Contents

Introduction **1**
- History . . . 2
- Meaning . . . 2
- Kubernetes and Docker . . . 2
- Kubernetes and Docker swarm . . . 3
- Kubernetes as the operating system of the cloud . . . 3
- Principles of operation . . . 3
  - Kubernetes from 40k feet . . . 3
  - Control plane and worker nodes . . . 4
  - Kubernetes DNS . . . 6
  - Packaging apps for Kubernetes . . . 6
  - Declarative model . . . 7
  - Pods . . . 8
  - Deployments . . . 9
  - Service objects . . . 9
- Getting Kubernetes . . . 10
  - Kubernetes playground . . . 10
  - Hosted Kubernetes . . . 10
  - DIY Kubernetes . . . 10

Introduction

`Kubernetes` is an application `orchestrator`, for the most part it orchestrates containerized cloud native micro services. An orchestrator is a system that deploys and manages apps, it can deploy your app and dynamically respond to changes, for example k8s, can:

1. deploy you app
2. scale it up and down dynamically based on demand
3. self heal when things break
4. perform zero downtime rolling updates and rollbacks
5. many, many more things

`What really is containerised app` - it is an app that runs in a container, before we had containers, apps ran on physical servers or virtual machines, containers are just the next iteration of how we package and run apps, as such they are faster more lightweight and more suited to modern business requirements than servers and virtual machines

`What is a cloud native app` - it is one that is designed to meet the cloud like demands of auto scaling self healing rolling updates, rollbacks and more, it is important to be clear that cloud native apps are not apps that will only run in the public cloud, yes they absolutely can run on a public clouds, but they can also run anywhere that you have k8s even your on premise datacenter.

`What are microservice apps` - is built from lots of independent small specialised parts that work together to form a meaningful app. For example you might have an e-commerce app that comprises all of the following small components

1. web front end
2. catalog service
3. shopping cart
4. authentication service
5. logging service

6. persistent store

Each of these individual services is called a micro service, typically each is coded and owned by a different team, each can have its own release cycle and can be scaled independently, for example you can patch and scale the logging micro service without affecting any of the others. Building apps this way is vital for cloud native features, For the most part, each microservice runs as a container, assuming that e-commerce app with the 6 microservice there would be one or more web front end containers one or more catalog containers one or more shipping cart containers etc. With all of this in mind

`Kubernetes deploys and manages (orchestrates)apps that are packaged and run as containers (containerized)and that are built in ways (cloud native microservice)that allows them to scale, self heal and be updated in line with modern cloud like requirements.`

History

Since amazon brought the Amazon Web Services, the world changed, since then everyone is playing catch-up. One of the companies trying to catch up was Google. It has its own very good cloud and needs a way to abstract the value of AWS and make it easier for potential customers to get off AWS and into their cloud. Google also has a lot of experience working with containers at scale, for example huge google apps such as Search and Gmail have been running at extreme scale on containers for a lot of years, since way before Docker brought us easy to use containers. To orchestrate and manage these containerised apps, Google had a couple of in-house proprietary systems called Borg and Omega. Well Google took the lessons leaned from these systems, and created a new platform called Kubernetes, and donated it to the newly formed `Cloud Native Computing Foundation (CNCF)` in 2014, as an open source project. Kubernetes enables two things Google and the rest of the industry needs

1. It abstracts underlying infrastructure such as `AWS`
2. It makes it easy to move apps on and off clouds

Since its introduction in 2014, Kubernetes has become the most important cloud native technology on the planet. Like many of the modern cloud native projects, it's written in Go, it is built in the open on GitHub it is actively discussed on the IRC channels, you can follow it everywhere on social media, there are also regular conferences and regular meetups

Meaning

The name Kubernetes comes from the Greek word meaning Helmsman - the person steers the seafaring ship. This theme is reflected in the logo which is the wheel (helm control) of a sea faring ship. You will often see it shortened to k8s - pronounced `kate`. The number 8 replaces the 8 characters between the K and the S in the name, and that is people sometimes joke that Kubernetes has girlfriend named Kate

Kubernetes and Docker

Kubernetes and Docker are two complementary technologies, Docker has tools that build and package apps as container images. It can also run containers, Kubernetes can't do either of those things, Instead, Kubernetes operates at a higher level providing orchestration services such as self-healing, scaling and updating. It is common practice to use Docker for build time tasks such as packaging apps as containers, but then use a combination of Kubernetes and Docker to run them. In this model, Kubernetes preforms the high level orchestration tasks, while Docker performs the low level tasks such as starting and stopping containers.

Assume you have a Kubernetes cluster with 10 nodes, to run your production app. Behind the scenes each cluster node is running Docker as its container runtime. This means Docker is the low-level technology that starts and stops the containerised apps. Kubernetes is the higher level technology that looks after the

bigger picture, such as deciding which nodes to run containers on, deciding when to scale up or down, and execute updates. Docker is not the only container runtime that Kubernetes supports, it also does support `gVisor`, `containerd` and `kata`. Kubernetes has features which abstract the container runtime and make it interchangeable

1. The container runtime interface (CRI) - is an abstraction layer that standardizes the way 3rd party container runtimes work with Kubernetes
2. Runtime Classes allows you to create different classes of runtimes. For example the `gVisor` or `Kata` Containers runtimes might provide better workload isolation than the Docker and `containerd` runtimes

Kubernetes and Docker swarm

In 2016 and 2017 we had the orchestrator wards, where `Docker Swarm`, `Mesosphere DCOS`, and `Kubernetes` completed to become the de-facto container orchestrator. To cut a long story short, Kubernetes WON.

There is a good chance you will hear people talk about how Kubernetes relates to Google's Borg and Omega systems, as previously mentioned, Google has been running containers at scale for a long time - apparently crunching through billions of containers a week. So yes, Google has been running things like `Search`, `Gmail`, and `GFS` on lots of containers for a very longs time. Orchestrating these containerised apps was the job of a couple of in-house technologies called Borg and Omega. So it is not a huge stretch to make the connection with Kubernetes - all three are in the game or orchestration or containers at scale, and they are all related to Google.

`Kubernetes is not an opened source version of Borg and Omega, it shares common traits and technologies, common DNA, if you wish, but the Borg and Omega are still proprietary closed source projects`

They are all separate but all three are related, in fact, some of the people who built Borg and Omega were and still are involved with Kubernetes. So although Kubernetes was built from scratch, it leverages much of what was learned at Google with Omega and Borg

Kubernetes as the operating system of the cloud

Kubernetes has merged as the de-facto platform for deploying and managing cloud native apps, in many ways it is like an operating system for the cloud. In the same way that Linux abstracts the hardware differences between server platforms, Kubernetes abstracts the differences between the different private and public clouds. Net result is that as long as you are running Kubernetes, it does not matter if the underlying systems are on premises, in your own datacenter, edge devices or in the public cloud or domain.

Principles of operation

This sub chapter will summarize the major components required to build a Kubernetes cluster and deploy an app. The aim is to give you an overview of the major concepts, so you do not worry if you do not understand everything straight away, most things will be covered again

Kubernetes from 40k feet

At the highest level, Kubernetes is two things - A cluster to run apps on & an orchestrator of cloud native microservice apps.

Kubernetes as a cluster Kubernetes is like any other cluster - a bunch of machines to host apps on. We call these machines nodes, and they can be physical servers, virtual machines, cloud instances, Raspberry Pis, and more. A Kubernetes cluster is made of a `control plane` and `nodes`. This control plane exposes the API,

has scheduler for assigning work, and records the state of of the cluster and apps in a persistent store. Nodes are where user apps run. It can be useful to think of the `control plane` as the brains of the cluster and the nodes as the muscle. In this analogy the `control plane` is the brains because it implements the clever features such as scheduling, auto-scaling and zero-downtime rolling updates.

Kubernetes as orchestrator Orchestrator is just a fancy word for a system that takes care of deploying and managing apps. If we take a quick analogy from the real world, a football team is made of individuals. Every individual is different 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 she or he gives everyone a position and organizes them into a team with a purpose. The coach also makes sure that the team keeps its formation sticks to the game-plan and deals with any injuries and other changes in the circumstances. Well microservices apps on Kubernetes are the same.

You start out with lots of individual specialised microservices. Some serve web pages, some do authentication some perform searches, other persist data. Kubernetes comes along - like the coach, organizes everything into a useful app and keeps things running smoothly. It even responds to events and other changes in the circumstances - auto-scaling, updating, rolling release etc.

When we start out with an app, package it as a container then give it to the cluster - Kubernetes. The cluster is made up of one or more control plane nodes and a bunch of worker nodes. As already stated, control plane nodes implement the cluster intelligence, worker nodes are where user apps run.

Control plane and worker nodes

As previously mentioned a Kubernetes cluster is made of control plane nodes and worker nodes. These are Linux hosts that can be virtual machines, bare metal servers in your datacenter or basement, instances in a private or public cloud. You can even run Kubernetes on ARM and IoT devices

The control plane A Kubernetes control plane node is a server running collection on system services that make up the control plane of the cluster. Sometimes we call those Masters, Heads or Head nodes. The simplest setups run a single control plane node. However this is only suitable for labs and test environments, for production environments multiple control plane nodes configured for high availability is vital. Also considered a good practice not to run user apps on control plane nodes. This frees them up to concentrate entirely on managing the cluster.

The API server The API server is the Grand Central of Kubernetes. All communication, between all components must go through the API server. It is important to understand that internal system components as well as external user components all communicate via the API server - all roads lead to the API server

It exposes a RESTful API that you POST YAML configuration files to over HTTPS. These YAML files which we sometimes call manifests describe the desired state of an app. This desired state includes things like which container image to use, which ports to expose and how many Pod replicas to run. All requests to API server are subject to authentication and authorization checks. Once these are done, the configuration in the YAML file is validated, persisted to the cluster store, and work is scheduled to the server

The cluster store The cluster store is the only stateful part of the control plane and persistently stores the entire configuration and state of the cluster. As such it is vital components of every Kubernetes cluster - no cluster store, no cluster. The cluster store is currently based on `etcd`, a popular distributed database. As it is the single source of truth for a cluster, you should run between 3-5 replicas of the `etcd` service for high-availability and you should provide adequate ways to recover when things to bad. A default installation of Kubernetes installs a replica of the cluster store on every control plane node and automatically configures the high availability (HA)

On the topic of availability, `etcd` prefers consistency over availability. This means it does not tolerate split brains and will halt updates to the cluster in order to maintain consistency. However, if this happens user apps should continue to work, you just won't be able to update the cluster configuration.

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

The controller manager and controllers The controller manager implements all the background controllers that monitor cluster components and respond to events. Architecturally, it is a controller of controllers, meaning it spawns all the independent controllers and monitors them. Some of the controllers include the Deployment controller, the `StatefulSet` controller and the `ReplicaSet` controller. Each one is responsible for a small subset of cluster intelligence and runs as a background watch loop constantly watching the API Server for changes.

`The aim of the game is to ensure the observed state of the cluster matches the desired state. The logic implemented by each controller is as follows, and is at the heart of Kubernetes and declarative design patterns`

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

Each controller is also extremely specialized and only interested in its own little corner of the Kubernetes cluster. No attempts is made to over complicate design by implementing awareness of other parts of the system each controller takes care of its own business and leaves everything else alone. This is key to the distributed design of Kubernetes and adheres to the Unix philosophy.

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

When identifying nodes capable of running a task, the scheduler performs various predicate checks. These include is the node tainted, are there any affinity or anti affinity rules, is the required network port available on the node does it have sufficient available resources etc. Any node incapable of running the task is ignored and those remaining are ranked according to things such as does it already have the required image how much free resources does it have, how many tasks is it currently running. Each is worth points and the node with the most points is selected to run the task. If the scheduler does not find a suitable node, the task is not schedule and gets marked as pending. The scheduler is not responsible for running tasks just picking the nodes to run them. A task is normally a Pod/container.

The cloud controller manager If one is running cluster on a supported public cloud platform such as `AWS, Azure, GCP, or Linode` your control plane will be running a cloud controller manager. Its job is to facilitate integrations with cloud services, such as instances , load-balancers, and storage. For example if your app asks for an internet facing load-balancer the cloud controller manager provisions a load-balancer from your cloud and connects it to your app.

Control plane summary Kubernetes control plane nodes are servers that run the cluster's control plane services. These services are the brains of the cluster where all the control and scheduling decisions happen. Behind the scenes, these services include the API server, the cluster store, scheduler and specialised controller.

The API server is the front end into the control plane and all instructions and communication pass through it. By default it exposes a RESTful endpoint on port 443.

Worker nodes Nodes are servers that 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 work assignments
3. Report back to the control plane

Kubelet The kubelet is main Kubelet agent and runs on every cluster node. In fact, it is common to use the terms node and kubelet interchangeably. When you join a node to a cluster the process installs the kubelet which is then responsible for registering it with the cluster. This process registers the node's CPU, memory, and storage into the wider cluster pool.

One of the main jobs of the kubelet is to watch the API server for new work tasks. Any time it sees one, it executed the task and maintains a reporting channel back to the control plane. If a kubelet can't run a task, it reports back to the control plane and lets the control plane decide what actions to take. For example if a kubelet can not execute a task, it 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 a container related task - things like pulling images and starting and stopping containers. In the early days, Kubernetes had native support for Docker, More recently it has moved to a plugin model called the Container Runtime Interface (CRI). 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 into. Kubernetes is dropping support for Docker as a container runtime, this is because Docker is bloated and does not support the CRI (requires a shim instead). `containerd` is replacing it as the most common container runtime on Kubernetes

```
containerd is the container supervisor and runtime logic stripped out from docker engine.
It was donated to the CNCF by Docker Inc, and has a lot of community support. Other CRI
container runtimes also exist.
```

Kube-proxy The last piece of the node puzzle is the kube proxy. This runs on every node and is responsible for local cluster networking. It ensures each node gets its own unique IP address, and it implements local `iptables` or `IPVS` rules to handle routing and load balancing of traffic on the Pod network. More on all of this later on in other chapters down below.

Kubernetes DNS

As well as the various control plane and node components, every Kubernetes cluster has an internal `DNS` service, that is vital to service discovery. The cluster's `DNS` service has a static IP address that is hard coded into every Pod on the cluster. This ensures every container and Pod can locate it and use it for discovery. Service registration is also automatic. This means apps do not need to be coded with the intelligence to register with Kubernetes service discovery. Cluster `DNS` is based on open source `CoreCNS` project.

Packaging apps for Kubernetes

An app needs to tick a few boxes to run on a Kubernetes cluster. These include:

1. Packages as a container image
2. Wrapped in a pod
3. Deployed via a declarative manifest file

It goes like this. You write an application microservice in a language of your choice. Then you build it into a container image and store it in a registry. At this point the app service is containerized. Next you define a Kubernetes Pod to run the containerized app. At the kind of high lever we are at, a Pod is just a wrapper that allows a container to run on a Kubernetes cluster. Once you have defined the pod, you are ready to deploy the app to Kubernetes.

While it is possible to run static Pods like this on a Kubernetes cluster, the preferred model is to deploy all Pods via a higher level controllers. The most common controller is the Deployment. It offers scalability, self healing and rolling updates for stateless apps. You define Deployments in YAML manifest files that specify things how many replicas to deploy and how to perform updates. Once everything is defined in the Deployment YAML file, you can use the Kubernetes command line tools to post it to the API server as the desired state of the app, and Kubernetes will implement it

Declarative model

The declarative model and the concept of desired state are at the very heart of Kubernetes. So it is vital you understand them. In Kubernetes the declarative model works like this.

1. Declare the desired state of an app, microservice in a manifest file
2. Post it to the API server
3. Kubernetes stores it in the cluster store as the app's desired state
4. Kubernetes implements the desired state in the cluster
5. A controller makes sure the observed state of the app does not vary from the desired state

Manifest files are written in simple YAML and tell Kubernetes what an app should look like. This is called desired state. It includes things such as which image to use, how many replicas to run, which network ports to listen on, and how to perform updates.

Once you have created the manifest you post it to the API server. The easiest way to do this is with the kubectl command line utility. This sends the manifest to the control plane as an HTTPS POST request (on port 443)

Once the request is authenticated and authorized. Kubernetes inspects the manifest, identifies which controller to send it to (i.e. Deployments controller) and records the configuration in the cluster store as part of the overall desired state. Once this is done any required work tasks get scheduled to the cluster nodes where the kubelet co-ordinates the hard work of pulling images starting containers attaching to networks, and starting app processes.

Finally controllers run as background reconciliation loops that constantly monitor the state of things, if the observed state deviates from the desired state, Kubernetes performs the tasks which are necessary to reconcile the differences and bring the observed state back in sync with the desired state

`It is important to understand that what we have described is the opposite of the traditional imperative model. The imperative model is where you write long scripts of platform specific commands to build and monitor things, not only is the declarative model a lot simpler than long scripts with lots of imperative commands, it also enables self-healing, scaling and lends itself to version control and self-documentation. It does all of this by telling the cluster how thing should look like. If they start to look different, the appropriate controller notices the discrepancy and does all the hard work to reconcile the situation.`

Example Assume you have an app with a desired state that includes 10 replicas of a web front end Pod. If a node running two replicas fails, the observed state will be reduced to 8 replicas but desired state will still be 10. This will be observed by a controller and Kubernetes will schedule two new replicas to bring the total back up to 10. The same thing will happen if you intentionally scale the desired number of replicas up or

down. You could even change the image you want to use (this is called a `rollout`). For example if the app is currently using `v2.00` of an image and you update the desired state to specify `v2.01` the relevant controller will notice the difference and go through the process of updating the cluster so all 10 replicas are running the new version.

To be clear. Instead of writing a complex script to step through the entire process of updating every replica to the new version, you simply tell Kubernetes you want the new version and Kubernetes does the hard work for you.

Pods

In the VMware world the atomic unit of scheduling is the virtual machine. In the docker world it is the container, in the Kubernetes world it is the Pod. It is true that Kubernetes runs containerized apps. However Kubernetes demands that every container runs inside a pod.

```
Pods are objects in the Kubernetes API, so we capitalize the first letter. This adds
clarity and the official Kubernetes docs are moving towards this standard.
```

Pods & 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, Kubernetes ran with the whale concept and that is why we have Pods. The simplest model is to run a single container in every Pod. This is why we often use the term Pod and container interchangeably. However there are advanced use cases that run multiple containers in a single Pod, Powerful examples of multi container Pods include:

- Service meshes
- Containers with a tightly coupled log scraper
- Web containers supported by a helper container pulling updated content

The point is that a Kubernetes Pod is a construct for running one or more containers.

Pod anatomy At the highest level a Pod is ring fenced environment to run containers. Pods themselves do not actually run apps, apps always run in containers, the Pod is just a sandbox to run one or more containers. Keeping it high level, Pods ring fence an area of the host OS, build a network stack, create a bunch of kernel namespaces and run one or more containers

If you are running multiple containers in a Pod they all share the same Pod environment. This includes the network stack, volumes `IPC` namespace, shared memory and more. As an example this means all containers in the same Pod will share the same IP address. If two containers in the same Pod want to talk to each other, they can use the Pod's `localhost` interface.

Multi container Pods are ideal when you have requirements for tightly coupled containers that may need to share memory and storage. However if you do not need to tightly couple 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. However it creates a lot of potentially `un-encrypted` network traffic. One must seriously consider using a service mesh to secure traffic between Pods and app services

Pods as unit of scaling Pods are also the minimum unit of scheduling in Kubernetes If you need to scale an app, you add or remove Pods. You do not scale by adding more containers to existing Pods. Multi-container Pods are only for situations where to different but complimentary containers need to share resources.

Pods atomic operations The deployment of a Pod is an atomic operation. This mean a Pod is only ready for service when all its containers are up and running. The entire Pod either comes up and is put into service or it does not and it fails. A single Pod can only be schedules to a single node - you can not schedule a single

Pod across multiple nodes. This is also true of multi container Pods - all containers in the same Pod run on the same node.

Pod lifecycle Pods are mortal. The are created they live and they die. If they die unexpectedly you do not have to 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 is not. It is a shiny new Pod with a shiny new ID and IP address.

This has implications on how you design you app. Do not design them to be tightly coupled to particular instance of a Pod. Instead design them so that when Pods fail a totally new one can pop up somewhere else in the cluster and seamlessly take its place

Pod immutabilitiy Pods are also immutable this means you do not change them once they are running. Once a Pod is running you never change its configuration. If you need to change or update it, you replace it with a new Pod instance running the new configuration. When we have talked about updating Pods, we have really meant delete the old one and replace it with a new one having the new configuration

Deployments

Most of the time you will deploy Pods, indirectly via a higher level controllers. Examples of higher level controllers include `Deployments`, `DeamonSets` and `StatefulSets`. As an example a Deployment is a higher level Kubernetes object that wraps around a Pod and adds features such as self-healing, scaling, zero-downtime rollouts, and versioned rollbacks.

Behind the scenes, Deployments, DeamonSets and StatefulSets are implemented as controllers that run as watch loops constantly observing the cluster making sure observed state matches desired state.

Service objects

Since we have already mentioned that Pods can die, they are also managed via a higher level controllers and get replaced when they die or fail. But replacements come with a totally different IP addresses. This also happens with rollouts and scaling operations. Rollouts replace old Pods with new ones with new IPs. Scaling up adds new Pods with new IP addresses, whereas calling down takes existing Pods away. Events like these cause a lot of IP churn. The point we are making is that Pods are unreliable and this poses a challenge. Assume you have got a microservice app with a bunch of Pods performing video rendering. How will this work if other parts of the app that use the rendering service can not rely on rendering Pods being there when needed. This is where Services come in to play. They provide reliable networking for a set of Pods.

Services are fully fledged objects in the Kubernetes API - just like Pods and Deployments. They have a front end consisting of a DNS name, IP address and port. On the back end they load balance traffic across a dynamic set of Pods. As pods come and go, the Service observes this, automatically updates itself, and continues to provide that stable networking endpoint. The same applies if you scale the number of Pods up or down. New Pods are seamlessly added to the Service and receive traffic. Terminated Pods are seamlessly removed from the Service and will not receive traffic. That is the job of a Service - it is a stable network abstraction point that provides TCP and UDP load balancing across a dynamic set or number (replicas) of Pods/containers

As they operate at the TCP and UDP layer, they do not posses application intelligence, this means they can not provide app layer host and path routing. For that you need an Ingress which understand HTTP and provides host and path based routing.

```
Services bring stable IP addresses and DNS names to the unstable world of Pods, they
are the abstraction layer, that allows other Services, Pods or Containers to communicate
without having to worry about the fact that a target Pod can die
```

Getting Kubernetes

This section will describe a few fast and quick ways to obtain Kubernetes. Will also introduce you to kubectl, the Kubernetes command line tool.

Kubernetes playground

Playgrounds are the quickest and easiest way to get Kubernetes but they do not function for production, Popular examples include `Play with Kubernetes, Katakoda, Docker Desktop, minikube, k3d and more`

Hosted Kubernetes

All of the major cloud platforms offer a hosted Kubernetes service. This is a model where you outsource a bunch of Kubernetes infrastructure to your cloud provider, letting them take care of things like high availability, performance and updates. Of course not all hosted Kubernetes solutions are equal and even though your cloud provider is managing a lot of the infrastructure for you, the ultimate responsibility remains with you.

DIY Kubernetes

By far the hardest way to get a Kubernetes cluster is to build it yourself. Yes, installations such as these are possible are a lot easier now than they used to be, but they can still be hard. However they provide most flexibility and give you ultimate control - which can be good for learning