Book review for Cloud Native with Kubernetes by Alexander Raul, reviewed by Zihao Yu

Summary: A very “COMPLETE and WIDE” k8s book with many examples/tutorials on many Kubernetes “Parts/Add-ons”, e.g. Rook/Ceph, Falco, Jaeger, EFK, Prometheus, Grafana, Helm, Kustomize, AWS Codebuild, FluxCD, CRDs/Operators for cloud-controller-manager, cluster-autoscaler, Sidecar proxies (Nginx and Envoy), Istio, Serverless (Knative and OpenFaaS), Stateful workloads (Minio, Cockroach DB, RabbitMQ). It is very wide in terms of coverage. However, do not expect very DEEP in some of the topics due to pages limitations. It will be a good starting point to learn all the basics and know many of the important parts/add-ons commonly used in a production k8s env.

There are 4 sections. The first two are basics to me (CKA already), but will be good for any beginners. I like more in section 3 and 4 with their examples/detail steps. I read more seriously starting from chap 8 till the end.

I like the following topics:

1/ ch 8 – node/pod affinity and anti-affinity

2/ ch 9 – observability on k8s: Metrics/Logging/Tracing/Alerts. You will learn how to install/use Prometheus, Grafana, Jaeger (and its CRD operator) and alerts manager.

3/ ch 10 – troubleshooting k8s: good discussions/case studies, e.g. placement failures, service not responding, etc.…

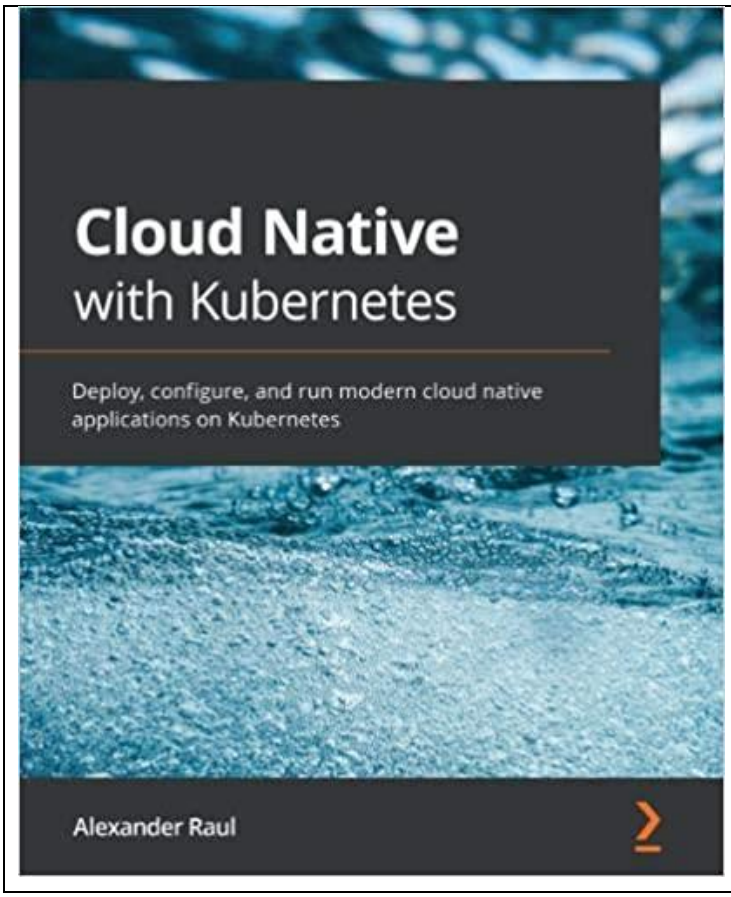
4/ ch 11 – CI/CD with Helm/Kustomize, in and out of cluster CI/CD, e.g. AWS Codebuild, FluxCD.

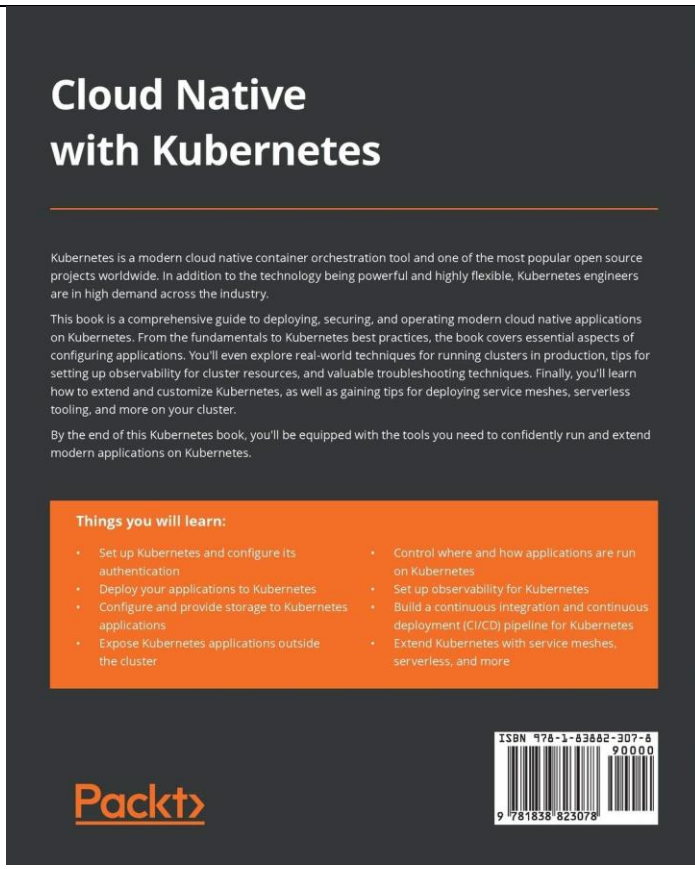
5/ ch 12 – k8s security and compliance: review CVEs, admission controllers, PSP, Network Policies (my favorite), Falco install/config/rules/use cases.

6/ ch 13 – extend k8s with CRDs: k8s operators, cloud-controller-manager, cluster-autoscaler (this is also called Vertical Auto Scaler), intro to CNCF.

7/ ch 14 – sidecar proxies, service meshes, serverless: Nginx/Envoy (important proxies!), Istio, Knative and OpenFaaS.

8/ ch 15 – Stateful workloads on k8s: Minio, Cockroach DB, RabbitMQ. It will help you understand the important points running stateful workloads and then you can apply the same concepts in others, e.g. Couchbase DB, MySQL, etc. in your env.





Book - https://www.amazon.com/Cloud-Native-Kubernetes-configure-applications/dp/1838823077/ref=sr_1_2 (kindle eBook $28)

https://www.packtpub.com/product/cloud-native-with-kubernetes/9781838823078 ($5 special sale now!)

Code - The code used in this chapter can be found in the book's GitHub repository at https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes

Section 1: chap 1-3

Basics, e.g. Set up k8s cluster, diff. component, Authentication and Authorization, e.g. RBAC, ABAC, kubectl, yaml, create clusters (easy and hard ways), setup storage, run/deploy containers/apps, etc.

1/ covers All 3 major clouds:

2
Setting Up Your Kubernetes Cluster

ALL 3 Major Clouds !

Section 2: chap 4-8

Chap 4 – Basics, e.g. replicasets, deployments, hpa, daemonsets, statefulsets, jobs

Chap 5- Services and Ingress, e.g. good diagram to explain ingress controller,

In our preceding YAML, the ingress has a singular `host` value, which would correspond to the host request header for traffic coming through the Ingress. Then, we have two paths, `/a` and `/b`, which lead to our two previously created `ClusterIP` Services. To put this configuration in a graphical format, let's take a look at the following diagram:

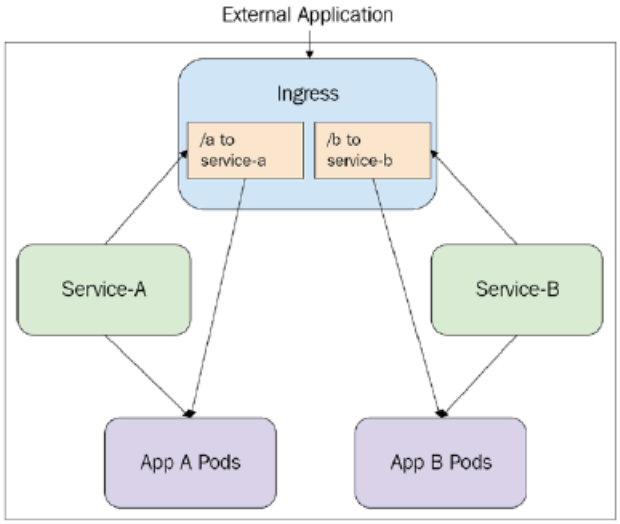


Figure 5.4 – Kubernetes Ingress example

As you can see, our simple path-based rules result in network requests getting routed directly to the proper Pods. This is because `nginx-ingress` uses the Service selector to get a list of Pod IPs, but does not directly use the Service to communicate with the Pods. Rather, the Nginx (in this case) config is automatically updated as new Pod IPs come online.

The `host` value isn't actually required. If you leave it out, any traffic that comes through the Ingress, regardless of the host header (unless it matches a different rule that specifies a host) will be routed according to the rule. The following YAML shows this:

ingress-no-host.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-first-ingress
```

Chap 6- Application configuration, e.g. configmaps, secrets, e.g. encrypted secrets at rest

Chap 7- Storages, e.g. PVs, PVCs, Storageclasses, e.g. AWS gp2Encrypted to match gp2 storage type with EBS encryption, Dynamic storage provisioning, On-Prem hostPath, Ceph with Rook, … etc.

Good demo on dynamic storage provisioning in AWS below:

This first config, `storageClassName`, represents the type of storage we want to use. For the `hostPath` volume type, we simply specify `manual`, but for AWS EBS, for instance, you could create and use a storage class called `gp2Encrypted` to match the `gp2` storage type in AWS with EBS encryption enabled. Storage classes are therefore combinations of configuration that are available for a particular volume type – which can be referenced in the volume spec.

Moving forward with our AWS `StorageClass` example, let's provision a new `StorageClass` for `gp2Encrypted`:

gp2-storageclass.yaml

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2Encrypted
  annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  encrypted: "true"
  fsType: ext4
```

Now, we can create our `PersistentVolume` using the `gp2Encrypte` storage class. However, there's a shortcut to creating `PersistentVolumes` using dynamically provisioned EBS (or other cloud) volumes. When using dynamically provisioned volumes, we create the `PersistentVolumeClaim` first, which then automatically generates the `PersistentVolume`.

Running `kubectl apply -f` on this file should result in a new, autogenerated **Persistent Volume** (**PV**) being created. If your AWS cloud provider is set up correctly, this will result in the creation of a new EBS volume with type GP2 and encryption enabled.

Before we attach our EBS-backed persistent volume to our Pod, let's confirm that the EBS volume was created correctly in AWS.

To do so, we can navigate to our AWS console and ensure we are in the same region that our EKS cluster is running in. Then go to **Services** > **EC2** and click on **Volumes** in the left menu under **Elastic Block Store**. In this section, we should see a line item with an autogenerated volume of the same size (**1 GiB**) as our PVC states. It should have the class of GP2, and it should have encryption enabled. Let's see what this would look like in the AWS console:

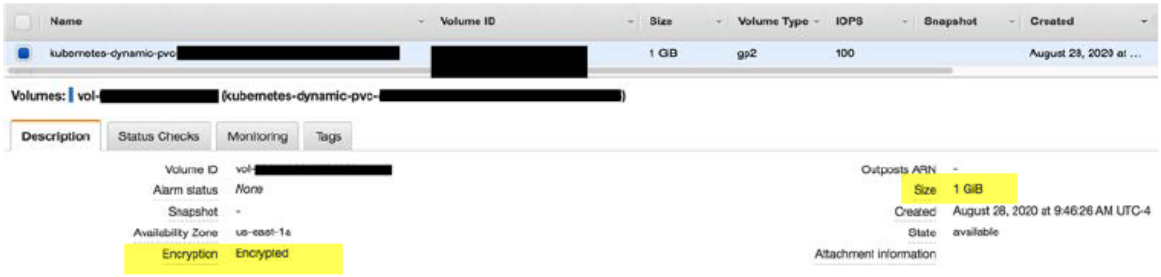


Figure 7.1 – AWS console with autocreated EBS volume

As you can see, we have our dynamically generated EBS volume properly created in AWS, with encryption enabled and the **gp2** volume type assigned. Now that we have our volume

Good demo on Rook and Ceph:

Rook is a popular open source Kubernetes storage abstraction layer. It can provide persistent volumes through a variety of providers, such as EdgeFS and NFS. In this case, we'll use Ceph, an open source storage project that provides object, block, and file storage. For simplicity, we'll use block mode.

Installing Rook on Kubernetes is actually pretty simple. We'll take you from installing Rook to setting up a Ceph cluster, to finally provisioning persistent volumes on our cluster.

Installing Rook

We're going to use a typical Rook installation default setup provided by the Rook GitHub repository. This could be highly customized depending on the use case but will allow us to quickly set up block storage for our workloads. Please refer to the following steps to do this:

1. First, let's clone the Rook repository:

```
> git clone --single-branch --branch master https://
github.com/rook/rook.git
> cd cluster/examples/kubernetes/ceph
```

2. Our next step is to create all the relevant Kubernetes resources, including several **Custom Resource Definitions** (**CRDs**). We'll talk about these in later chapters, but for now, consider them new Kubernetes resources that are specific to Rook, outside of the typical Pods, Services, and so on. To create common resources, run the following command:

```
> kubectl apply -f ./common.yaml
```

3. Next, let's start our Rook operator, which will handle provisioning all the necessary resources for a particular Rook provider, which in this case will be Ceph:

You can see the Ceph operator at

https://github.com/rook/rook/blob/master/cluster/examples/kubernetes/ceph/operator.yaml

Very detail Rook and Ceph demo:

The rook-ceph-block storage class

Now our cluster is working, we can create our storage class that will be used by our PVs. We will call this storage class `rook-ceph-block`. Here's our YAML file (`ceph-rook-combined.yaml`), which will include our `CephBlockPool` (which will handle our block storage in Ceph – see `https://rook.io/docs/rook/v0.9/ceph-pool-crd.html` for more information) as well as the storage class itself:

ceph-rook-combined.yaml

```
apiVersion: ceph.rook.io/v1
kind: CephBlockPool
metadata:
  name: replicapool
  namespace: rook-ceph
spec:
  failureDomain: host
  replicated:
    size: 3
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-block
provisioner: rook-ceph.rbd.csi.ceph.com
parameters:
    clusterID: rook-ceph
    pool: replicapool
    imageFormat: "2"
currently supports only `layering` feature.
    imageFeatures: layering
    csi.storage.k8s.io/provisioner-secret-name: rook-csi-rbd-
provisioner
    csi.storage.k8s.io/provisioner-secret-namespace: rook-ceph
    csi.storage.k8s.io/node-stage-secret-name: rook-csi-rbd-
node
    csi.storage.k8s.io/node-stage-secret-namespace: rook-ceph
csi-provisioner
```

Up to creating the Ceph cluster, all the previous steps apply. At this point, we need to create our filesystem. Let's use the following YAML file to create it:

rook-ceph-fs.yaml

```
apiVersion: ceph.rook.io/v1
kind: CephFilesystem
metadata:
  name: ceph-fs
  namespace: rook-ceph
spec:
  metadataPool:
    replicated:
      size: 2
  dataPools:
    - replicated:
        size: 2
  preservePoolsOnDelete: true
  metadataServer:
    activeCount: 1
    activeStandby: true
```

In this case, we're replicating metadata and data to at least two pools for reliability, as configured in the `metadataPool` and `dataPool` blocks. We are also preserving the pools on delete using the `preservePoolsOnDelete` key.

Next, let's create our new storage class specifically for Rook/Ceph filesystem storage. The following YAML does this:

Chap 8- Pod Placement Controls, e.g. use cases, node selectors, taints, tolerations, node affinity, inter-pod affinity and anti-affinity. Good topics, e.g. multi-tenancy, multiple failure domains, multiple taints and tolerations,

Code - https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/tree/master/Chapter8

Some default taints (which we'll discuss in the next section) that Kubernetes uses are as follows:

- `memory-pressure`
- `disk-pressure`
- `unreachable`
- `not-ready`
- `out-of-disk`
- `network-unavailable`
- `unschedulable`
- `uninitialized` (only for cloud-provider-created nodes)

These conditions can mark nodes as unable to receive new Pods, though there is some flexibility in how these taints are handled by the scheduler, as we will see later. The purpose of these system-created placement controls is to prevent unhealthy nodes from receiving workloads that may not function properly.

In addition to system-created placement controls for node health, there are several use cases where you, as a user, may want to implement fine-tuned scheduling, as we will see in the next section.

Multiple taints and tolerations

When there are multiple taints or tolerations on a Pod and node, the scheduler will check all of them. There is no `OR` logic operator here – if any of the taints on the node do not have a matching toleration on the Pod, it will not be scheduled on the node (with the exception of `PreferNoSchedule`, in which case, as before, the scheduler will try to not schedule on the node if possible). Even if out of six taints on the node, the Pod tolerates five of them, it will still not be scheduled for a `NoSchedule` taint, and it will still be evicted for a `NoExecute` taint.

For a tool that gives us a much more subtle way of controlling placement, let's look at node affinity.

Controlling Pods with node affinity

As you can probably tell, taints and tolerations – while much more flexible than node selectors – still leave some use cases unaddressed and in general only allow a *filter* pattern where you can match on a specific taint using `Exists` or `Equals`. There may be more advanced use cases where you want more flexible methods of selecting nodes – and *affinities* are a feature of Kubernetes that addresses this.

There are two types of affinity:

- **Node affinity**
- **Inter-Pod affinity**

Node affinity is a similar concept to node selectors except that it allows for a much more robust set of selection characteristics. Let's look at some example YAML and then pick apart the various pieces:

Good Example to show combined affinity and anti-affinity:

Combined affinity and anti-affinity

This is one of those situations where you can really put undue load on your cluster control plane. Combining Pod affinities with anti-affinities can allow incredibly nuanced rules that can be passed to the Kubernetes scheduler, which has the Herculean task of working to fulfill them.

Let's look at some YAML for a Deployment spec that combines these two concepts. Remember, affinity and anti-affinity are concepts that are applied to Pods – but we normally do not specify Pods without a controller like a Deployment or a ReplicaSet. Therefore, these rules are applied at the Pod spec level in the Deployment YAML. We are only showing the Pod spec part of this deployment for conciseness, but you can find the full file on the GitHub repository:

---

pod-with-both-antiaffinity-and-affinity.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hungry-app-deployment
# SECTION REMOVED FOR CONCISENESS
      spec:
       affinity:
         podAntiAffinity:
           preferredDuringSchedulingIgnoredDuringExecution:
           - labelSelector:
```

...

```
              matchExpressions:
              - key: app
                operator: In
                values:
                - other-hungry-app
            topologyKey: "rack"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - hungry-app-cache
            topologyKey: "rack"
      containers:
      - name: hungry-app
        image: hungry-app:latest
```

In this code block, we are telling the scheduler to treat the Pods in our Deployment as such: the Pod must be scheduled onto a node with a `rack` label such that it or any other node with a `rack` label and the same value has a Pod with `app=hungry-label-cache`.

Secondly, the scheduler must attempt to schedule the Pod, if possible, to a node with the `rack` label such that it or any other node with the `rack` label and the same value does not have a Pod with the `app=other-hungry-app` label running.

Pod affinity and anti-affinity limitations

The biggest restriction on affinity and anti-affinity is that you are not allowed to use a blank `topologyKey`. Without restricting what the scheduler treats as a single topology type, some very unintended behavior can happen.

The second limitation is that, by default, if you're using the hard version of anti-affinity – `requiredOnSchedulingIgnoredDuringExecution`, you cannot just use any label as a `topologyKey`.

Kubernetes will only let you use the `kubernetes.io/hostname` label, which essentially means that you can only have one topology per node if you're using `required` anti-affinity. This limitation does not exist for either the `prefer` anti-affinity or either of the affinities, even the `required` one. It is possible to change this functionality, but it requires writing a custom admission controller – which we will discuss in *Chapter 12, Kubernetes Security and Compliance*, and *Chapter 13, Extending Kubernetes with CRDs*.

So far, our work with placement controls has not discussed namespaces. However, with Pod affinities and anti-affinities, they do hold relevance.

Pod affinity and anti-affinity namespaces

Since Pod affinities and anti-affinities cause changes in behavior based on the location of other Pods, namespaces are a relevant piece to decide which Pods count for or against an affinity or anti-affinity.

By default, the scheduler will only look to the namespace in which the Pod with the affinity or anti-affinity was created. For all our previous examples, we haven't specified a namespace so the default namespace will be used.

If you want to add one or more namespaces in which Pods will affect the affinity or anti-affinity, you can do so using the following YAML:

https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/blob/master/Chapter8/pod-with-anti-affinity-namespace.yaml

Section 3: chap 9-12

*** Running k8s in PRODUCTION: Day 2 Operations, CI/CD, custom and extend k8s, observability, troubleshooting, Security and Compliance. Chap 12 has good reviews on CVEs, tools, e.g. admission controllers, PSP, Netpol, Falco, etc.…

9

Observability on Kubernetes

10
Troubleshooting Kubernetes

11
Template Code Generation and CI/CD on Kubernetes

12

Kubernetes Security and Compliance

Section 4: Extending Kubernetes

13

Extending Kubernetes with CRDs

Ch. 9: Observability on Kubernetes – good troubleshooting steps and metrics/logs/traces/alerts.

3. First, let's create the Elasticsearch cluster itself. This runs as a StatefulSet on Kubernetes, and also provides a Service. To create the cluster, we need to run two `kubectl` commands:

```
kubectl apply -f ./fluentd-elasticsearch/es-statefulset.
yaml
kubectl apply -f ./fluentd-elasticsearch/es-service.yaml
```

> **Important note**
>
> A word of warning for the Elasticsearch StatefulSet – by default, the resource request for each Pod is 3 GB of memory, so if none of your Nodes have that available, you will not be able to deploy it as configured by default.

p 208 - GOOD WARNING ! Need 3GB for elasticsearch SS !

4. Next, let's deploy the FluentD logging agents. These will run as a DaemonSet – one per Node – and forward logs from the Nodes to Elasticsearch. We also need to create the ConfigMap YAML, which contains the base FluentD agent configuration. This can be further customized to add things such as log filters and new sources.

Next Jaegar on p.214 for distributed tracing! Jaeger Operator code at https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/blob/master/Chapter9/jaeger-allinone.yaml

"Kind: Jaeger"

Installing Jaeger using the Jaeger Operator

To install Jaeger, we are going to use the Jaeger Operator, which is the first operator that we've come across in this book. An *operator* in Kubernetes is simply a pattern for creating custom application controllers that speak Kubernetes's language. This means that instead of having to deploy all the various Kubernetes resources for an application, you can deploy a single Pod (or usually, single Deployment) and that application will talk to Kubernetes and spin up all the other required resources for you. It can even go further and self-operate the application, making resource changes when necessary. Operators can be highly complex, but they make it easier for us as end users to deploy commercial or open source software on our Kubernetes clusters.

To get started with the Jaeger Operator, we need to create a few initial resources for Jaeger, and then the operator will do the rest. A prerequisite for this installation of Jaeger is that the `nginx-ingress` controller is installed on our cluster, since that is how we will access the Jaeger UI.

First, we need to create a namespace for Jaeger to live in. We can get this via the `kubectl create namespace` command:

```
kubectl create namespace observability
```

If the operator is running correctly, you will see something similar to the following output, with one available Pod for the deployment:

```
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
jaeger-operator   1         1         1            1           1m
```

Figure 9.20 – Jaeger Operator Pod output

We now have our Jaeger Operator up and running – but Jaeger itself isn't running. Why is this the case? Jaeger is a highly complex system and can run in different configurations, and the operator makes it easier to deploy these configurations.

The Jaeger Operator uses a CRD called `Jaeger` to read a configuration for your Jaeger instance, at which time the operator will deploy all the necessary Pods and other resources on Kubernetes.

Jaeger can run in three main configurations: *AllInOne*, *Production*, and *Streaming*. A full discussion of these configurations is outside the scope of this book (check the Jaeger docs link shared previously), but we will be using the AllInOne configuration. This configuration combines the Jaeger UI, Collector, Agent, and Ingestor into a single Pod, without any persistent storage included. This is perfect for demo purposes – to see production-ready configurations, check the Jaeger docs.

In order to create our Jaeger deployment, we need to tell the Jaeger Operator about our chosen configuration. We do that using the CRD that we created earlier – the Jaeger CRD. Create a new file for this CRD instance:

Generally, most tooling in metrics and logging will require you to provision resources on your cluster to forward metrics and logs to your service of choice. In the examples we've used in this chapter, these services are running in the cluster, though in commercial products these can often be separate SaaS applications where you log on to analyze your logs and see your metrics. For instance, with the EFK stack we provisioned in this chapter, you can pay Elastic for a hosted solution where the Elasticsearch and Kibana pieces of the solution would be hosted on Elastic's infrastructure, reducing complexity in the solution. There are also many other solutions in this space, from vendors including Sumo Logic, Logz.io, New Relic, DataDog, and AppDynamics.

For a production environment, it is common to use separate compute (either a separate cluster, service, or SaaS tool) to perform log and metric analytics. This ensures that the cluster running your actual software can be dedicated to the application alone, and any costly log searching or querying functionality can be handled separately. It also means that if our application cluster goes down, we can still view logs and metrics up until the point of the failure.

Summary

In this chapter, we learned about observability on Kubernetes. We first learned about the four major tenets of observability: metrics, logging, traces, and alerts. Then we discovered how Kubernetes itself provides tooling for observability, including how it manages logs and resource metrics and how to deploy Kubernetes Dashboard. Finally, we learned how to implement and use some key open source tools to provide visualization, searching, and alerting for the four pillars. This knowledge will help you build robust observability infrastructure for your future Kubernetes clusters and help you decide what is most important to observe in your cluster.

Understanding failure modes for distributed applications

Kubernetes components (and applications running on Kubernetes) are distributed by default if they run more than one replica. This can result in some interesting failure modes, which can be hard to debug.

For this reason, applications on Kubernetes are less prone to failure if they are stateless – in which case, the state is offloaded to a cache or database running outside of Kubernetes. Kubernetes primitives such as StatefulSets and PersistentVolumes can make it much easier to run stateful applications on Kubernetes – and with every release, the experience of running stateful applications on Kubernetes improves. Still, deciding to run fully stateful applications on Kubernetes introduces complexity and therefore the potential for failure.

Failure in distributed applications can be introduced by many different factors. Things as simple as network reliability and bandwidth constraints can cause major issues. These are so varied that *Peter D* https://www.rgoarchitects.com/Files/fallacies.pdf pen the *Fallacies of distributed computing* (along with *James Gosling*, who added the 8th point), which are commonly agreed-upon factors for failures in distributed applications. In the paper *Fallacies of distributed computing explained*, *Arnon Rotem-Gal-Oz* discusses the source of these fallacies (`https://www.rgoarchitects.com/Files/fallacies.pdf`).

The fallacies are as follows, in numerical order:

1. The network is reliable.
2. Latency is zero.

3. Bandwidth is infinite.
4. The network is secure.
5. The topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

Kubernetes has been engineered and developed with these fallacies in mind and is therefore more tolerant. It also helps address these issues for applications running on Kubernetes – but not perfectly. It is therefore very possible that your applications, when containerized and running on Kubernetes, will exhibit problems when faced with any of these issues. Each fallacy, when assumed to be untrue and taken to its logical conclusion, can introduce failure modes in distributed applications. Let's go through each of the fallacies as applied to Kubernetes and applications running on Kubernetes.

Chap 11: CI/CD on k8s

To begin, we will cover the landscape of Kubernetes resource template generation, and the reasons why a template generation tool should be used at all. Then, we will cover implementing CI/CD to Kubernetes, first with AWS CodeBuild, and next with FluxCD.

In this chapter, we will cover the following topics:

- Understanding options for template code generation on Kubernetes
- Implementing templates on Kubernetes with Helm and Kustomize

---

- Understanding CI/CD paradigms on Kubernetes – in-cluster and out-of-cluster
- Implementing in-cluster and out-of-cluster CI/CD with Kubernetes

Understanding options for template code generation on Kubernetes

As discussed in *Chapter 1*, *Communicating with Kubernetes*, one of the greatest strengths of Kubernetes is that its API can communicate in terms of declarative resource files. This allows us to run commands such as `kubectl apply` and have the control plane ensure that whatever resources are running in the cluster match our YAML or JSON file.

However, this capability introduces some unwieldiness. Since we want to have all our workloads declared in configuration files, any large or complex applications, especially if they include many microservices, could result in a large number of configuration files to write and maintain.

This issue is further compounded with multiple environments. Say we want development, staging, UAT, and production environments, this would require four separate YAML files per Kubernetes resource, assuming we wanted to maintain one resource per file for cleanliness.

Template variables:

Using template variables

Adding variables to our Helm chart templates is as simple as using double bracket – { { } } – syntax. What we put in the double brackets will be taken directly from the values that we use when installing our chart using dot notation.

Let's look at a quick example. So far, we have our app name (and container image name/version) hardcoded into our YAML file. This constrains us significantly if we want to use our Helm chart to deploy different applications or different application versions.

In order to address this, we're going to add template variables to our chart. Take a look at this resulting template:

See code - https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/blob/master/Chapter11/templated-deployment.yaml

```
apiVersion:
apps/v1
              kind: Deployment
              metadata:
                name: frontend-{{ .Release.Name }}
                labels:
                  app: frontend-{{ .Release.Name }}
                  chartVersion: {{ .Chart.version }}
              spec:
                replicas: 2
                selector:
                  matchLabels:
                    app: frontend-{{ .Release.Name }}
                template:
                  metadata:
                    labels:
                      app: frontend-{{ .Release.Name }}
                  spec:
                    containers:
                    - name: frontend-{{ .Release.Name }}
                      image: myrepo/{{ .Values.image.name }}
              :{{ .Values.image.tag }}
                      ports:
                      - containerPort: 80
```

Out-of-cluster CI/CD

In the first pattern, our CI/CD tool runs outside of our target Kubernetes cluster. We call this out-of-cluster CI/CD. There is a gray area where the tool may run in a separate Kubernetes cluster that is focused on CI/CD, but we will ignore that option for now as the difference between the two categories is still mostly valid.

You'll often find industry standard tooling such as Jenkins used with this pattern, but any CI tool that has the ability to run scripts and retain secret keys in a secure way can work here. A few examples are **GitLab CI**, **CircleCI**, **TravisCI**, **GitHub Actions**, and **AWS CodeBuild**. Helm is also a big part of this pattern, as out-of-cluster CI scripts can call Helm commands in lieu of kubectl.

Some of the strengths of this pattern are to be found in its simplicity and extensibility. This is a `push`-based pattern where changes to code synchronously trigger changes in Kubernetes workloads.

Some of the weaknesses of out-of-cluster CI/CD are scalability when pushing to many clusters, and the need to keep cluster credentials in the CI/CD pipeline so it has the ability to call kubectl or Helm commands.

In-cluster CI/CD

In the second pattern, our tool runs on the same cluster that our applications run on, which means that CI/CD happens within the same Kubernetes context as our applications, as pods. We call this in-cluster CI/CD. This in-cluster pattern can still have the "build" steps occur outside the cluster, but the deploy step happens from within the cluster.

These types of tools have been gaining popularity since Kubernetes was released, and many use custom resource definitions and custom controllers to do their jobs. Some examples are **FluxCD**, **Argo CD**, **JenkinsX**, and **Tekton Pipelines**. The **GitOps** pattern, where a Git repository is used as the source of truth for what applications should be running on a cluster, is popular in these tools.

Some of the strengths of the in-cluster CI/CD pattern are scalability and security. By having the cluster "pull" changes from GitHub via a GitOps operating model, the solution can be scaled to many clusters. Additionally, it removes the need to keep powerful cluster credentials in the CI/CD system, instead having GitHub credentials on the cluster itself, which can be much better from a security standpoint.

The weaknesses of the in-cluster CI/CD pattern include complexity, since this pull-based operation is slightly asynchronous (as `git pull` usually occurs on a loop, not always occurring exactly when changes are pushed).

Implementing in-cluster and out-of-cluster CI/CD with Kubernetes

Since there are so many options for CI/CD with Kubernetes, we will choose two options and implement them one by one so you can compare their feature sets. First, we'll implement CI/CD to Kubernetes on AWS CodeBuild, which is a great example implementation that can be reused with any external CI system that can run Bash scripts, including Bitbucket Pipelines, Jenkins, and others. Then, we'll move on to FluxCD, an in-cluster GitOps-based CI option that is Kubernetes-native. Let's start with the external option.

Chap 12 – security and compliance

Reviewing CVEs and security audits for Kubernetes

Kubernetes has encountered several **Common Vulnerabilities and Exposures** (**CVEs**) in its storied history. The MITRE CVE database, at the time of writing, lists 73 CVE announcements from 2015 to 2020 when searching for `kubernetes`. Each one of these is related either directly to Kubernetes, or to a common open source solution that runs on Kubernetes (like the NGINX ingress controller, for instance).

Several of these were critical enough to require hotfixes to the Kubernetes source, and thus they list the affected versions in the CVE description. A full list of all CVEs related to Kubernetes can be found at `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=kubernetes`. To give you an idea of some of the issues that have been found, let's review a few of these CVEs in chronological order.

Understanding CVE-2016-1905 – Improper admission control

This CVE was one of the first major security issues with production Kubernetes. The National Vulnerability Database (a NIST website) gives this issue a base score of 7.7, putting it in the high-impact category.

With this issue, a Kubernetes admission controller would not ensure that a `kubectl patch` command followed admission rules, allowing users to completely work around the admission controller – a nightmare in a multitenant scenario.

PSP, network policy: good Netpol in p 29x-303!

Full-restriction-policy.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: full-restriction-policy
  namespace: development
spec:
```



```
  policyTypes:
  - Ingress
  - Egress
  podSelector: {}
```

In this `NetworkPolicy`, we specify that we will be including both an `Ingress` and `Egress` policy, but we don't write a block for either of them. This has the effect of automatically denying any traffic for both `Egress` and `Ingress` since there are no rules for traffic to match against.

Additionally, our `{}` Pod selector value corresponds to selecting every Pod in the

Falco p. 303 -

How does Falco work? In real time, Falco parses system calls from the Linux kernel. It then filters these system calls through rules – which are sets of configurations that can be applied to the Falco engine. Whenever a rule is broken by a system call, Falco triggers an alert. It's that simple!

Falco ships with an extensive set of default rules that add significant observability at the kernel level. Custom rules are of course supported by Falco – and we will show you how to write them.

First, however, we need to install Falco on our cluster! Luckily, Falco can be installed using Helm. However, it is very important to note that there are a few different ways to install Falco, and they differ significantly in how effective they can be in the event of a breach.

We're going to be installing Falco using the Helm chart, which is simple and works well for managed Kubernetes clusters, or any scenario where you may not have direct access to the worker nodes.

However, for the best possible security posture, Falco should be installed directly onto the Kubernetes nodes at the Linux level. The Helm chart, which uses a DaemonSet is great for ease of use but is inherently not as secure as a direct Falco installation. To install Falco directly to your nodes, check the installation instructions at `https://falco.org/docs/installation/`.

First, let's look at the rule entry itself. This rule uses a few helper entries, several macros, and lists – but we'll get to those in a second:

```
- rule: Launch Privileged Container
  desc: Detect the initial process started in a privileged
container. Exceptions are made for known trusted images.
```



```
  condition: >
    container_started and container
    and container.privileged=true
    and not falco_privileged_containers
    and not user_privileged_containers
  output: Privileged container started (user=%user.name
command=%proc.cmdline %container.info image=%container.image.
repository:%container.image.tag)
  priority: INFO
  tags: [container, cis, mitre_privilege_escalation, mitre_
lateral_movement]
```

As you can see, a Falco rule has several parts. First, we have the rule name and

Understanding Kubernetes audit event rules in Falco

Structurally, these Kubernetes audit event rules work the same way as Falco's Linux system call rules. Here's an example of one of the default Kubernetes rules in Falco:

```
- rule: Create Disallowed Pod
  desc: >
    Detect an attempt to start a pod with a container image
outside of a list of allowed images.
  condition: kevt and pod and kcreate and not allowed_k8s_
containers
  output: Pod started with container not in allowed list
(user=%ka.user.name pod=%ka.resp.name ns=%ka.target.namespace
images=%ka.req.pod.containers.image)
  priority: WARNING
  source: k8s_audit
  tags: [k8s]
```

This rule acts on Kubernetes audit events in Falco (essentially, control plane events) to alert when a Pod is created that isn't on the list `allowed_k8s_containers`. The default `k8s` audit rules contain many similar rules, most of which output formatted logs when triggered.

Now, we talked about Pod security policies a bit earlier in this chapter – and you may be seeing some similarities between PSPs and Falco Kubernetes audit event rules. For instance, take this entry from the default Kubernetes Falco rules:

```
- rule: Create HostNetwork Pod
  desc: Detect an attempt to start a pod using the host
```

Section4: chap 13-15

*** Extending k8s, e.g. CRDs, cloud controller manager, cluster-autoscalar add-on, service meshes/serverless, Minio, stateful workloads (Cockroach DB), RabbitMQ on k8s …

Chap 13: CRD

Writing a custom resource definition

For CRDs, Kubernetes uses the OpenAPI V3 specification. For more information on OpenAPI V3, you can check the official documentation at `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md`, but we'll soon see how exactly this translates into Kubernetes CRD definitions.

Let's take a look at an example CRD spec. Now let's be clear, this is not how YAMLs of any specific record of this CRD would look. Instead, this is simply where we define the requirements for the CRD inside of Kubernetes. Once created, Kubernetes will accept resources matching the spec and we can start making our own records of this type.

Here's an example YAML for a CRD spec, which we are calling `delayedjob`. This highly simplistic CRD is intended to start a container image job on a delay, which prevents users from having to script in a delayed start for their container. This CRD is quite brittle, and we don't recommend anyone actually use it, but it does well to highlight the process of building a CRD. Let's start with a full CRD spec YAML, then break it down:

Code - https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/tree/master/Chapter13

Using cloud-specific Kubernetes extensions

Usually available by default in managed Kubernetes services such as Amazon EKS, Azure AKS, and Google Cloud's GKE, cloud-specific Kubernetes extensions and controllers can integrate tightly with the cloud platform in question and make it easy to control other cloud resources from Kubernetes.

Even without adding any additional third-party components, a lot of this cloud-specific functionality is available in upstream Kubernetes via the **cloud-controller-manager** (**CCM**) component, which contains many options for integrating with the major cloud providers. This is the functionality that is usually enabled by default in the managed Kubernetes services on each public cloud – but they can be integrated with any cluster running on that specific cloud platform, managed or not.

In this section, we will review a few of the more common cloud extensions to Kubernetes, both in **cloud-controller-manager** (**CCM**) and functionality that requires the installation of other controllers such as **external-dns** and **cluster-autoscaler**. Let's start with some of the heavily used CCM functionality.

Using cloud-specific Kubernetes extensions

Usually available by default in managed Kubernetes services such as Amazon EKS, Azure AKS, and Google Cloud's GKE, cloud-specific Kubernetes extensions and controllers can integrate tightly with the cloud platform in question and make it easy to control other cloud resources from Kubernetes.

Even without adding any additional third-party components, a lot of this cloud-specific functionality is available in upstream Kubernetes via the **cloud-controller-manager** (**CCM**) component, which contains many options for integrating with the major cloud providers. This is the functionality that is usually enabled by default in the managed Kubernetes services on each public cloud – but they can be integrated with any cluster running on that specific cloud platform, managed or not.

In this section, we will review a few of the more common cloud extensions to Kubernetes, both in **cloud-controller-manager** (**CCM**) and functionality that requires the installation of other controllers such as **external-dns** and **cluster-autoscaler**. Let's start with some of the heavily used CCM functionality.

Chap 14 – service mesh and serverless:

14
Service Meshes and Serverless

This chapter discusses advanced Kubernetes patterns. First, it details the in-vogue service mesh pattern, where observability and service-to-service discovery are handled by a sidecar proxy, as well as a guide to setting up Istio, a popular service mesh. Lastly, it describes the serverless pattern and how it can be applied in Kubernetes. The major case study in this chapter will include setting up Istio for an example application and service discovery, along with Istio ingress gateways.

Let's start with a discussion of the sidecar proxy, which builds the foundation of service-to-service connectivity for service meshes.

In this chapter, we will cover the following topics:

- Using sidecar proxies
- Adding a service mesh to Kubernetes
- Implementing serverless on Kubernetes

Nginx, Envoy

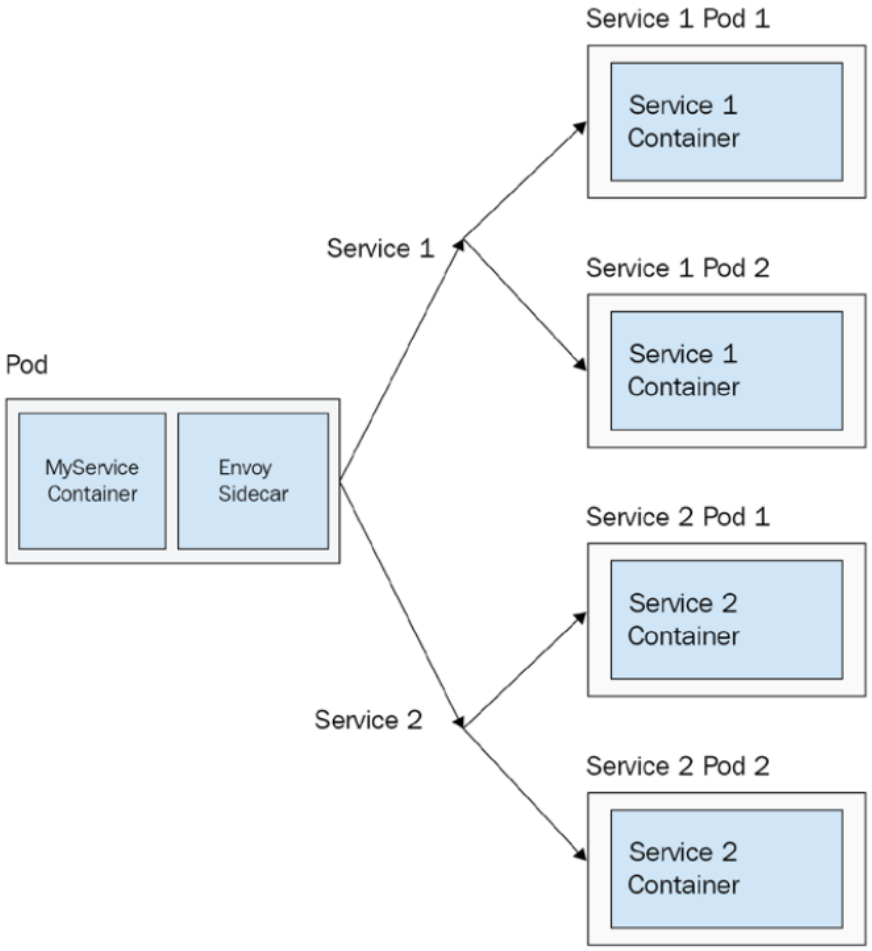


Figure 14.2 – Outbound envoy proxy

Meshes:

Adding a service mesh to Kubernetes

A *service mesh* pattern is a logical extension of the sidecar proxy. By attaching sidecar proxies to every Pod, a service mesh can control functionality for service-to-service requests, such as advanced routing rules, retries, and timeouts. In addition, by having every request pass through a proxy, service meshes can implement mutual TLS encryption between services for added security and can give administrators incredible observability into requests in their cluster.

There are several service mesh projects that support Kubernetes. The most popular are as follows:

- *Istio*
- *Linkerd*
- *Kuma*
- *Consul*

Implementing serverless on Kubernetes

Serverless patterns on cloud providers have quickly been gaining in popularity. Serverless architectures consist of compute that can automatically scale up and down, even scaling all the way to zero (where zero compute capacity is being used to serve a function or other application). **Function-as-a-Service** (**FaaS**) is an extension of the serverless pattern, where function code is the only input, and the serverless system takes care of routing requests to compute and scale as necessary. AWS Lambda, Azure Functions, and Google Cloud Run are some of the more popular FaaS/serverless options officially supported by cloud providers. Kubernetes also has many different serverless frameworks and libraries that can be used to run serverless, scale-to-zero workloads as well as FaaS on Kubernetes. Some of the most popular ones are as follows:

- *Knative*
- *Kubeless*
- *OpenFaaS*
- *Fission*

A full discussion of all serverless options on Kubernetes is beyond the scope of this book, so we'll focus on two different ones, which aim to serve two vastly different use cases: *OpenFaaS* and *Knative*.

While Knative is highly extensible and customizable, it uses multiple coupled components that add complexity. This means that some added configuration is necessary to get started with an FaaS solution, since functions are just one of many other patterns that Knative supports. OpenFaaS, on the other hand, makes getting up and running with serverless and FaaS on Kubernetes extremely easy. Both technologies are valuable for different reasons.

15

Stateful Workloads on Kubernetes

This chapter details the current state of the industry when it comes to running stateful workloads in databases. We will discuss the use of Kubernetes (and popular open source projects) for running databases, storage, and queues on Kubernetes. Case study tutorials will include running object storage, a database, and a queue system on Kubernetes.

In this chapter, we will first understand how stateful applications run on Kubernetes and then learn how to use Kubernetes storage for stateful applications. We will then learn how to run databases on Kubernetes, as well as covering messaging and queues. Let's start with a discussion of why stateful applications are much more complex than stateless applications on Kubernetes.

In this chapter, we will cover the following topics:

- Understanding stateful applications on Kubernetes
- Using Kubernetes storage for stateful applications
- Running databases on Kubernetes
- Implementing messaging and queues on Kubernetes

Kubernetes-compatible databases

In addition to typical **databases** (**DBs**) and key-value stores such as **Postgres**, **MySQL**, and **Redis** that can be deployed on Kubernetes with StatefulSets or community operators, there are some major made-for-Kubernetes options:

- **CockroachDB**: A distributed SQL database that can be deployed seamlessly on Kubernetes
- **Vitess**: A MySQL sharding orchestrator that allows global scalability for MySQL, also installable on Kubernetes via an operator
- **YugabyteDB**: A distributed SQL database similar to **CockroachDB** that also supports Cassandra-like querying

Next, let's look at queuing and messaging on Kubernetes.

Queues, streaming, and messaging on Kubernetes

Again, there are industry-standard options such as **Kafka** and **RabbitMQ** that can be deployed on Kubernetes using community Helm charts and operators, in addition to some purpose-made open- and closed-source options:

- **NATS**: Open source messaging and streaming system
- **KubeMQ**: Kubernetes-native message broker

Next, let's look at object storage on Kubernetes.

Object storage on Kubernetes

Object storage takes volume-based persistent storage from Kubernetes and adds on an object storage layer, similar to (and in many cases compatible with the API of) Amazon S3:

- **Minio**: S3-compatible object storage built for high performance.
- **Open IO**: Similar to *Minio*, this has high performance and supports S3 and Swift storage.

Next, let's look at a few honorable mentions.

Honorable mentions

In addition to the preceding generic components, there are some more specialized (but still categorical) stateful applications that can be run on Kubernetes:

- **Key and auth management**: **Vault**, **Keycloak**
- **Container registries**: **Harbor**, **Dragonfly**, **Quay**
- **Workflow management**: **Apache Airflow** with a Kubernetes Operator

Now that we've reviewed a few categories of stateful applications, let's talk about how these state-heavy applications are typically implemented on Kubernetes.

Understanding strategies for running stateful applications on Kubernetes

Though there is nothing inherently wrong with deploying a stateful application on Kubernetes with a ReplicaSet or Deployment, you will find that the majority of stateful applications on Kubernetes use StatefulSets. We talked about StatefulSets in *Chapter 4*, *Scaling and Deploying Your Application*, but why are they so useful for applications? We will review and answer this question in this chapter.

The main reason is Pod identity. Many distributed stateful applications have their own clustering mechanism or consensus algorithm. In order to smooth over the process for these types of applications, StatefulSets provide static Pod naming based on an ordinal system, starting from `0` to `n`. This, in combination with a rolling update and creation method, makes it much easier for applications to cluster themselves, which is extremely important for cloud-native databases such as CockroachDB.

To illustrate how and why StatefulSets can help run stateful applications on Kubernetes, let's look at how we might run MySQL on Kubernetes with StatefulSets.

Now, to be clear, running a single Pod of MySQL on Kubernetes is extremely simple. All we need to do is find a MySQL container image and ensure that it has the proper configuration and `startup` command.

However, when we look to scale our database, we start to run into issues. Unlike a simple stateless application, where we can scale our deployment without creating new state, MySQL (like many other DBs) has its own method of clustering and consensus. Each member of a MySQL cluster knows about the other members, and most importantly, it knows which member of the cluster is the leader. This is how databases like MySQL can offer consistency guarantees and **Atomicity, Consistency, Isolation, Durability (ACID)** compliance.

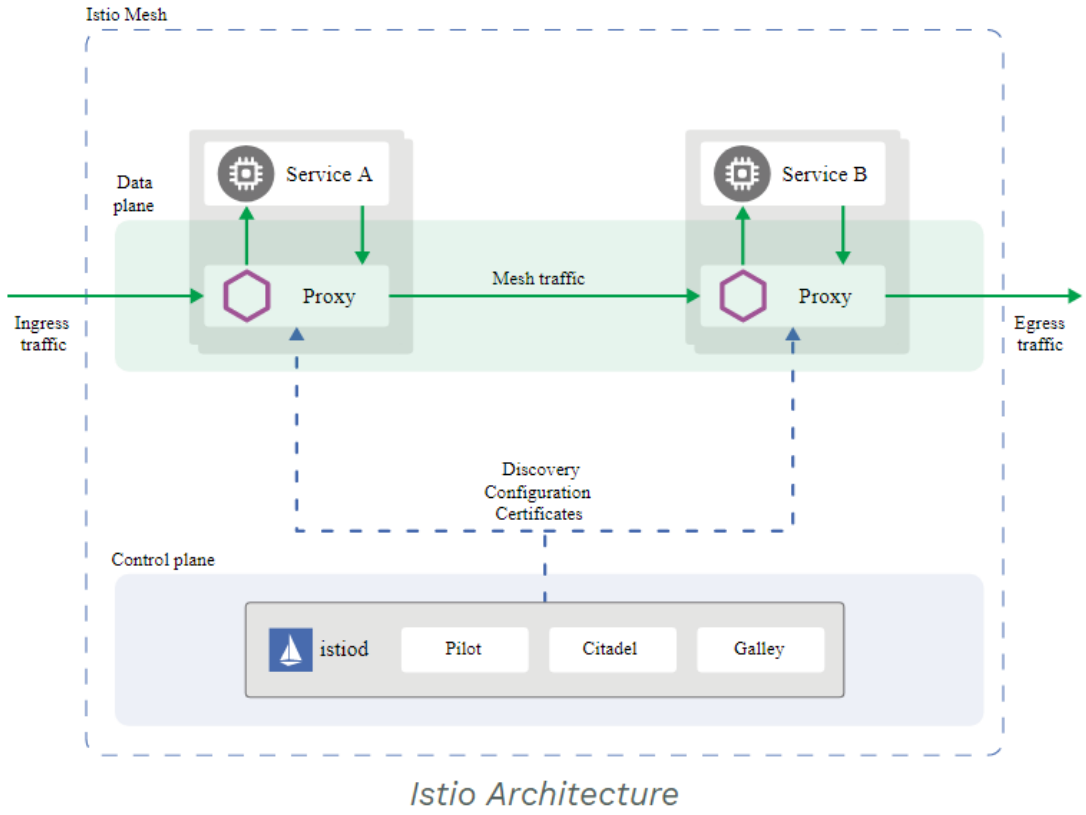
Minio, cockroach DB, RabbitMQ examples! https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/tree/master/Chapter15

A lot of questions and quiz!

Suggestions and Typos:

Suggestions:

1/ can add more graphs all over the book, e.g. Istio components, e.g. Pilot, Citadel, Gallery, etc. see https://istio.io/v1.6/docs/ops/deployment/architecture/

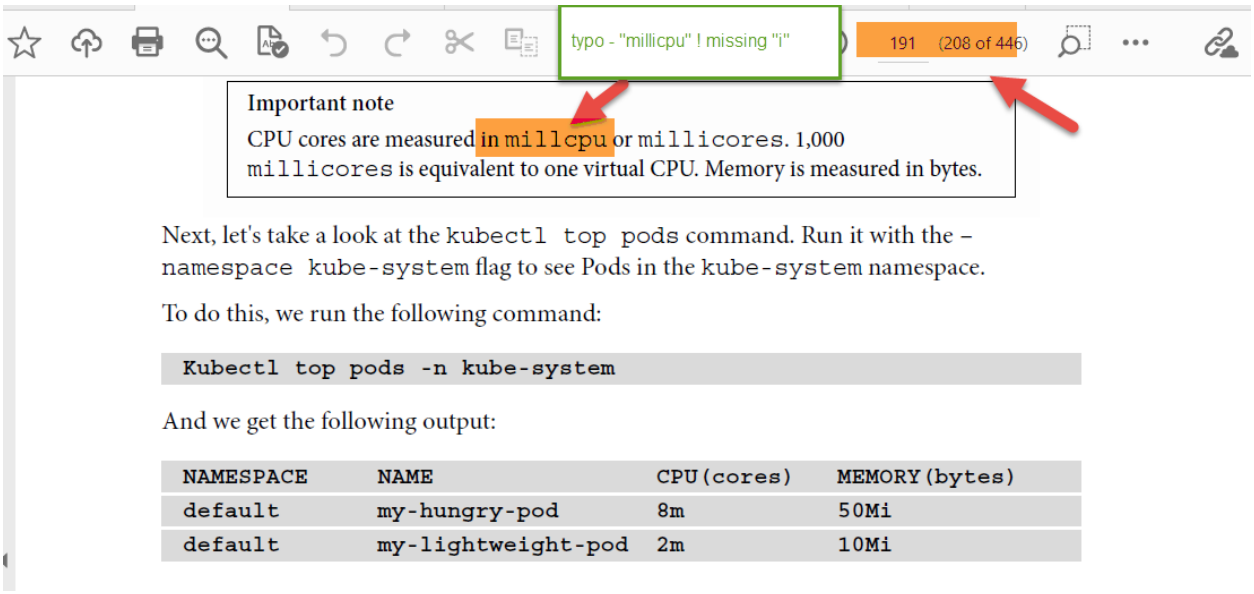


Istio Architecture

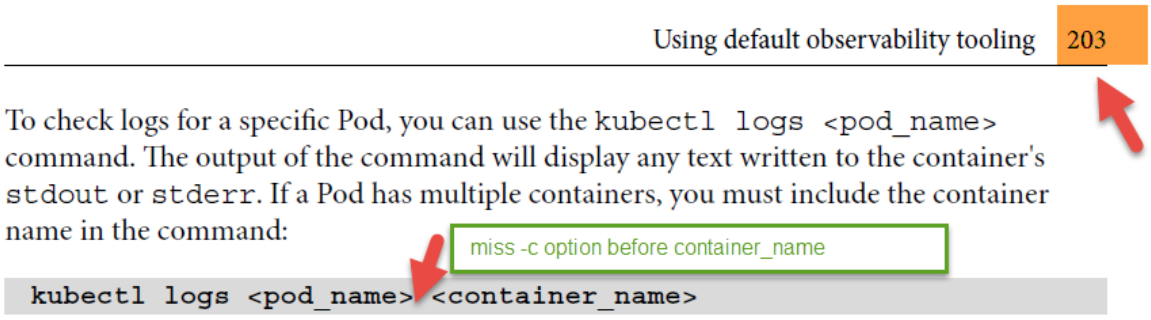
Similar suggestions for EFK, Jaeger, etc. so new readers can understand the set up better.

2/ I filed a PR https://github.com/PacktPublishing/Cloud-Native-with-Kubernetes/pull/1 for the code on p.244

Typos:



Also miss –c below:



Like they talk about Prometheus and Gafana (p.209). Also, EFK stack on p.208. Very complete in k8s observability!

In our `Chart.yaml` file, `apiVersion` corresponds to the version of Helm that the chart corresponds to. Somewhat confusingly, the current release of Helm, Helm V3, uses `apiVersion v2`, while older versions of Helm, including Helm V2, also use `apiVersion v2`.

Next, the `name` field corresponds to the name of our chart. This is pretty self-explanatory, although remember that we have the ability to name a specific release of a chart – something that comes in handy for multiple environments.

Finally, we have the `version` field, which corresponds to the version of the chart. This field supports **SemVer** (semantic versioning).

So, what do our templates actually look like? Helm charts use the Go templates library under the hood (see `https://golang.org/pkg/text/template/` for more information) and support all sorts of powerful manipulations, helper functions, and much, much more. For now, we will keep things extremely simple to give you an idea of the basics. A full discussion of Helm chart creation could be a book on its own!

To start, we can use a Helm CLI command to autogenerate our `Chart` folder, with all the previous files and folders, minus subcharts and values files, generated for you. Let's try it – first create a new Helm chart with the following command:

```
helm create myfakenodeapp
```

This command will create an autogenerated chart in a folder named `myfakenodeapp`. Let's check the contents of our `templates` folder with the following command:

```
Ls myfakenodeapp/templates
```

lowercase L

This command will result in the following output:

```
helpers.tpl
deployment.yaml
NOTES.txt
service.yaml
```


It looks like our Pod is mounting an empty volume as a scratch disk. It also has two containers in each Pod – a sidecar used for application tracing, and our app itself. We'll need this information to `ssh` into one of the Pods (it doesn't matter which one for this exercise) using the `kubectl exec` command.

We can do it using the following command:

no dot at the end

```
kubectl exec -it app-2-ss-1 app2 -- sh.
```

This command should give you a bash terminal as the output:

```
> kubectl exec -it app-2-ss-1 app2 -- sh
#
```