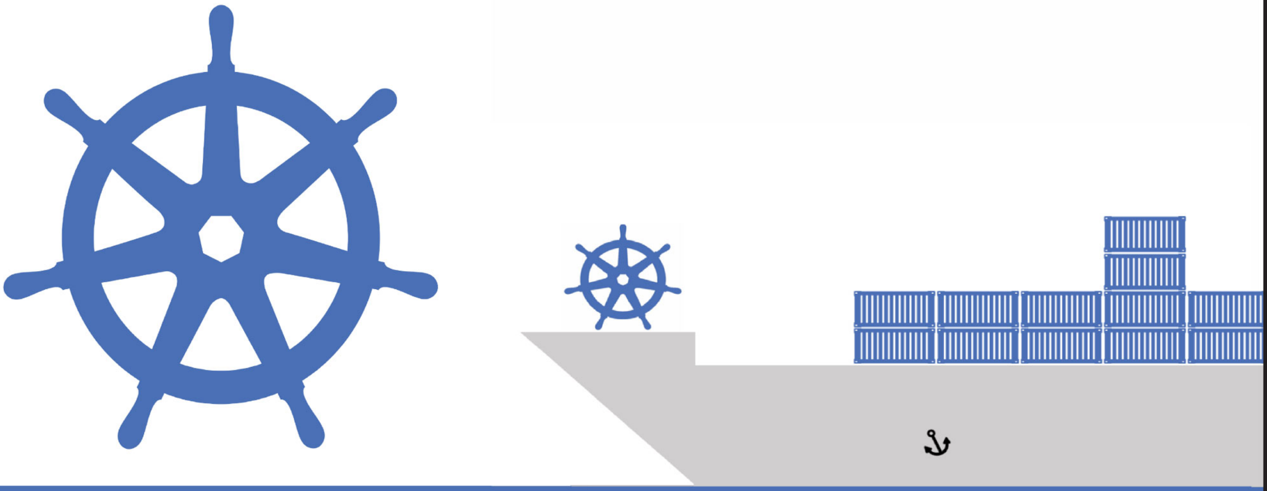
Mar 2019

The Kubernetes Book



Nigel Poulton

& Pushkar Joglekar

The Kubernetes Book

Nigel Poulton

This book is for sale at http://leanpub.com/thekubernetesbook

This version was published on 2019-03-09

Education is about inspiring and creating opportunities. I hope this book, and my video training courses, inspire you and create some great new opportunities!

A huge thanks to my family for putting up with me. I'm a geek who thinks he's software running on midrange biological hardware. I know it's not easy living with me.

Thanks as well to everyone who watches my Pluralsight and A Cloud Guru training videos. I love connecting with you and appreciate all the feedback I've had over the years. That is what inspired me to write this book. I think you'll love it, and I hope it helps drive your career forward.

@nigelpoulton

Contents

0: About the book

This is an *up-to-date* book about Kubernetes. It's relatively short, and it's straight-to-the-point.

Let me be clear about this, as I don't want to mislead people… **This is not a deep dive, and it does not attempt to cover everything.** This is an easy-to-read book that covers the fundamental and important parts of Kubernetes.

Paperback

A paperback version is available in selected Amazon markets. I have no control over which markets Amazon makes the paperback available in – if it was my choice, I'd make it available everywhere.

I've opted for a **high-quality**, **full-color paperback** that I think you'll love. That means no cheap paper, and no black-and-white diagrams from the 1990's.

Audio book

I plan to make an audio version of the book available via Audible in April 2019. There will be minor tweaks to the examples and labs so that they are easier to follow in an audio book.

eBook and Kindle editions

The easiest place to get an electronic copy is leanpub.com. It's a slick platform and updates are free.

You can also get a Kindle edition from Amazon, which also gets free updates. However, Kindle is notoriously bad at delivering updates. If you have problems getting updates to your Kindle, contact Kindle Support and they will resolve the issue.

Feedback

I'd love it if you'd give the book a review on Amazon. Writing technology books is lonely work. I literally spent months making this book as great as possible, so a couple of minutes of your time for a review would be magic. No pressure though, I won't hunt you down at the next KubeCon if you don't.

Why should anyone read this book or care about Kubernetes?

Kubernetes is white-hot, and Kubernetes skills are in high demand. So, if you want to push ahead with your career and work with a technology that's shaping the future, you need to read this book. If you don't care about your career and are fine being left behind, don't read it. It's the truth.

Should I buy the book if I've already watched your video training courses?

Kubernetes is Kubernetes. So yes, there's obviously some similar content between my books and video courses. But reading books and watching videos are totally different experiences. In my opinion, videos are more fun, but books are easier to write notes in and flick through when you're trying to find something.

If I was you, I'd watch the videos *and* get the book. They complement each other, and learning via multiple methods is a proven strategy.

Final word: Take a look at the reviews my videos and books have. That should reassure you they'll be great investments.

Some of my Video courses:

- Getting Started with Kubernetes (pluralsight.com)
- Kubernetes Deep Dive (acloud.guru)
- Docker Deep Dive (pluralsight.com)

Free updates to the book

I've done everything I can to make sure your investment in this book is as safe as possible!

All Kindle and Leanpub customers receive all updates at no extra cost. Updates work well on Leanpub, but it's a different story on Kindle. Many readers complain that their Kindle devices don't get access to updates. This is a common issue, and one that is easily resolved by *contacting Kindle Support.*

If you buy the paperback version from **Amazon.com**, you can get the Kindle version at the discounted price of $2.99. This is done via the *Kindle Matchbook* program. Unfortunately, Kindle Matchbook is only available in the US, and it's buggy — sometimes the Kindle Matchbook icon doesn't appear on the book's Amazon selling page. Contact Kindle Support if you have issues like this and they'll sort things out.

That's the best I can do!

Things will be different if you buy the book through other channels, as I have no control over them. I'm a techie, not a book publisher ;-)

Versions of the book

Kubernetes is developing fast! As a result, the value of a book like this is inversely proportional to how old it is. In other words, the older any Kubernetes book is, the less valuable it is. With this in mind, **I'm committed to updating the book at least once per year**. And when I say "update", I mean real updates — every word and concept is reviewed, and every example is tested and updated. **I'm 100% committed to making this book the best Kubernetes book in the world**

If at least one update a year seems like a lot… welcome to the new normal.

We no longer live in a world where a 2-year-old technology book is valuable. In fact, I question the value of a 1-year-old book on a topic that's developing as fast as Kubernetes. As an author I'd love to write a book that was useful for 5 years. But that's not the world we live in. Again… welcome to the new normal.

- **Version 4** March 2019. All content updated and all examples tested on the latest versions of Kubernetes. Added new Storage Chapter. Added new real-world security section with two new chapters.
- **Version 3** November 2018. Re-ordered some chapters for better flow. Removed the *ReplicaSets* chapter and shifted that content to an improved *Deployments* chapter. Added new chapter giving overview of other major concepts not covered in dedicated chapters.
- **Version 2.2** January 2018. Fixed a few typos, added several clarifications, and added a couple of new diagrams.
- **Version 2.1** December 2017. Fixed a few typos and updated Figures 6.11 and 6.12 to include missing labels.
- **Version 2.** October 2017. Added new chapter on ReplicaSets. Added significant changes to Pods chapter. Fixed typos and made a few other minor updates to existing chapters.
- **Version 1.** Initial version.

2: Kubernetes principles of operation

In this chapter, we'll learn about the major components needed to build a Kubernetes cluster and deploy an app. The aim is to give you an overview of the major concepts. But don't worry if you don't understand everything straight away, we'll cover most things again as we progress through the book.

We'll divide the chapter as follows:

- Kubernetes from 40K feet
- Masters and nodes
- Packaging apps
- Declarative configuration and desired state
- Pods
- Deployments
- Services

Kubernetes from 40K feet

At the highest level, Kubernetes is two things:

- A cluster for running applications
- An orchestrator of cloud-native microservices apps.

On the *cluster* front, Kubernetes is like any other cluster – a bunch of nodes and a control plane. The control plane exposes an API, has a scheduler for assigning work to nodes, and state is recorded in a persistent store. Nodes are where application services run.

Kubernetes is API-driven and uses standard HTTP RESTful verbs to view and update the cluster.

On the *orchestrator* front, "orchestrator" is just a fancy name for an application that's made from lots of small independent services that work together to form a useful app.

Let's look at a quick analogy.

In the real world, a football (soccer) team is made of individuals. No two are the same, and each has a different role to play in the team – some defend, some attack, some are great at passing, some tackle, some shoot… Along comes the coach, and he or she gives everyone a position and organizes them into a team with a purpose. We go from Figure 2.1 to Figure 2.2.

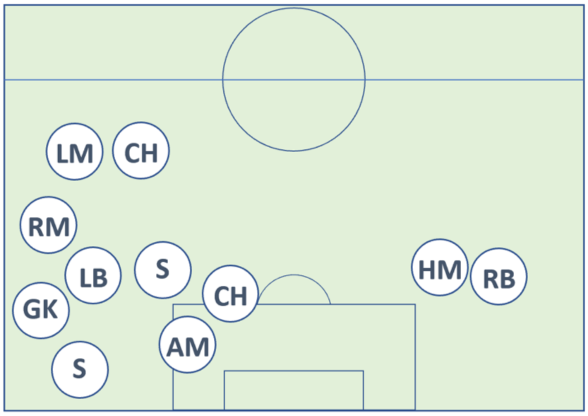


Figure 2.1

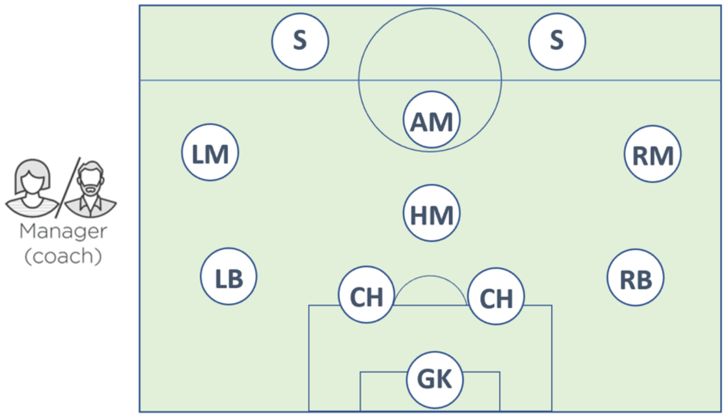


Figure 2.2

The coach also makes sure that the team maintains its formation, sticks to the game-plan, and deals with any injuries and other changes in circumstances. Well guess what… microservices apps in the Kubernetes world are the same.

Stick with me on this…

We start out with lots of individual specialised services. Some serve web pages, some do authentication, some perform searches, others persist data. Kubernetes comes along – a bit like the coach in the football analogy –- organizes everything into a useful app and keeps things running smoothly. It even responds to changes in circumstances.

In the sports world we call this *coaching*. In the application world we call it *orchestration*. Kubernetes is an *orchestrator*.

To make this happen, we start out with an app, package it up and give it to the cluster (Kubernetes).

The cluster is made up of one or more *masters* and a bunch of *nodes*.

The masters, sometimes called *heads* or *head nodes*, are in-charge of the cluster. This means they make the scheduling decisions, perform monitoring, implement changes, respond to events, and more. For these reasons, we often refer to the masters as the *control plane*.

The nodes are where application services run, and we sometimes call them the *data plane*. They have a reporting line back to the masters, and constantly watch for new work assignments.

To run applications on a Kubernetes cluster we follow this simple pattern:

1. Write the application as small independent services in our favourite languages.
2. Package each service in its own container.
3. Wrap each container in its own Pod.
4. Deploy Pods to the cluster via higher-level controllers such as; *Deployments, DaemonSets, StatefulSets, CronJobs etc.*

We're still near the beginning of the book and you're not expected to know what all of this means yet. However, at a high-level, *Deployments* offer scalability and rolling updates, *DaemonSets* run one instance of a Pod on every node in the cluster, *StatefulSets* are for stateful application components, and *CronJobs* are for work that needs to run at set times. There are more than these, but these will do for now.

Kubernetes likes to manage applications *declaratively*. This is a pattern where we describe how we want our application to look and feel in a set of YAML files, `POST` these files to Kubernetes, then sit back while Kubernetes makes it all happen.

But it doesn't stop there. Because the declarative pattern defines how we want an application to look, Kubernetes can watch it and make sure things are how they should be. If something isn't as it should be, Kubernetes tries to fix it.

That's the big picture. Let's dig a bit deeper.

Masters and nodes

A Kubernetes cluster is made of masters and nodes. These are Linux hosts that can be VMs, bare metal servers in your data center, or instances in a private or public cloud.

Masters (control plane)

A Kubernetes master is a collection of system services that make up the control plane of the cluster.

The simplest setups run all the master *services* on a single host. However, this is only suitable for labs and test environments. For production environments, multi-master high availability (HA) is a **must**

have. This is why the major cloud providers implement HA masters as part of their Kubernetes-as-a-Service platforms such as Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS), and Google Kubernetes Engine (GKE).

Generally speaking, running 3 or 5 replicated masters in an HA configuration is recommended.

It's also considered a good practice not to run user applications on masters. This allows masters to concentrate entirely on managing the cluster.

Let's take a quick look at the different master services that make up the control plane.

The API server

The API server is the Grand Central Station of Kubernetes. All communication, between all components, goes through the API server. We'll get into the detail later in the book, but it's important to understand that internal system components, as well as external user components, all communicate via the same API.

It exposes a RESTful API that we `POST` YAML configuration files to over HTTPS. These YAML files, which we sometimes call *manifests*, contain the desired state of our application. This includes things like; which container image to use, which ports to expose, and how many Pod replicas to run.

All requests to the API Server are subject to authentication and authorization checks, but once these are done, the config in the YAML file is validated, persisted to the cluster store, and deployed to the cluster.

You can think of the API server as the brains of the cluster – where the smarts are implemented.

The cluster store

If the API server is the brains of the cluster, the *cluster store* is its heart. It's the only stateful part of the control plane, and it persistently stores the entire configuration and state of the cluster. As such, it's a vital component of the cluster – no cluster store, no cluster.

The cluster store is currently based on **etcd**, a popular distributed database. As it's the *single source of truth* for the cluster, you should run between 3-5 etcd replicas for high-availability, and you should provide adequate ways to recover when things go wrong.

On the topic of *availability*, etcd prefers consistency over availability. This means that it will not tolerate a split-brain situation and will halt updates to the cluster in order to maintain consistency. However, if etcd becomes unavailable, applications running on the cluster should continue to work, it's just updates to the cluster configuration that will be halted.

As with all distributed databases, consistency of writes to the database is important. For example, multiple writes to the same value originating from different nodes needs to be handled. etcd uses the popular RAFT consensus algorithm to accomplish this.

The controller manager

The controller manager is a *controller of controllers* and is shipped as a single monolithic binary. However, despite it running as a single process, it implements multiple independent control loops that watch the cluster and respond to events.

Some of the control loops include; the node controller, the endpoints controller, and the replicaset controller. Each one runs as a background watch-loop that is constantly watching the API Server for changes – the aim of the game is to ensure the *current state* of the cluster matches the *desired state* (more on this shortly).

The logic implemented by each control loop is effectively this:

1. Obtain desired state
2. Observe current state
3. Determine differences
4. Reconcile differences

This logic is at the heart of Kubernetes and declarative design patterns.

Each control loop is also extremely specialized and only interested in its own little corner of the Kubernetes world. No attempt is made to over-complicate things by implementing awareness of other parts of the system – each takes care of its own task and leaves other components alone. This is key to the distributed design of Kubernetes and adheres to the Unix philosophy of building complex systems from small specialized parts.

> **Note**: Throughout the book we'll use terms like *control loop, watch loop,* and *reconciliation loop* to mean the same thing.

The scheduler

At a high level, the scheduler watches for new work tasks and assigns them to appropriate healthy nodes. Behind the scenes, it implements complex logic that filters out nodes incapable of running the Pod and then ranks the nodes that are capable. The ranking system itself is complex, but the node with the highest ranking points is eventually selected to run the Pod.

When identifying nodes that are capable of running the Pod, the scheduler performs various predicate checks. These include; is the node tainted, are there any affinity or anti-affinity rules, is the Pod's network port available on the node, does the node have sufficient free resources etc. Any node incapable of running the Pod is ignored, and the remaining Pods are ranked according to things such as; does the node already have the required image, how much free resource does the node have, how many Pods is the node already running. Each criteria is worth points, and the node with the most points is selected to run the Pod.

If the scheduler cannot find a suitable node, the Pod cannot be scheduled and goes into pending.

It's not the job of the scheduler to perform the mechanics of *running* Pods, it just picks the nodes they will be *scheduled* on.

The cloud controller manager

If you're running your cluster on a supported public cloud platform, such as AWS, Azure, or GCP, your control plane will be running a *cloud controller manager*. Its job is to manage integrations with underlying cloud technologies and services such as, instances, load-balancers, and storage.

Control Plane summary

Kubernetes masters run all of the cluster's control plane services. Think of it as brains of the cluster where all the control and scheduling decisions are made. Behind the scenes, a master is made up of lots of small specialized control loops and services. These include the API server, the cluster store, the controller manager, and the scheduler.

The API Server is the front-end into the control plane and the only component in the control plane that we interact with directly. By default, it exposes a RESTful endpoint on port 443.

Figure 2.3 shows a high-level view of a Kubernetes master (control plane).

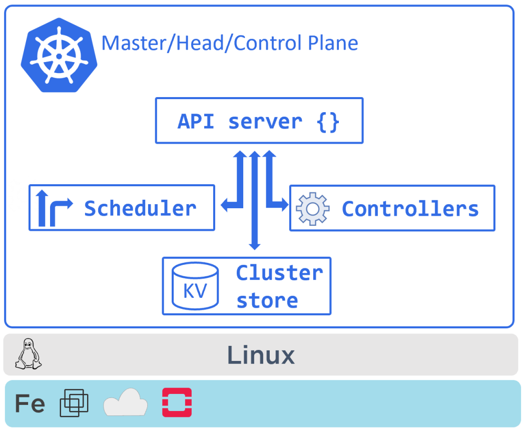


Figure 2.3 - Kubernetes Master

Nodes

Nodes are the workers of a Kubernetes cluster. At a high-level they do three things:

1. Watch the API Server for new work assignments
2. Execute new work assignments
3. Report back to the control plane

As we can see from Figure 2.4, they're are a bit simpler than *masters*. Let's look at the three major components of a node.

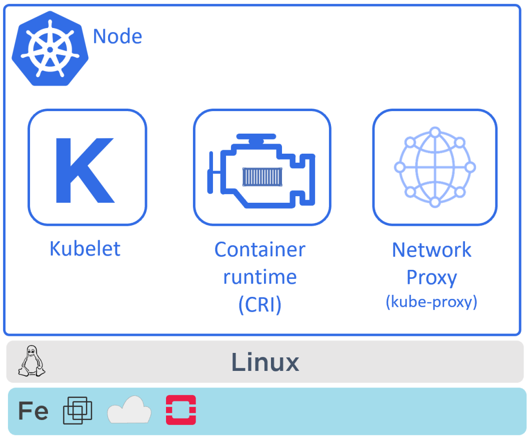


Figure 2.4 - Kubernetes Node (formerly Minion)

Kubelet

The Kubelet is the star of the show on every Node. It's the main Kubernetes agent, and it runs on every node in the cluster. In fact, it's common to use the terms *node* and *kubelet* interchangeably.

When you join a new node to a cluster, the process involves installation of the kubelet which is then responsible for the node registration process. This effectively pools the node's CPU, RAM, and storage into the wider cluster pool. Think back to the previous chapter where we talked about Kubernetes being a data center OS and abstracting data center resources into a single usable pool.

One of the main jobs of the kubelet is to watch the API server for new work assignments. Any time it sees one, it executes the task and maintains a reporting channel back to the control plane. It also keeps an eye on local static Pod definitions.

If a kubelet can't run a particular task, it reports back to the master and lets the control plane decide what actions to take. For example, if a Pod fails to start on a node, the kubelet is **not** responsible for finding another node to run it on. It simply reports back to the control plane and the control plane decides what to do.

Container runtime

The Kubelet needs a container runtime to perform container-related tasks – things like pulling images and starting and stopping containers.

In the early days, Kubernetes had native support for a few container runtimes such as Docker. More recently, it has moved to a plugin model called the Container Runtime Interface (CRI). This is an abstraction layer for external (3rd-party) container runtimes to plug in to. At a high-level, the CRI masks the internal machinery of Kubernetes and exposes a clean documented interface for 3rd-party container runtimes to plug in to.

The CRI is the supported method for integrating runtimes into Kubernetes.

There are lots of container runtimes available for Kubernetes. One popular example is `cri-containerd`. This is a community-based open-source project porting the CNCF `containerd` runtime to the CRI interface. It has a lot of support and is replacing Docker as the preferred container runtime used in Kubernetes.

> **Note**: `containerd` (pronounced "container-dee") is the container supervisor and runtime logic stripped out from the Docker Engine. It was donated to the CNCF by Docker, Inc. and has a lot of community support. Other CRI-compliant container runtimes exist.

Kube-proxy

The last piece of the *node* puzzle is the kube-proxy. This runs on every node in the cluster and is responsible for local networking. For example, it makes sure each node gets its own unique IP address, and implements local IPTABLES or IPVS rules to handle routing and load-balancing of traffic on the Pod network.

Kubernetes DNS

As well as the various control plane and node components, every Kubernetes cluster has an internal DNS service that is vital to operations.

The cluster's DNS service has a static IP address that is hard-coded into every Pod on the cluster, meaning all containers and Pods know how to find it. Every new service is automatically registered with the cluster's DNS so that all components in the cluster can find every Service by name. Some other components that are registered with the cluster DNS are StatefulSets and the individual Pods that a StatefulSet manages.

Cluster DNS is based on CoreDNS (https://coredns.io/).

Now that we understand the fundamentals of masters and nodes, let's switch gears and look at how we package applications to run on Kubernetes.

Packaging apps

For an application to run on a Kubernetes cluster it needs to tick a few boxes. These include:

1. Packaged as a container
2. Wrapped in a Pod
3. Deployed via a declarative manifest file

It goes like this… We write an application service in a language of our choice. We then build it into a container image and store it in a registry. At this point, the application service is *containerized*.

Next, we define a Kubernetes Pod to run the containerized service in. At the kind of high level we're at, a Pod is just a wrapper that allows containers to run on a Kubernetes cluster. Once we've defined a Pod for the container, we're ready to deploy it on the cluster.

Kubernetes offers several objects for deploying and managing Pods. The most common is the *Deployment*, which offers scalability, self-healing, and rolling updates. We define them in a YAML file that specifies things like which image to use and how many replicas to deploy.

Figure 2.5 shows application code packaged as a *container*, running inside a *Pod*, managed by a *Deployment*.

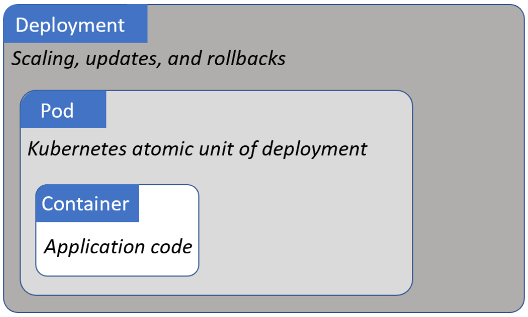

Deployment
Scaling, updates, and rollbacks
Pod
Kubernetes atomic unit of deployment
Container
Application code

Figure 2.5

Once everything is defined in the *Deployment* YAML file, we `POST` it to the cluster as the *desired state* of the application and let Kubernetes implement it.

Speaking of desired state…

The declarative model and desired state

The *declarative model* and the concept of *desired state* are at the very heart of Kubernetes. Take them away and Kubernetes crumbles.

In Kubernetes, the declarative model works like this:

1. Declare the desired state of the application (microservice) in a manifest file
2. POST it to the Kubernetes API server
3. Kubernetes stores this in the cluster store as the application's *desired state*
4. Kubernetes implements the desired state on the cluster
5. Kubernetes implements watch loops to make sure the *current state* of the application doesn't vary from the *desired state*

Let's look at each step in a bit more detail.

Manifest files are written in simple YAML, and they tell Kubernetes how we want an application to look. We call this is the *desired state*. It includes things such as; which image to use, how many replicas to have, which network ports to listen on, and how to perform updates.

Once we've created the manifest, we `POST` it to the API server. The most common way of doing this is with the `kubectl` command-line utility. This POSTs the manifest as a request to the control plane, usually on port 443.

Once the request is authenticated and authorized, Kubernetes inspects the manifest, identifies which controller to send it to (e.g. the *Deployments controller*), and records the config in the cluster store as part of the cluster's overall *desired state*. Once this is done, the work gets scheduled on the cluster. This includes the hard work of pulling images, starting containers, building networks, and starting the application's processes.

Finally, Kubernetes utilizes background reconciliation loops that constantly monitor the state of the cluster. If the *current state* of the cluster varies from the *desired state*, Kubernetes will perform whatever tasks are necessary to reconcile the issue.

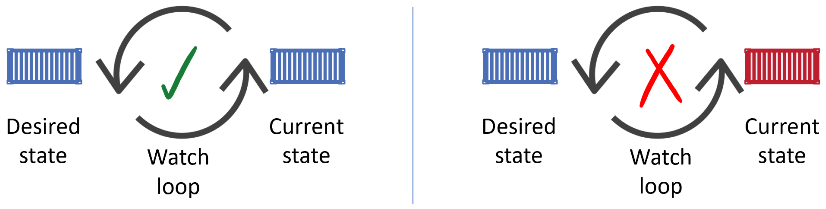


Figure 2.6

It's important to understand that what we've described is the opposite of the traditional *imperative model*. The imperative model is where we issue long lists of platform-specific commands to build things.

Not only is the declarative model a lot simpler than long lists of imperative commands, it also enables self-healing, scaling, and lends itself to version control and self-documentation. It does this by telling the cluster *how things should look*. If they stop looking like this, the cluster notices the discrepancy and does all of the hard work to reconcile the situation.

But the declarative story doesn't end there – things go wrong, and things change. When they do, the ***current state*** of the cluster no longer matches the ***desired state***. As soon as this happens, Kubernetes kicks into action and attempts to bring the two back into harmony.

Let's consider an example.

Assume we have an app with a desired state that includes 10 replicas of a web front-end Pod. If a node that was running two replicas fails, the *current state* will be reduced to 8 replicas, but the *desired state* will still be 10. This will be observed by a reconciliation loop and Kubernetes will schedule two new replicas on other nodes in the cluster.

The same thing will happen if we intentionally scale the desired number of replicas up or down. We could even change the image we want to use. For example, if the app is currently using `v2.00` of an image, and we update the desired state to use `v2.01`, Kubernetes will notice the discrepancy and go through the process of updating all replicas so that they are using the new image version specified in the new *desired state*.

To be clear. Instead of writing a long list of commands to go through the process of updating every replica to the new version, we simply tell Kubernetes we want the new version, and Kubernetes does the hard work for us.

Despite how simple this might seem, it's extremely powerful. It's also at the very heart of how Kubernetes operates. We give Kubernetes a declarative manifest that describes how we want an application to look. This forms the basis of the application's desired state. The Kubernetes control plane records it, implements it, and runs background reconciliation loops that constantly check what is running is what we've asked for. When current state matches desired state, the world is a happy place. When it doesn't, Kubernetes gets busy fixing it.

Pods

In the VMware world, the atomic unit of scheduling is the virtual machine (VM). In the Docker world, it's the container. Well… in the Kubernetes world, it's the ***Pod***.

It's true that Kubernetes runs containerized apps. However, you cannot run a container directly on a Kubernetes cluster – containers must **always** run inside of Pods.

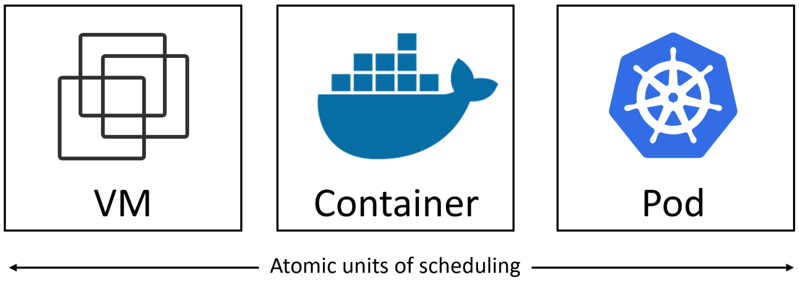


Figure 2.7

Pods and containers

The very first thing to understand is that the term *Pod* comes from a *pod of whales* – in the English language we call a group of whales a *pod of whales*. As the Docker logo is a whale, it makes sense that we call a group of containers a *Pod*.

The simplest model is to run a single container per Pod. However, there are advanced use-cases that run multiple containers inside a single Pod. These *multi-container Pods* are beyond the scope of what we're discussing here, but powerful examples include:

- Service meshes
- Web containers supported by a *helper* container that pulls the latest content
- Containers with a tightly coupled log scraper

The point is, a Kubernetes Pod is a construct for running one or more containers. Figure 2.8 shows a multi-container Pod.

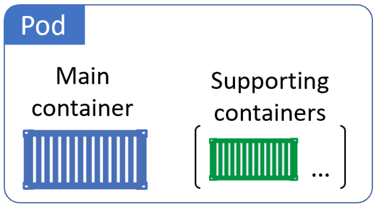


Figure 2.8

Pod anatomy

At the highest-level, a *Pod* is a ring-fenced environment to run containers. The Pod itself doesn't actually run anything, it's just a sandbox for hosting containers. Keeping it high level, you ring-fence an area of the host OS, build a network stack, create a bunch of kernel namespaces, and run one or more containers in it. That's a Pod.

If you're running multiple containers in a Pod, they all share the **same environment**. This includes things like the IPC namespace, shared memory, volumes, network stack and more. As an example, this means that all containers in the same Pod will share the same IP address (the Pod's IP). This is shown in Figure 2.9.

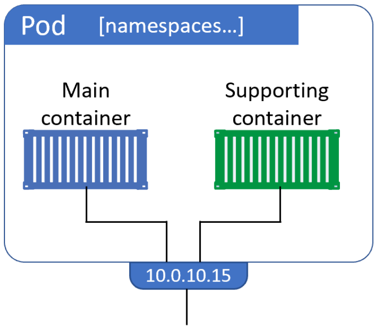


Figure 2.9

If two containers in the same Pod need to talk to each other (container-to-container within the Pod) they can use ports on the Pod's `localhost` interface as shown in Figure 2.10.

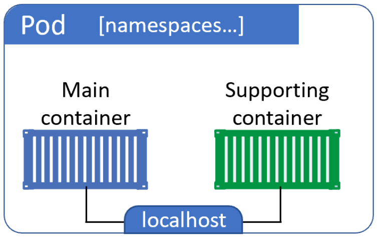


Figure 2.10

Multi-container Pods are ideal when you have requirements for tightly coupled containers that may need to share memory and storage. However, if you don't **need** to tightly couple your containers, you should put them in their own Pods and loosely couple them over the network. This keeps things clean by having each Pod dedicated to a single task.

Pods as the unit of scaling

Pods are also the minimum unit of scheduling in Kubernetes. If you need to scale your app, you add or remove Pods. You **do not** scale by adding more containers to an existing Pod. Multi-container Pods are only for situations where two different, but complimentary, containers need to share resources. Figure 2.11 shows how to scale the `nginx` front-end of an app using multiple Pods as the unit of scaling.

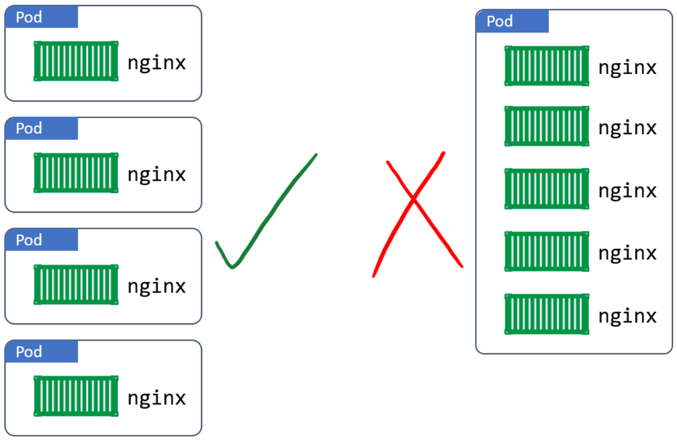


Figure 2.11 - Scaling with Pods

Pods - atomic operations

The deployment of a Pod is an atomic operation. This means that a Pod is either entirely deployed, or not deployed at all. There is never a situation where a partially deployed Pod will be servicing requests. The entire Pod either comes up and is put into service, or it doesn't, and it fails.

A single Pod can only be scheduled to a single node. This is also true of multi-container Pods – all containers in the same Pod will run on the same node.

Pod lifecycle

Pods are mortal. They're created, they live, and they die. If they die unexpectedly, we don't bring them back to life. Instead, Kubernetes starts a new one in its place. However, even though the new Pod looks, smells, and feels like the old one, it isn't. It's a shiny new Pod with a shiny new ID and IP address.

This has implications on how we should design our applications. Don't design them so they are tightly coupled to a particular instance of a Pod. Instead, design them so that when Pods fail, a totally new one (with a new ID and IP address) can pop up somewhere else in the cluster and seamlessly take its place.

Deployments

We normally deploy Pods indirectly as part of something bigger. Examples include; *Deployments*, *DaemonSets*, and *StatefulSets*.

For example, a Deployment is a higher-level Kubernetes object that wraps around a particular Pod and adds features such as scaling, zero-downtime updates, and versioned rollbacks.

Behind the scenes, they implement a controller and a watch loop that is constantly observing the cluster making sure that current state matches desired state.

Deployments have existed in Kubernetes since version 1.2 and were promoted to GA (stable) in 1.9. You'll see them a lot.

Services

We've just learned that Pods are mortal and can die. However, if they're managed via Deployments or DaemonSets, they get replaced when they fail. But replacements come with totally different IPs. This also happens when we perform scaling operations – scaling up adds new Pods with new IP addresses, whereas scaling down takes existing Pods away. Events like these cause a lot of *IP churn*.

The point we're making is that **Pods are unreliable**, which poses a challenge… Assume we've got a microservices app with a bunch of Pods performing video rendering. How will this work if other parts of the app that need to use the rendering service cannot rely on the rendering Pods being there when they need them?

This is where *Services* come in to play. **Services provide reliable networking for a set of Pods**.

Figure 2.12 shows the uploader microservice talking to the renderer microservice via a Kubernetes Service. The Kubernetes Service is providing a reliable name and IP, and is load-balancing requests to the two renderer Pods behind it.

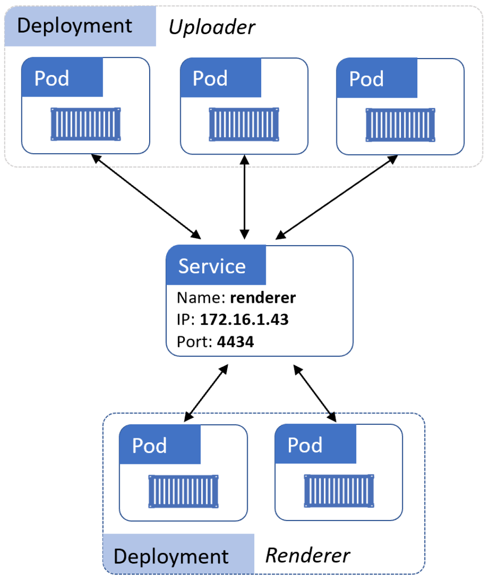


Figure 2.12

Digging in to a bit more detail. Services are fully-fledged objects in the Kubernetes API – just like Pods and Deployments. They have a front-end that consists of a stable DNS name, IP address, and port. On the back-end, they load-balance across a dynamic set of Pods. Pods come and go, the Service observes this, automatically updates itself, and continues to provide that stable networking endpoint.

The same applies if we scale the number of Pods up or down. New Pods are seamlessly added to the Service, whereas terminated Pods are seamlessly removed.

That's the job of a Service – it's a stable network abstraction point that provides TCP and UDP load-balancing across a dynamic set of Pods.

As they operate at the TCP and UDP layer, Services do not possess application intelligence and cannot provide application-layer routing. For that, you need an Ingress, which understands HTTP and provides host and path-based routing.

Connecting Pods to Services

Services use *labels* and a *label selector* to know which set of Pods to load-balance traffic to. The Service has a *label selector* that is a list of all the *labels* a Pod must possess in order for it to receive traffic from the Service.

Figure 2.13 shows a Service configured to send traffic to all Pods on the cluster possessing the following three labels:

- zone=prod

- env=be
- ver=1.3

Both Pods in the diagram have all three labels, so the Service will load-balance traffic to them both.

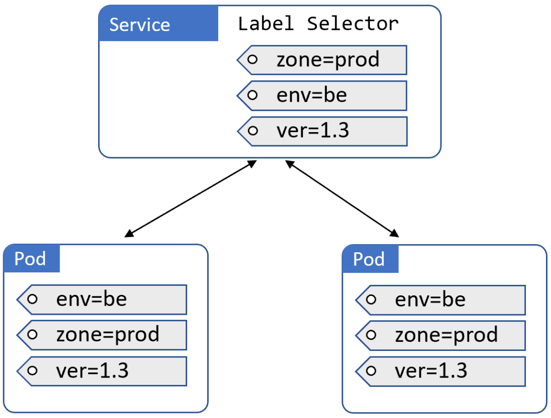


Figure 2.13

Figure 2.14 shows a similar setup. However, an additional Pod, on the right, does not match the set of labels configured in the Service's label selector. This means the Service will not load balance requests to it.

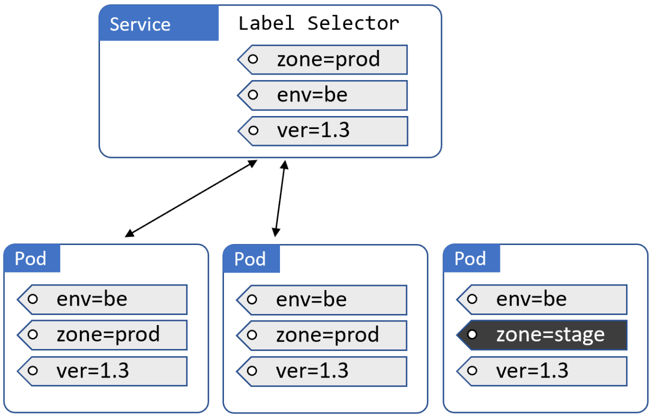


Figure 2.14

One final thing about Services. They only send traffic to **healthy Pods**. This means a Pod that is failing health-checks will not receive traffic from the Service.

That's the basics. Services bring stable IP addresses and DNS names to the unstable world of Pods.

Chapter summary

In this chapter, we introduced some of the major components of a Kubernetes cluster.

The masters are where the control plane components run. Under-the-hood, there's a combination of several system-services, including the API Server that exposes the public REST interface. Masters make all of the deployment and scheduling decisions, and multi-master HA is important for production-grade environments.

Nodes are where user applications run. Each node runs a service called the `kubelet` that registers the node with the cluster and communicates with the control plane. This includes receiving new work tasks and maintaining a reporting channel. Nodes also have a container runtime and the `kube-proxy` service. The container runtime, such as Docker or containerd, is responsible for all container-related operations. The `kube-proxy` is responsible for networking on the node.

We also talked about some of the major Kubernetes API objects such as Pods, Deployments, and Services. The Pod is the basic building-block. Deployments add self-healing, scaling and updates. Services add stable networking and load-balancing.

Now that we know the basics, we're going to start getting into the detail.