:            []string{country},
                Locality:           []string{city},
                Organization:       []string{org},
                OrganizationalUnit: []string{orgUnit},
                Province:           []string{state},
        }

        asn1, err := asn1.Marshal(subject.ToRDNSequence())
        if err != nil {
                panic(err)
        }
        csr := x509.CertificateRequest{
                RawSubject:         asn1,
                EmailAddresses:     []string{emailAddress},
                SignatureAlgorithm: x509.SHA256WithRSA,
        }

        bytes, err := x509.CreateCertificateRequest(rand.Reader, &csr, key)
        if err != nil {
                panic(err)
        }
        csrFile, err := os.Create(name + ".csr")
        if err != nil {
                panic(err)
        }

        pem.Encode(csrFile, &pem.Block{Type: "CERTIFICATE REQUEST", Bytes:
bytes})
        csrFile.Close()
}
```

You can run this as follows:

```
go run csr-gen.go client <user-name>;
```

This creates files called *client-key.pem* and *client.csr*. You then can run the following script to create and download a new certificate:

```
#!/bin/bash

csr_name="my-client-csr"
```

```
name="${1:-my-user}"

csr="${2}"


cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: ${csr_name}
spec:
  groups:
  - system:authenticated
  request: $(cat ${csr} | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - client auth
EOF

echo
echo "Approving signing request."
kubectl certificate approve ${csr_name}

echo
echo "Downloading certificate."
kubectl get csr ${csr_name} -o jsonpath='{.status.certificate}' \
        | base64 --decode > $(basename ${csr} .csr).crt

echo
echo "Cleaning up"
kubectl delete csr ${csr_name}

echo
echo "Add the following to the 'users' list in your kubeconfig file:"
echo "- name: ${name}"
echo "  user:"
echo "    client-certificate: ${PWD}/$(basename ${csr} .csr).crt"
echo "    client-key: ${PWD}/$(basename ${csr} .csr)-key.pem"
echo
echo "Next you may want to add a role-binding for this user."
```

This script prints out the final information that you can add to a *kubeconfig* file to enable that user. Of course, the user has no access privileges, so you will need to apply Kubernetes RBAC for the user in order to grant them privileges to a namespace.

Creating and Securing a Namespace

The first step in provisioning a namespace is actually just creating it. You can do this using **`kubectl create namespace my-namespace`**.

But the truth is that when you create a namespace, you want to attach a bunch of metadata to that namespace, for example, the contact information for the team that builds the component deployed into the namespace. Generally, this is in the form of annotations; you can either generate the YAML file using some templating, such as Jinja or others, or you can create and then annotate the namespace. A simple script to do this looks like:

```
ns='my-namespace'
kubectl create namespace ${ns}
kubectl annotate namespace ${ns} annotation_key=annotation_value
```

When the namespace is created, you want to secure it by ensuring that you can grant access to the namespace to a specific user. To do this, you can bind a role to a user in the context of that namespace. You do this by creating a `RoleBinding` object within the namespace itself. The `RoleBinding` might look like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: example
  namespace: my-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: edit
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: myuser
```

To create it, you simply run `kubectl create -f role-binding.yaml`. Note that you can reuse this binding as much as you want so long as you update the namespace in the binding to point to the correct namespace. If you ensure that the user doesn't have any other role bindings, you can be assured that this namespace is the only part of the cluster to which the user has access. A reasonable practice is to also grant reader access to the entire cluster; in this way developers can see what others are doing in case it is interfering with their work. Be careful in granting such read access, however, because it will include access to secret resources in the cluster. Generally, in a development cluster this is OK because everyone is in the same organization and the secrets are used only for development; however, if this is a concern, then you can create a more fine-grained role that eliminates the ability to read secrets.

If you want to limit the amount of resources consumed by a particular namespace, you can use the ResourceQuota resource to set a limit to the total number of resources that any particular namespace consumes. For example, the following quota limits the namespace to 10 cores and 100 GB of memory for both Request and Limit for the pods in the namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: limit-compute
  namespace: my-namespace
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 100Gi
    limits.cpu: "10"
    limits.memory: 100Gi
```

Managing Namespaces

Now that you have seen how to onboard a new user and how to create a namespace to use as a workspace, the question remains how to assign a developer to the namespace. As with many things, there is no single perfect answer; rather, there are two approaches. The first is to give each user their own namespace as part of the onboarding process. This is useful because after a user is onboarded, they always have a dedicated workspace in which they can develop and manage their applications. However, making the developer's namespace too persistent encourages the developer to leave things lying around in the namespace after they are done with them, and garbage-collecting and accounting individual resources is more complicated. An alternate approach is to temporarily create and assign a namespace with a bounded time to live (TTL). This ensures that the developer thinks of the resources in the cluster as transient and that it is easy to build automation around the deletion of entire namespaces when their TTL has expired.

In this model, when the developer wants to begin a new project, they use a tool to allocate a new namespace for the project. When they create the namespace, it has a selection of metadata associated with the namespace for management and accounting. Obviously, this metadata includes the TTL for the namespace, but it also includes the developer to which it is assigned, the resources that should be allocated to the namespace (e.g., CPU and memory), and the team and purpose of the namespace. This metadata ensures that you can both track resource usage and delete the namespace at the right time.

Developing the tooling to allocate namespaces on demand can seem like a challenge, but simple tooling is relatively simple to develop. For example, you can achieve the

allocation of a new namespace with a simple script that creates the namespace and prompts for the relevant metadata to attach to the namespace.

If you want to get more integrated with Kubernetes, you can use custom resource definitions (CRDs) to enable users to dynamically create and allocate new namespaces using the `kubectl` tool. If you have the time and inclination, this is definitely a good practice because it makes namespace management declarative and also enables the use of Kubernetes RBAC.

After you have tooling to enable the allocation of namespaces, you also need to add tooling to reap namespaces when their TTL has expired. Again, you can accomplish this with a simple script that examines the namespaces and deletes those that have an expired TTL.

You can build this script into a container and use a `ScheduledJob` to run it at an interval like once per hour. Combined together, these tools can ensure that developers can easily allocate independent resources for their project as needed, but those resources will also be reaped at the proper interval to ensure that you don't have wasted resources and that old resources don't get in the way of new development.

Cluster-Level Services

In addition to tooling to allocate and manage namespaces, there are also useful cluster-level services, and it's a good idea to enable them in your development cluster. The first is log aggregation to a central Logging as a Service (LaaS) system. One of the easiest things for a developer to do to understand the operation of their application is to write something to STDOUT. Although you can access these logs via `kubectl logs`, that log is limited in length and is not particularly searchable. If you instead automatically ship those logs to a LaaS system such as a cloud service or an Elasticsearch cluster, developers can easily search through logs for relevant information as well as aggregate logging information across multiple containers in their service.

Enabling Developer Workflows

Now that we succesfully have a shared cluster setup and we can onboard new application developers to the cluster itself, we need to actually get them developing their application. Remember that one of the key KPIs that we are measuring is the time from onboarding to an initial application running in the cluster. It's clear that via the just-described onboarding scripts we can quickly authenticate a user to a cluster and allocate a namespace, but what about getting started with the application? Unfortunately, even though there are a few techniques that help with this process, it generally requires more convention than automation to get the initial application up and running. In the following sections, we describe one approach to achieving this; it is by no

means the only approach or the only solution. You can optionally apply the approach as is or be inspired by the ideas to arrive at your own solution.

Initial Setup

One of the main challenges to deploying an application is the installation of all of the dependencies. In many cases, especially in modern microservice architectures, to even get started developing on one of the microservices requires the deployment of multiple dependencies, either databases or other microservices. Although the deployment of the application itself is relatively straightforward, the task of identifying and deploying all of the dependencies to build the complete application is often a frustrating case of trial and error married with incomplete or out-of-date instructions.

To address this issue, it is often valuable to introduce a convention for describing and installing dependencies. This can be seen as the equivalent of something like `npm install`, which installs all of the required JavaScript dependencies. Eventually, there is likely to be a tool similar to `npm` that provides this service for Kubernetes-based applications, but until then, the best practice is to rely on convention within your team.

One such option for a convention is the creation of a *setup.sh* script within the root directory of all project repositories. The responsibility of this script is to create all dependencies within a particular namespace to ensure that all of the application's dependencies are correctly created. For example, a setup script might look like the following:

```
kubectl create my-service/database-stateful-set-yaml
kubectl create my-service/middle-tier.yaml
kubectl create my-service/configs.yaml
```

You then could integrate this script with `npm` by adding the following to your *package.json*:

```
{
    ...
    "scripts": {
        "setup": "./setup.sh",
        ...
    }
}
```

With this setup, a new developer can simply run `npm run setup` and the cluster dependencies will be installed. Obviously, this particular integration is Node.js/npm specific. In other programming languages, it will make more sense to integrate with the language-specific tooling. For example, in Java you might integrate with a Maven *pom.xml* file instead.

Enabling Active Development

Having set up the developer workspace with required dependencies, the next task is to enable them to iterate on their application quickly. The first prerequisite for this is the ability to build and push a container image. Let's assume that you have this already set up; if not, you can read how to do this in a number of other online resources and books.

After you have built and pushed a container image, the task is to roll it out to the cluster. Unlike traditional rollouts, in the case of developer iteration, maintaining availability is really not a concern. Thus, the easiest way to deploy new code is to simply delete the Deployment object associated with the previous Deployment and then create a new Deployment pointing to the newly built image. It is also possible to update an existing Deployment in place, but this will trigger the rollout logic in the Deployment resource. Although it is possible to configure a Deployment to roll out code quickly, doing so introduces a difference between the development environment and the production environment that can be dangerous or destabilizing. Imagine, for example, that you accidentally push the development configuration of the Deployment into production; you will suddenly and accidentally deploy new versions to production without appropriate testing and delays between phases of the rollout. Because of this risk and because there is an alternative, the best practice is to delete and recreate the Deployment.

Just like installing dependencies, it is also a good practice to make a script for performing this deployment. An example *deploy.sh* script might look like the following:

```
kubectl delete -f ./my-service/deployment.yaml
perl -pi -e 's/${old_version}/${new_version}/' ./my-service/deployment.yaml
kubectl create -f ./my-service/deployment.yaml
```

As before, you can integrate this with existing programming language tooling so that (for example) a developer can simply run `npm run deploy` to deploy their new code into the cluster.

Enabling Testing and Debugging

After a user has successfully deployed their development version of their application, they need to test it and, if there are problems, debug any issues with the application. This can also be a hurdle when developing in Kubernetes because it is not always clear how to interact with your cluster. The `kubectl` command line is a veritable Swiss army knife of tools to achieve this, from `kubectl logs` to `kubectl exec` and `kubectl port-forward`, but learning how to use all of the different options and achieving familiarity with the tool can take a considerable amount of experience. Furthermore, because the tool runs in the terminal, it often requires the composition of

multiple windows to simultaneously examine both the source code for the application and the running application itself.

To streamline the testing and debugging experience, Kubernetes tooling is increasingly being integrated into development environments, for example, the open source extension for Visual Studio (VS) Code for Kubernetes. The extension is easily installed for free from the VS Code marketplace. When installed, it automatically discovers any clusters that you already have in your *kubeconfig* file, and it provides a tree-view navigation pane for you to see the contents of your cluster at a glance.

In addition to being able to see your cluster state at a glance, the integration allows a developer to use the tools available via `kubectl` in an intuitive, discoverable way. From the tree view, if you right-click a Kubernetes pod, you can immediately use port forwarding to bring a network connection to the pod directly to the local machine. Likewise, you can access the logs for the pod or even get a terminal within the running container.

The integration of these commands with prototypical user interface expectations (e.g., right-click shows a context menu), as well as the integration of these experiences alongside the code for the application itself, enable developers with minimal Kubernetes experience to rapidly become productive in the development cluster.

Of course this VS Code extension isn't the only integration between Kubernetes and a devlopment environment; there are several others that you can install depending on your choice of programming environment and style (`vi`, `emacs`, etc.).

Setting Up a Development Environment Best Practices

Setting up successful workflows on Kubernetes is key to productivity and happiness. Following these best practices will help to ensure that developers are up and running quickly:

- Think about developer experience in three phases: onboarding, developing, and testing. Make sure that the development environment you build supports all three of these phases.
- When building a development cluster, you can choose between one large cluster and a cluster per developer. There are pros and cons to each, but generally a single large cluster is a better approach.
- When you add users to a cluster, add them with their own identity and access to their own namespace. Use resource limits to restrict how much of the cluster they can use.
- When managing namespaces, think about how you can reap old, unused resources. Developers will have bad hygiene about deleting unused things. Use automation to clean it up for them.

- Think about cluster-level services like logs and monitoring that you can set up for all users. Sometimes, cluster-level dependencies like databases are also useful to set up on behalf of all users using templates like Helm charts.

Summary

We've reached a place where creating a Kubernetes cluster, especially in the cloud, is a relatively straightforward exercise, but enabling developers to productively use such a cluster is significantly less obvious and easy. When thinking about enabling developers to successfully build applications on Kubernetes, it's important to think about the key goals around onboarding, iterating, testing, and debugging applications. Likewise, it pays to invest in some basic tooling specific to user onboarding, namespace provisioning, and cluster services like basic log aggregation. Viewing a development cluster and your code repositories as an opportunity to standardize and apply best practices will ensure that you have happy and productive developers, successfully building code to deploy to your production Kubernetes clusters.

CHAPTER 11

Policy and Governance for Your Cluster

Have you ever wondered how you can ensure that all containers running on a cluster come only from an approved container registry? Or maybe you've been asked to ensure that services are never exposed to the internet. These are precisely the problems that policy and governance for your cluster set out to answer. As Kubernetes matures and becomes adopted by more and more enterprises, the question of policy and governance is becoming increasingly frequent. Although this area is still relatively new and upcoming, in this chapter we share what you can do to make sure that your cluster is in compliance with the defined policies of your enterprise.

Why Policy and Governance Are Important

Whether you operate in a highly regulated environment—for example, health care or financial services—or you simply want to make sure that you maintain a level of control over what's running on your clusters, you're going to need a way to implement the stated policies of the enterprise. After these policies are defined, you will need to determine how to implement policy and maintain clusters that are compliant to these policies. These policies might be in place to meet regulatory compliance or simply to enforce best practices. Whatever the reason, you must be sure that you do not sacrifice developer agility and self-service when implementing these policies.

How Is This Policy Different?

In Kubernetes, policy is everywhere. Whether it be network policy or pod security policy, we've all come to understand what policy is and when to use it. We trust that whatever is declared in Kubernetes resource specifications is implemented as per the policy definition. Both network policy and pod security policy are implemented at runtime. However, who manages the content that is actually defined in these

Kubernetes resource specifications? That's the job for policy and governance. Rather than implementing policy at runtime, when we talk about policy in the context of governance, what we mean is defining policy that controls the fields and values in the Kubernetes resource specifications themselves. Only Kubernetes resource specifications that are compliant against these policies are allowed and committed to the cluster state.

Cloud-Native Policy Engine

To be able to make decisions about what resources are compliant, we need a policy engine that is flexible enough to meet a variety of needs. The Open Policy Agent (OPA) is an open source, flexible, lightweight policy engine that has become increasingly popular in the cloud-native ecosystem. Having OPA in the ecosystem has allowed many implementations of different Kubernetes governance tools to appear. One such Kubernetes policy and governance project the community is rallying around is called Gatekeeper. For the rest of this chapter, we use Gatekeeper as the canonical example to illustrate how you might achieve policy and governance for your cluster. Although there are other implementations of policy and governance tools in the ecosystem, they all seek to provide the same user experience (UX) by allowing only compliant Kubernetes resource specifications to be committed to the cluster.

Introducing Gatekeeper

Gatekeeper is an open source customizable Kubernetes admission webhook for cluster policy and governance. Gatekeeper takes advantage of the OPA constraint framework to enforce custom resource definition (CRD)-based policies. Using CRDs allows for an integrated Kubernetes experience that decouples policy authoring from implementation. Policy templates are referred to as *constraint templates*, which can be shared and reused across clusters. Gatekeeper enables resource validation and audit functionality. One of the great things about Gatekeeper is that it's portable, which means that you can implement it on any Kubernetes clusters, and if you are already using OPA, you might be able to port that policy over to Gatekeeper.

Gatekeeper is still under active development and is subject to change. For the most recent updates on the project, visit the official upstream repository.

Example Policies

It's important not to become too stuck in the weeds and actually consider the problem that we are trying to solve. Let's take a look at some policies that solve some of the most common compliance issues for context:

- Services must not be exposed publicly on the internet.
- Allow containers only from trusted container registries.
- All containers must have resource limits.
- Ingress hostnames must not overlap.
- Ingresses must use only HTTPS.

Gatekeeper Terminology

Gatekeeper has adopted much of the same terminology as OPA. It's important that we cover what that terminology is so that you can understand how Gatekeeper operates. Gatekeeper uses the OPA constraint framework. Here, we introduce three new terms:

- Constraint
- Rego
- Constraint template

Constraint

The best way to think about constraints is as restrictions that you apply to specific fields and values of Kubernetes resource specifications. This is really just a long way of saying policy. This means that when constraints are defined, you are effectively stating that you *DO NOT* want to allow this. The implications of this approach mean that resources are implicitly allowed without a constraint that issues a deny. This is important because instead of allowing the Kubernetes resources specification fields and values you want, you are denying only the ones you do not want. This architectural decision suits Kubernetes resource specifications nicely because they are ever changing.

Rego

Rego is an OPA-native query language. Rego queries are assertions on the data stored in OPA. Gatekeeper stores rego in the constraint template.

Constraint template

You can think of this as a policy template. It's portable and reusable. Constraint templates consist of typed parameters and the target rego that is parameterized for reuse.

Defining Constraint Templates

Constraint templates are a Custom Resource Definition (CRD) that provide a means of templating policy so that it can be shared or reused. In addition, parameters for the policy can be validated. Let's take a look at a constraint template in the context of the earlier examples. In the following example, we share a constraint template that provides the policy "Only allow containers from trusted container registries":

```
apiVersion: templates.gatekeeper.sh/v1alpha1
kind: ConstraintTemplate
metadata:
  name: k8sallowedrepos
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRepos
        listKind: K8sAllowedReposList
        plural: k8sallowedrepos
        singular: k8sallowedrepos
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          properties:
            repos:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedrepos

        deny[{"msg": msg}] {
          container := input.review.object.spec.containers[_]
          satisfied := [good | repo = input.constraint.spec.parame
ters.repos[_] ; good = startswith(container.image, repo)]
          not any(satisfied)
          msg := sprintf("container <%v> has an invalid image repo <%v>,
allowed repos are %v", [container.name, container.image, input.con
straint.spec.parameters.repos])
        }
```

The constraint template consists of three main components:

Kubernetes-required CRD metadata
: The name is the most important part. We reference this later.

Schema for input parameters
: Indicated by the validation field, this section defines the input parameters and their associated types. In this example, we have a single parameter called `repo` that is an array of strings.

Policy definition
: Indicated by the `target` field, this section contains templated rego (the language to define policy in OPA). Using a constraint template allows the templated rego to be reused and means that generic policy can be shared. If the rule matches, the constraint is violated.

Defining Constraints

To use the previous constraint template, we must create a constraint resource. The purpose of the constraint resource is to provide the necessary parameters to the constraint template that we created earlier. You can see that the `kind` of the resource defined in the following example is `K8sAllowedRepos`, which maps to the constraint template defined in the previous section:

```
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  name: prod-repo-is-openpolicyagent
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "production"
  parameters:
    repos:
      - "openpolicyagent"
```

The constraint consists of two main sections:

Kubernetes metadata
: Notice that this constraint is of `kind K8sAllowedRepos`, which matches the name of the constraint template.

The spec
: The `match` field defines the scope of intent for the policy. In this example, we are matching pods only in the production namespace.

  The parameters define the intent for the policy. Notice that they match the type from the constraint template schema from the previous section. In this case, we allow only container images that start with `openpolicyagent`.

Constraints have the following operational characteristics:

- Logically AND-ed together
  - When multiple policies validate the same field, if one violates then the whole request is rejected
- Schema validation that allows early error detection
- Selection criteria
  - Can use label selectors
  - Constrain only certain kinds
  - Constrain only in certain namespaces

Data Replication

In some cases, you might want to compare the current resource against other resources that are in the cluster, for example, in the case of "Ingress hostnames must not overlap." OPA needs to have all of the other Ingress resources in its cache in order to evaluate the rule. Gatekeeper uses a `config` resource to manage which data is cached in OPA in order to perform evaluations such as the one previously mentioned. In addition, `config` resources are also used in the audit functionality, which we explore a bit later on.

The following example `config` resource caches v1 service, pods, and namespaces:

```
apiVersion: config.gatekeeper.sh/v1alpha1
kind: Config
metadata:
name: config
  namespace: gatekeeper-system
spec:
  sync:
    syncOnly:
    - kind: Service
      version: v1
    - kind: Pod
      version: v1
    - kind: Namespace
      version: v1
```

UX

Gatekeeper enables real-time feedback to cluster users for resources that violate defined policy. If we consider the example from the previous sections, we allow containers only from repositories that start with `openpolicyagent`.

Let's try to create the following resource; it is not compliant given the current policy:

```
apiVersion: v1
kind: Pod
metadata:
  name: opa
  namespace: production
spec:
  containers:
    - name: opa
      image: quay.io/opa:0.9.2
```

This gives you the violation message that's defined in the constraint template:

```
$ kubectl create -f bad_resources/opa_wrong_repo.yaml
Error from server (container <opa> has an invalid image repo <quay.io/opa:
0.9.2>, allowed repos are ["openpolicyagent"]): error when creating "bad_resour
ces/opa_wrong_repo.yaml": admission webhook "validation.gatekeeper.sh" denied
the request: container <opa> has an invalid image repo <quay.io/opa:0.9.2>,
allowed repos are ["openpolicyagent"]
```

Audit

Thus far, we have discussed only how to define policy and have it enforced as part of the request admission process. How do you handle a cluster that already has resources deployed where you want to know what is in compliance with the defined policy? That is exactly what audit sets out to achieve. When using audit, Gatekeeper periodically evaluates resources against the defined constraints. This helps with the detection of misconfigured resources according to policy and allows for remediation. The audit results are stored in the status field of the constraint, making them easy to find by simply using `kubectl`. To use audit, the resources to be audited must be replicated. For more details, refer to "Data Replication" on page 20.

Let's take a look at the constraint called `prod-repo-is-openpolicyagent` that you defined in the previous section:

```
$ kubectl get k8sallowedrepos prod-repo-is-openpolicyagent -o yaml
apiVersion: constraints.gatekeeper.sh/v1alpha1
kind: K8sAllowedRepos
metadata:
  creationTimestamp: "2019-06-04T06:05:05Z"
  finalizers:
  - finalizers.gatekeeper.sh/constraint
  generation: 2820
  name: prod-repo-is-openpolicyagent
  resourceVersion: "4075433"
  selfLink: /apis/constraints.gatekeeper.sh/v1alpha1/k8sallowedrepos/prod-repo-
is-openpolicyagent
  uid: b291e054-868e-11e9-868d-000d3afdb27e
spec:
  match:
    kinds:
```

```
    - apiGroups:
      - ""
      kinds:
      - Pod
    namespaces:
    - production
  parameters:
    repos:
    - openpolicyagent
status:
  auditTimestamp: "2019-06-05T05:51:16Z"
  enforced: true
  violations:
  - kind: Pod
    message: container <nginx> has an invalid image repo <nginx>, allowed repos
are
      ["openpolicyagent"]
    name: nginx
    namespace: production
```

Upon inspection, you can see the last time the audit ran in the `auditTimestamp` field. We also see all of the resources that violate this constraint under the `violations` field.

Becoming Familiar with Gatekeeper

The Gatekeeper repository ships with fantastic demonstration content that walks you through a detailed example of building policies to meet compliance for a bank. We would strongly recommend walking through the demonstration for a hands-on approach to how Gatekeeper operates. You can find the demonstration in this Git repository.

Gatekeeper Next Steps

The Gatekeeper project is continuing to grow and is looking to solve other problems in the areas of policy and governance, which includes features like these:

- Mutation (modifying resources based on policy; for example, add these labels)
- External data sources (integration with Lightweight Directory Access Protocol [LDAP] or Active Directory for policy lookup)
- Authorization (using Gatekeeper as a Kubernetes authorization module)
- Dry run (allow users to test policy before making it active in a cluster)

If these sound like interesting problems that you might be willing to help solve, the Gatekeeper community is always looking for new users and contributors to help shape the future of the project. If you would like to learn more, head over to the upstream repository on GitHub.

Policy and Governance Best Practices

You should consider the following best practices when implementing policy and governance on your clusters:

- If you want to enforce a specific field in a pod, you need to make a determination of which Kubernetes resource specification you want to inspect and enforce. Let's consider the case of Deployments, for example. Deployments manage ReplicaSets, which manage pods. We could enforce at all three levels, but the best choice is the one that is the lowest handoff point before the runtime, which in this case is the pod. This decision, however, has implications. The user-friendly error message when we try to deploy a noncompliant pod, as seen in "UX" on page 20, is not going to be displayed. This is because the user is not creating the noncompliant resource, the ReplicaSet is. This experience means that the user would need to determine that the resource is not compliant by running a `kubectl describe` on the current ReplicaSet associated with the Deployment. Although this might seem cumbersome, this is consistent behavior with other Kubernetes features, such as pod security policy.
- Constraints can be applied to Kubernetes resources on the following criteria: kinds, namespaces, and label selectors. We would strongly recommend scoping the constraint to the resources to which you want it to be applied as tightly as possible. This ensures consistent policy behavior as the resources on the cluster grow, and means that resources that don't need to be evaluated aren't being passed to OPA, which can result in other inefficiencies.
- Synchronizing and enforcing on potentially sensitive data such as Kubernetes secrets is *not* recommended. Given that OPA will hold this in its cache (if it is configured to replicate that data) and resources will be passed to Gatekeeper, it leaves surface area for a potential attack vector.
- If you have many constraints defined, a deny of constraint means that the entire request is denied. There is no way to make this function as a logical OR.

Summary

In this chapter, we covered why policy and governance are important and walked through a project that's built upon OPA, a cloud-native ecosystem policy engine, to provide a Kubernetes-native approach to policy and governance. You should now be prepared and confident the next time the security teams asks, "Are our clusters in compliance with our defined policy?"

CHAPTER /0

Managing Multiple Clusters

In this chapter, we discuss best practices for managing multiple Kubernetes clusters. We dive into the details of the differences between multicluster management and federation, tools to manage multiple clusters, and operational patterns for managing multiple clusters.

You might wonder why you would need multiple Kubernetes clusters; Kubernetes was built to consolidate many workloads to a single cluster, correct? This is true, but there are scenarios such as workloads across regions, concerns of blast radius, regulatory compliance, and specialized workloads.

We discuss these scenarios and explore the tools and techniques for managing multiple clusters in Kubernetes.

Why Multiple Clusters?

When adopting Kubernetes, you will likely have more than one cluster, and you might even start with more than one cluster to break out production from staging, user acceptance testing (UAT), or development. Kubernetes provides some multitenancy features with namespaces, which are a logical way to break up a cluster into smaller logical constructs. Namespaces allow you to define Role-Based Access Control (RBAC), quotas, pod security policies, and network policies to allow separation of workloads. This is a great way to separate out multiple teams and projects, but there are other concerns that might require you to build a multicluster architecture. Following are concerns to think about when deciding to use multicluster versus a single-cluster architecture:

- Blast radius
- Compliance

- Security
- Hard multitenancy
- Regional-based workloads
- Specialized workloads

When thinking through your architecture, *blast radius* should come front and center. This is one of the main concerns that we see with users designing for multicluster architectures. With microservice architectures we employ circuit breakers, retries, bulkheads, and rate limiting to constrain the extent of damage to our systems. You should design the same into your infrastructure layer, and multiple clusters can help with preventing the impact of cascading failures due to software issues. For example, if you have one cluster that serves 500 applications and you have a platform issue, it takes out 100% of the 500 applications. If you had a platform layer issue with 5 clusters serving those 500 applications, you affect only 20% of the applications. The downside to this is that now you need to manage five clusters, and your consolidation ratios will not be as good with a single cluster. Dan Woods wrote a great article about an actual cascading failure in a production Kubernetes environment. It is a great example of why you will want to consider multicluster architectures for larger environments.

Compliance is another area of concern for multicluster design because there are special considerations for Payment Card Industry (PCI), Health Insurance Portability and Accountability (HIPAA), and other workloads. It's not that Kubernetes doesn't provide some multitenant features, but these workloads might be easier to manage if they are segregated out from general purpose workloads. These compliant workloads might have specific requirements with respect to security hardening, nonshared components, or dedicated workload requirements. It's just much easier to separate these workloads than have to treat the cluster in such a specialized fashion.

Security in large Kubernetes clusters can become difficult to manage. As you start onboarding more and more teams to a Kubernetes cluster each team may have different security requirements and it can become very difficult to meet those needs in a large multi-tenant cluster. Even just managing RBAC, network policies, and pod security policies can become difficult at scale in a single cluster. A small change to a network policy can inadvertently open up security risk to other users of the cluster. With multiple clusters you can limit the security impact with a misconfiguration. If you decide that a larger Kubernetes cluster fits your requirements, then ensure that you have a very good operational process for making security changes and understand the blast radius of making a change to RBAC, network policy, and pod security policies.

Kubernetes doesn't provide *hard multitenancy* because it shares the same API boundary with all workloads running within the cluster. With namespacing this gives us

good soft multitenancy, but not enough to protect against hostile workloads within the cluster. Hard multitenancy is not a requirement for a lot of users; they trust the workloads that will be running within the cluster. Hard multitenancy is typically a requirement if you are a cloud provider, hosting Software as a Service (SaaS)-based software or untrusted workloads with untrusted user control.

When running workloads that need to serve traffic from in-region endpoints, your design will include multiple clusters that are based per region. When you have a globally distributed application, it becomes a requirement at that point to run multiple clusters. When you have workloads that need to be *regionally distributed*, it's a great use case for cluster federation of multiple clusters, which we dig into further later in this chapter.

Specialized workloads, such as high-performance computing (HPC), machine learning (ML), and grid computing, also need to be addressed in the multicluster architecture. These types of specialized workloads might require specific types of hardware, have unique performance profiles, and have specialized users of the clusters. We've seen this use case to be less prevalent in the design decision because having multiple Kubernetes node pools can help address specialized hardware and performance profiles. When you have the need for a very large cluster for an HPC or machine learning workload, you should take into consideration just dedicating clusters for these workloads.

With multicluster, you get isolation for "free," but it also has design concerns that you need to address at the outset.

Multicluster Design Concerns

When choosing a multicluster design there are some challenges that you'll run into. Some of these challenges might deter you from attempting a multicluster design given that the design might overcomplicate your architecture. Some of the common challenges we find users running into are:

- Data replication
- Service discovery
- Network routing
- Operational management
- Continuous deployment

Data replication and consistency has always been the crux of deploying workloads across geographical regions and multiple clusters. When running these services, you need to decide what runs where and develop a replication strategy. Most databases have built-in tools to perform the replication, but you need to design the application

to be able to handle the replication strategy. For NoSQL-type database services this can be easier because they can can handle scaling across multiple instances, but you still need to ensure that your application can handle eventual consistency across geographic regions or at least the latency across regions. Some cloud services, such as Google Cloud Spanner and Microsoft Azure CosmosDB, have built database services to help with the complications of handling data across multiple geographic regions.

Each Kubernetes cluster deploys its own *service discovery* registry, and registries are not synchronized across multiple clusters. This complicates applications being able to easily identify and discover one another. Tools such as HashiCorp's Consul can transparently synchronize services from multiple clusters and even services that reside outside of Kubernetes. There are other tools like Istio, Linkerd, and Cillium that are building on multiple cluster architectures to extend service discovery between clusters.

Kubernetes makes networking from within the cluster very easy, as it's a flat network and avoids using network address translation (NAT). If you need to route traffic in and out of the cluster, this becomes more complicated. Ingress into the cluster is implemented as a 1:1 mapping of ingress to the cluster because it doesn't support multicluster topologies with the Ingress resource. You'll also need to consider the egress traffic between clusters and how to route that traffic. When your applications reside within a single cluster this is easy, but when introducing multicluster, you need to think about the latency of extra hops for services that have application dependencies in another cluster. For applications that have tightly coupled dependencies, you should consider running these services within the same cluster to remove latency and extra complexity.

One of the biggest overheads to managing multiclusters is the *operational management*. Instead of one or a couple of clusters to manage and keep consistent, you might now have many clusters to manage in your environment. One of the most important aspects to managing multiclusters is ensuring that you have good automation practices in place because this will help to reduce the operational burden. When automating your clusters, you need to take into account the infrastructure deployment and managing add-on features to your clusters. For managing the infrastructure, using a tool like HashioCrp's Terraform can help with deploying and managing a consistent state across your fleet of clusters.

Using an *Infrastructure as Code* (IaC) tool like Terraform will give you the benefit of providing a reproducible way to deploy your clusters. On the other hand, you also need to be able to consistently manage add-ons to the cluster, such as monitoring, logging, ingress, security, and other tools. Security is also an important aspect of operational management, and you must be able to maintain security policies, RBAC, and network policies across clusters. Later in this chapter, we dive deeper into the topic of maintaining consistent clusters with automation.

With multiple clusters and *Continuous Delivery* (CD), you now need to deal with multiple Kubernetes API endpoints versus a single API endpoint. This can cause challenges in the distribution of applications. You can easily manage multiple pipelines, but suppose that you have a hundred different pipelines to manage, which can make application distribution very difficult. With this in mind, you need to look at different approaches to managing this situation. We take a look at solutions to help manage this later in the chapter.

Managing Multiple Cluster Deployments

One of the first steps that you want to take when managing multicluster deployments is to use an IoC tool like Terraform to set up deployments. Other deployment tools, such as kubespray, kops, or other cloud provider–specific tools, are all valid choices but, most importantly, use a tool that allows you to source control your cluster deployment for repeatability.

Automation is key to successfully managing multiple clusters in your environment. You might not have everything automated on day one, but you should make it a priority to automate all aspects of your cluster deployments and operations.

An interesting project in development is the Kubernetes Cluster API. The Cluster API is a Kubernetes project to bring declarative, Kubernetes-style APIs to cluster creation, configuration, and management. It provides optional, additive functionality on top of core Kubernetes. The Cluster API provides a cluster-level configuration declared through a common API, which will give you the ability to easily automate and build tooling around cluster automation. As of this writing, the project is still in development, so make sure to keep an eye out for it as it matures.

Deployment and Management Patterns

Kubernetes operators were introduced as an implementation of the *Infrastructure as Software* concept. Using them allows you to abstract the deployment of applications and services in a Kubernetes cluster. For example, suppose that you want to standardize on Prometheus for monitoring your Kubernetes clusters. You would need to create and manage various objects (deployment, service, ingress, etc.) for each cluster and team. You would also need to maintain the fundamental configurations of Prometheus, such as versions, persistence, retention policies, and replicas. As you can imagine, the maintenance of such a solution could be difficult across a large number of clusters and teams.

Instead of dealing with so many objects and configurations, you could install the `prometheus-operator`. This extends the Kubernetes API, exposing multiple new object kinds called `Prometheus`, `ServiceMonitor`, `PrometheusRule`, and `AlertManager`, which allow you to specify all of the details of a Prometheus deployment using

just a few objects. You can use the `kubectl` tool to manage such objects, just as it manages any other Kubernetes API object.

Figure 12-1 shows the architecture of the `prometheus-operator`.

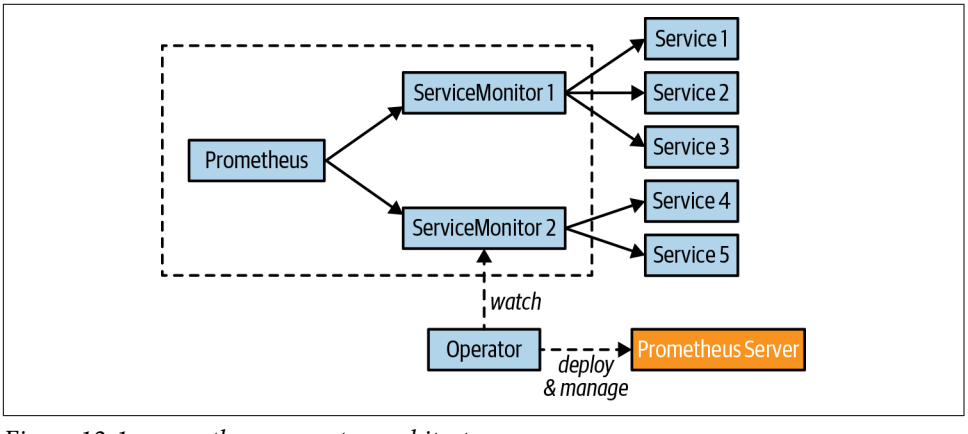


Figure 12-1. prometheus-operator architecture

Utilizing the *Operator* pattern for automating key operational tasks can help improve your overall cluster management capabilities. The Operator pattern was introduced by the CoreOS team in 2016 with the etcd operator and `prometheus-operator`. The Operator pattern builds on two concepts:

- Custom resource definitions
- Custom controllers

Custom resource definitions (CRDs) are objects that allow you to extend the Kubernetes API, based on your own API that you define.

Custom controllers are built on the core Kubernetes concepts of resources and controllers. Custom controllers allow you to build your own logic by watching events from Kubernetes API objects such as namespaces, Deployments, pods, or your own CRD. With custom controllers, you can build your CRDs in a declarative way. If you consider how the Kubernetes Deployment controller works in a reconciliation loop to always maintain the state of the deployment object to maintain its declarative state, this brings the same advantages of controllers to your CRDs.

When utilizing the Operator pattern, you can build in automation to operational tasks that need to be performed on operational tooling in multiclusters. Let's take the following Elasticsearch operator as an example. As in Chapter XX, we utilized the Elasticsearch, Logstash, and Kibana (ELK) stack to perform log aggregation of our cluster. The Elasticsearch operator can perform the following operations:

- Replicas for master, client, and data nodes
- Zones for highly available deployments
- Volume sizes for master and data nodes
- Resizing of cluster
- Snapshot for backups of the Elasticsearch cluster

As you can see, the operator provides automation for many tasks that you would need to perform when managing Elasticsearch, such as automating snapshots for backup and resizing the cluster. The beauty of this is that you manage all of this through familiar Kubernetes objects.

Think about how you can take advantage of different operators like the `prometheus-operator` in your environment and also how you can build your own custom operator to offload common operational tasks.

The GitOps Approach to Managing Clusters

GitOps was popularized by the folks at Weaveworks, and the idea and fundamentals were based on their experience of running Kubernetes in production. GitOps takes the concepts of the software development life cycle and applies them to operations. With GitOps, your Git repository becomes your source of truth, and your cluster is synchronized to the configured Git repository. For example, if you update a Kubernetes Deployment manifest, those configuration changes are automatically reflected in the cluster state.

By using this method, you can make it easier to maintain multiclusters that are consistent and avoid configuration drift across the fleet. GitOps allows you to declaratively describe your clusters for multiple environments and drives to maintain that state for the cluster. The practice of GitOps can apply to both application delivery and operations, but in this chapter, we focus on using it to manage clusters and operational tooling.

Weaveworks Flux was one of the first tools to enable the GitOps approach, and it's the tool we will use throughout the rest of the chapter. There are many new tools that have been released into the cloud-native ecosystem that are worth a look, such as Argo CD, from the folks at Intuit, which has also been widely adopted for the GitOps approach.

Figure 12-2 presents a representation of a GitOps workflow.

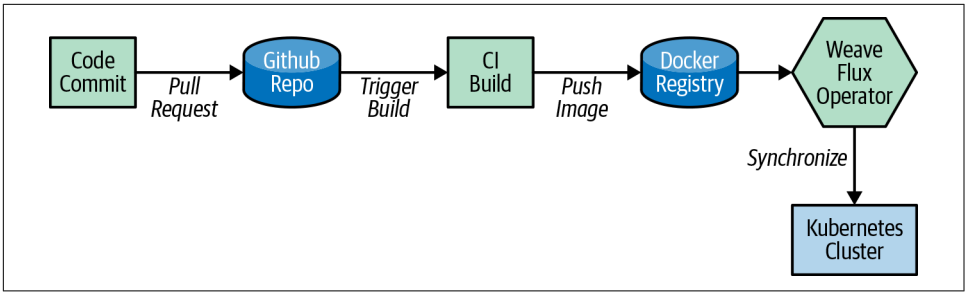


Figure 12-2. GitOps workflow

So, let's get Flux set up in your cluster and get a repository synchronized to the cluster:

```
git clone https://github.com/weaveworks/flux
cd flux
```

You now need to make a change to the Deployment manifest to configure it with your forked repo from Chapter XX. Modify the following line in the Deployment file to match your forked GitHub repository:

```
vim deploy/flux-deployment.yaml
```

Modify the following line with your Git repository:

```
--git-url=git@github.com:weaveworks/flux-get-started  (ex. --git-
url=git@github.com:your_repo/kbp )
```

Now, go ahead and deploy Flux to your cluster:

```
kubectl apply -f deploy
```

When Flux installs, it creates an SSH key so that it can authenticate with the Git repository. Use the Flux command-line tool to retrieve the SSH key so that you can configure access to your forked repository; first, you need to install `fluxctl`.

For MacOS:

```
brew install fluxctl
```

For Linux Snap Packages:

```
snap install fluxctl
```

For all other packages, you can find the latest binaries here:

```
fluxctl identity
```

Open GitHub, navigate to your fork, go to Setting > "Deploy keys," click "Add deploy key," give it a Title, select the "Allow write access" checkbox, paste the Flux public key, and then click "Add key." See the GitHub documentation for more information on how to manage deploy keys.

Now, if you view the Flux logs, you should see that it is synchronizing with your GitHub repository:

```
kubectl -n default logs deployment/flux -f
```

After you see that it's synchronizing with your GitHub repository, you should see that the Elasticsearch, Prometheus, Redis, and frontend pods are created:

```
kubectl get pods -w
```

With this example complete, you should be able to see how easy it is for you to synchronize your GitHub repository state with your Kubernetes cluster. This makes managing the multiple operational tools in your cluster much easier, because multiple clusters can synchronize with a single repository and remove the situation of having snowflake clusters.

Multicluster Management Tools

When working with multiple clusters, using Kubectl can immediately become confusing because you need to set different contexts to manage the different clusters. Two tools that you will want to install right away when dealing with multiple clusters are *kubectx* and *kubens*, which allow you to easily change between multiple contexts and namespaces.

When you need a full-fleged multicluster management tool, there are a few within the Kubernetes ecosystem to look at for managing multiple clusters. Following is a summary of some of the more popular tools:

- *Rancher* centrally manages multiple Kubernetes clusters in a centrally managed user interface (UI). It monitors, manages, backs up, and restores Kubernetes clusters across on-premises, cloud, and hosted Kubernetes setups. It also has tools for controlling applications deployed across multiple clusters and provides operational tooling.
- *KQueen* provides a multitenant self-service portal for Kubernetes cluster provisioning and focuses on auditing, visibility, and security of multiple Kubernetes clusters. KQueen is an open source project that was developed by the folks at Mirantis.
- *Gardener* takes a different approach to multicluster management in that it utilizes Kubernetes primitives to provide Kubernetes as a Service to your end users. It provides support for all major cloud vendors and was developed by the folks at SAP. This solution is really geared toward users who are building a Kubernetes as a Service offering.

Kubernetes Federation

Kubernetes first introduced Federation v1 in Kubernetes 1.3, and it has since been deprecated in lieu of Federation v2. Federation v1 set out to help with the distribution of applications to multiple clusters. Federation v1 was built utilizing the Kubernetes API and heavily relied on Kubernetes annotations, which imposed some problems in its design. The design was tightly coupled to the core Kubernetes API, which made Federation v1 quite monolithic in nature. At the time, the design decisions were probably not bad choices, but were built on the primitives that were available. The introducton of Kubernetes CRDs allowed a different way of thinking about how Federation could be designed.

Federation v2 (now called *KubeFed*) requires Kubernetes 1.11+ and is currently in alpha as of this writing. Federation v2 is built around the concept of CRDs and custom controllers, which allows you to extend Kubernetes with new APIs. Building around CRDs allows Federation to have new API types and not be restricted just to previous v1 deployment objects.

KubeFed is not necessarily about multicluster management, but providing high availability (HA) deployments across multiple clusters. It allows you to combine multiple clusters into a single management endpoint for delivering applications on Kubernetes. For example, if you have a cluster that resides in multiple public cloud environments, you can combine these clusters into a single control plane to manage deployments to all clusters to increase the resiliency of your application.

As of this writing, the following Federated resources are supported:

- Namespaces
- ConfigMaps
- Secrets
- Ingress
- Services
- Deployments
- ReplicaSets
- Horizontal Pod Autoscalers
- DaemonSets
- Jobs

To understand how this all works, let's first take a look at the architecture in Figure 12-3.

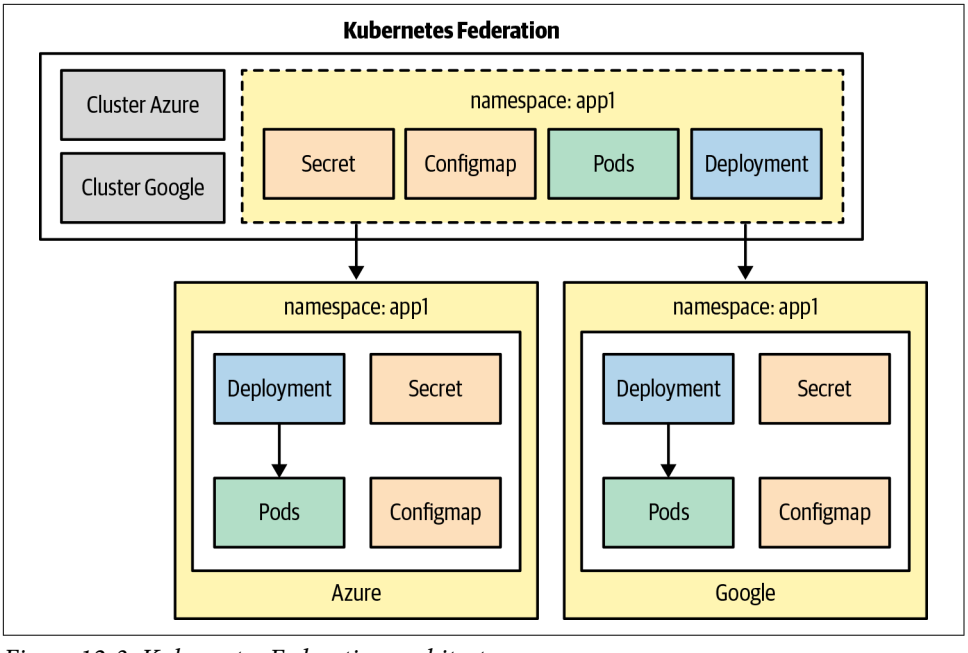


Figure 12-3. Kubernetes Federation architecture

It's important to understand that with Federation, not everything is just copied down to all clusters. For example, with Deployments and ReplicaSets, you define the number of replicas, which are then spread out across the clusters. This is the default for Deployments, but you can change the configuration. On the other hand, if you create a namespace, that namespace is cluster scoped and created in each cluster. Secrets, ConfigMaps, and DaemonSets work the same way and are copied down to each cluster. The Ingress resource is also different from the aforementioned objects because it creates a global multicluster resource with a single entry point into a service. As you can see from how KubeFed works, the use cases Kubefed supports are multiregion, multicloud, and global application deployments to Kubernetes.

Following is an example of a federated Deployment:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
metadata:
  name: test-deployment
  namespace: test-namespace
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 5
```

```
      selector:
        matchLabels:
          app: nginx
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
          - image: nginx
            name: nginx
  placement:
    clusters:
    - name: azure
    - name: google
```

This example creates a federated Deployment of an NGINX pod with five replicas, which are then spread across our clusters in Azure and another cluster in Google.

Setting up federated Kubernetes clusters is beyond the scope of this book, but you can learn more about the subject by referring to the KubeFed User Guide.

KubeFed is still in alpha, so keep an eye on it, but embrace the tools that you already have or can implement now so that you can be successful with Kubernetes HA and multicluster deployments.

Managing Multiple Clusters Best Practices

Consider the following best practices when managing multiple Kubernetes clusters:

- Limit the blast radius of your clusters to ensure cascading failures don't have a bigger impact on your applications.
- If you have regulatory concerns such as PCI, HIPPA, or HiTrust, think about utilizing multiclusters to ease the complexity of mixing these workloads with general workloads.
- If hard multitenancy is a business requirement, workloads should be deployed to a dedicated cluster.
- If multiple regions are needed for your applications, utilize a Global Load Balancer to manage traffic between clusters.
- You can break out specialized workloads such as HPC into their own individual clusters to ensure that the specialized needs for the workloads are met.
- If you're deploying workloads that will be spread across multiple regional datacenters, first ensure that there is a data replication strategy for the workload. Multiple clusters across regions can be easy, but replicating data across regions

can be complicated, so ensure that there is a sound strategy to handle asynchronous and synchronous workloads.

- Utilize Kubernetes operators like the `prometheus-operator` or Elasticsearch operator to handle automated operational tasks.
- When designing your multicluster strategy, also consider how you will do service discovery and networking between clusters. Service mesh tools like HashiCorp's Consul or Istio can help with networking across clusters.
- Be sure that your CD strategy can handle multiple rollouts between regions or multiple clusters.
- Investigate utilizing a GitOps approach to managing multiple cluster operational components to ensure consistency between all clusters in your fleet. The GitOps approach doesn't always work for everyone's environment, but you should at least investigate it to ease the operational burden of multicluster environments.

Summary

In this chapter, we discussed different strategies for managing multiple Kubernetes clusters. It's important to think about what your needs are at the outset and whether those needs match a multicluster topology. The first scenario to think about is whether you truly need *hard* multitenancy because this will automatically require a multicluster strategy. If you don't, consider your compliance needs and whether you have the operational capacity to consume the overhead of multicluster architectures. Finally, if you're going with more, smaller clusters, ensure that you put automation around the delivery and management of them to reduce the operational burden.

About the Authors

Brendan Burns is a distinguished engineer at Microsoft Azure and cofounder of the Kubernetes open source project. He's been building cloud applications for more than a decade.

Eddie Villalba is a software engineer with Microsoft's Commercial Software Engineering division, focusing on open source cloud and Kubernetes. He's helped many real-world users adopt Kubernetes for their applications.

Dave Strebel is a global cloud native architect at Microsoft Azure focusing on open source cloud and Kubernetes. He's deeply involved in the Kubernetes open source project, helping with the Kubernetes release team and leading SIG-Azure.

Lachlan Evenson is a principal program manager on the container compute team at Microsoft Azure. He's helped numerous people onboard to Kubernetes through both hands-on teaching and conference talks.

Colophon

The animal on the cover of *Kubernetes Best Practices* is an Old World mallard duck (*Anas platyrhynchos*), a kind of dabbling duck that feeds on the surface of water rather than diving for food. Species of Anas are typically separated by their ranges and behavioral cues; however, mallards frequently interbreed with other species, which has introduced some fully fertile hybrids.

Mallard ducklings are precocial and capable of swimming as soon as they hatch. Juveniles begin flying between three and four months of age. They reach full maturity at 14 months and have an average life expectancy of 3 years.

The mallard is a medium-sized duck that is just slightly heavier than most dabbling ducks. Adults average 23 inches long with a wingspan of 36 inches, and weigh 2.5 pounds. Ducklings have yellow and black plumage. At around six months of age, males and females can be distinguished visually as their coloring changes. Males have green head feathers, a white collar, purple-brown breast, gray-brown wings, and a yellowish-orange bill. Females are mottled brown, which is the color of most female dabbling ducks.

Mallards have a wide range of habitats across both northern and southern hemispheres. They are found in fresh- and salt-water wetlands, from lakes to rivers to seashores. Northern mallards are migratory, and winter father south. The mallard diet is highly variable, and includes plants, seeds, roots, gastropods, invertebrates, and crustaceans.

Brood parasites will target mallard nests. These are species of other birds who may lay their eggs in the mallard nest. If the eggs resemble those of the mallard, the mallard will accept them and raise the hatchlings with their own.

Mallards must contend with a wide variety of predators, most notably foxes and birds of prey such as falcons and eagles. They have also been preyed upon by catfish and pike. Crows, swans, and geese have all been known to attack the ducks over territorial disputes. Unihemispheric sleep (or sleeping with one eye open), which allows one hemisphere of the brain to sleep while the other is awake, was first noted in mallards. It is common among aquatic birds as a predation-avoidance behavior.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Jose Marzan, based on a black and white engraving from *The Animal World*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.