Introduction to Kubernetes

- Why Orchestration
- What is Kubernetes
- Why Kubernetes
- Important features of Kubernetes
- Kubernetes Architecture
- Overview of Kubernetes Objects

Introduction to Kubernetes

Why Orchestration

Containerization has brought a lot of flexibility for developers in terms of managing the deployment of the applications. However, the more granular the application is, the more components it consists of and hence requires some sort of management for those.

One still needs to take care of scheduling the deployment of a certain number of containers to a specific node, managing networking between the containers, following the resource allocation, moving them around as they grow and much more.

Nearly all applications nowadays need to have answers for things like

- Replication of components
- Load balancing
- Auto-scaling
- Rolling updates
- Logging across components
- Monitoring and health checking
- Service discovery
- Security

Container Orchestration

- The process of organizing **multiple containers and managing them as needed** is known as **container orchestration**.
- Container orchestration automates the **deployment, management, scaling, and networking** of containers across the cluster.
- It is focused on managing the life cycle of containers.
- Enterprises that need to deploy and manage hundreds or thousands of Linux® containers and hosts can benefit from container orchestration.
- Examples: Docker Swarm, ECS, **Kubernetes**, Mosospehere, Azure Service Fabric

What is Kubernetes?

Kubernetes, is a container orchestration technology used to orchestrate the deployment and management of hundreds and thousands of containers in clustered environment. It could be thought of as the **operating system** for **microservices applications**, it is the platform that applications run on, just as desktop applications run on MacOS, Windows, or Linux.

Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.

- The name **Kubernetes** originates from Greek, meaning ***helmsman*** *(who steers ship or boat)* or *pilot*, and is the root of *governor* and cybernetic (theory or study of communication and control).
- **K8s** is an abbreviation derived by replacing the 8 letters "ubernete" with "8".
- It is a **platform** designed to completely manage the life cycle of containerized applications and services using methods that provide **predictability, scalability, and high availability**.
- Written in **Golang**.
- It orchestrates **computing**, **networking**, and **storage** infrastructure on behalf of user workloads.
- It is easier to **deploy**, **scale**, and **manage** applications.

History

- 2003 – Started at Google as The Borg System to manage Google Search. They needed a sane way to manage their large-scale container clusters! But it was still very primitive compared to what we have today. It becomes big and rather messy, with many different languages and concepts due to it's organic growth.
- 2013 – Docker hits the scene and really revolutionizes computing by providing build tools, image distribution, and runtimes. This makes containers user friendly and adoption of containers explodes.
- 2014 – 3 Google engineers decide to build a next generation orchestrator that takes many lessons learned into account, built for public clouds, and open sourced. They build Kubernetes. Microsoft, Red Hat, IBM, and Docker join in.
- 2015 – Cloud Native Computing Foundation is created by Google and the Linux Foundation. More companies join in. Kubernetes 1.0 is released, followed by more major upgrades that year. KubeCon is launched.
- 2016 – K8S goes mainstream. Many supporting products are introduced including Minikube, kops, Helm. Rapid releases of big features. More companies join in and the community of passionate people explodes.
- 2017 to now – Kubernetes becomes the dominant orchestration system and de-facto standard for Docker microservices. K8S now has fully managed services by all major cloud providers. Handsome and talented instructors travel the country preaching K8S.

Kubernetes is:

- a container platform.
- a microservices platform.
- a portable cloud platform.

Certified Kubernetes Distributions

- Cloud Managed: EKS by AWS, AKS by Microsoft and GKE by Google
- Self Managed: OpenShift by Redhat and Docker Enterprise
- Local dev/test: Micro K8s by Canonical, Minikube, Docker Desktop
- Vanilla Kubernetes: The core Kubernetes project(baremetal), Kubeadm
- Special builds: K3s by Rancher, a light weight K8s distribution for Edge devices

Why Kubernetes?

- Decades of experience running at **Google**.
- Great momentum in terms of activities & contribution at its **Open Source Project**.
- Support of **multiple OS** and infrastructure software vendors.
- Rate at which features are being released. Number of features available.
- Modern tooling, have CLI and management REST API support.

Important features of Kubernetes .

- **Simplifies Application Deployment**: Deploy a specified number of containers to a specified host and keep them running in a desired state.
- **Rolling updates:** A rollout is a change to a deployment. Kubernetes lets you initiate, pause, resume, or rollback rollouts.
- **Service discovery**: Kubernetes can automatically expose a container to the internet or to other containers using a DNS name or IP address.
- **Storage provisioning:** Set Kubernetes to mount persistent local or cloud storage for your containers as needed.
- **Load balancing and scaling:** When traffic to a container spikes, Kubernetes can employ load balancing and scaling to distribute it across the network to maintain stability.
- **Self-healing for high availability**: When a container fails, Kubernetes can restart or replace it automatically; it can also take down containers that don't meet your health-check requirements.
- **DevOps Support:** Continues development, integration and deployment using DevOps tools and services.

Kubernetes vs. Docker

Kubernetes is neither an alternative nor a competitor to Docker (it is an alternative to **Docker Swarm**).

If we have adopted Docker and are creating large-scale Docker-based container deployments, Kubernetes orchestration is a next logical step for managing these workloads.

Kubernetes	**Docker Swarm**
Installation is complicated; but once setup, the cluster is very strong	Installation is very simple; but cluster is not very strong.
GUI is the Kubernetes Dashboard	There is no GUI
Highly scalable & scales fast	Highly scalable & **scales 5x faster** than Kubernetes
Kubernetes can do auto-scaling	Docker Swarm cannot do autoscaling
Can deploy Rolling updates & does automatic Rollbacks	Updates & Rollbacks Can deploy Rolling updates, but not automatic Rollbacks
Can share storage volumes only with other containers in same Pod	Can share storage volumes with any other container
In-built tools for logging & monitoring	3rd party tools like ELK should be used for logging & monitoring

Kubernetes Architecture

A cluster is a group of nodes. They can be physical servers or virtual machines that have the Kubernetes platform installed.

Components in Kubernetes are divided into

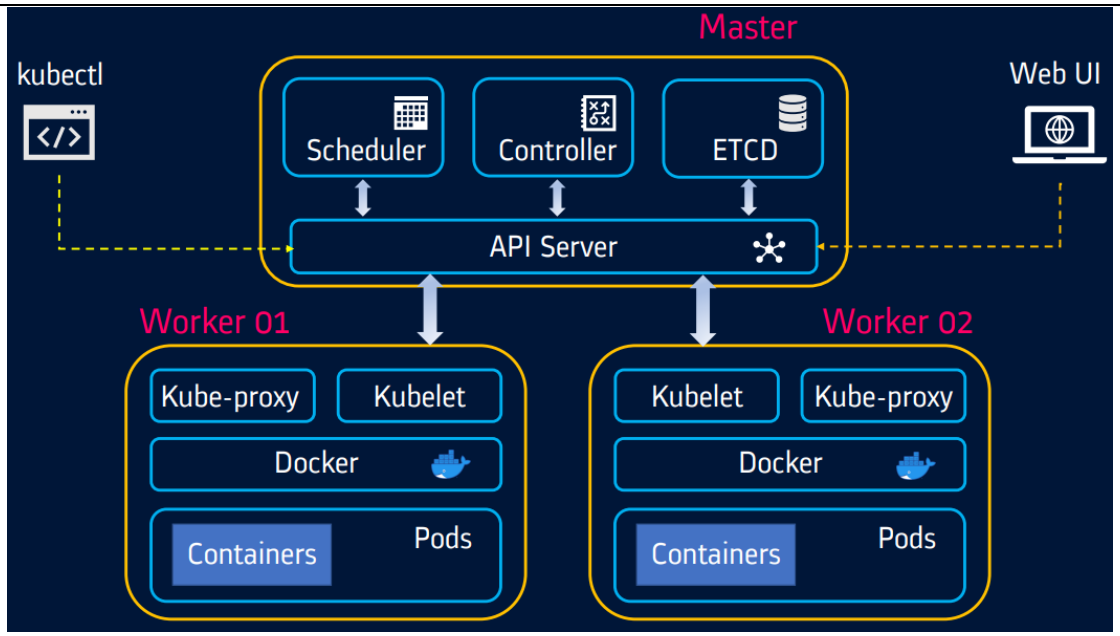
1. **Master Node(s):** hosts the Kubernetes control plane components and manages the cluster.
2. **Worker Nodes:** runs your containerized applications.

Kubernetes Master:

A single master host will manage the cluster and run several **core** Kubernetes services.

They are responsible for providing **global decisions** about the cluster like **scheduling** and **detecting** and **responding** to cluster events eg: starting up a new pod when a replication controllers replica field is unsatisfied.

Note that the loss of the master node does not mean that your cluster is down! No management will happen until the master is back up, but it is not a gateway for traffic to your services.



Following are the components of Master Node:

1. **API Server:** It's a **frontend** used by users, clients, tools and libraries with Kubernetes. It exposes REST API for health checking of other servers, allocating pods to available nodes, and orchestrating communication between other components.
2. **Etcd:** It is majorly employed for shared configuration, as well as for service discovery. It is a high available **key-value store** that can be distributed among multiple nodes. It stores the configuration of the Kubernetes cluster and information about pods, services etc. It also stores the actual state of the system and the desired state of the system.
3. **Scheduler**: It finds a suitable worker node for a newly created pod and service. The scheduler has the information regarding resources available on the members of the cluster, as well as the ones required for the configured service to run and hence is able to decide where to deploy a specific service. Impacts the availability, performance and capacity of the system.
4. **Controller Manager:** It is a control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the **current state towards the desired state**.

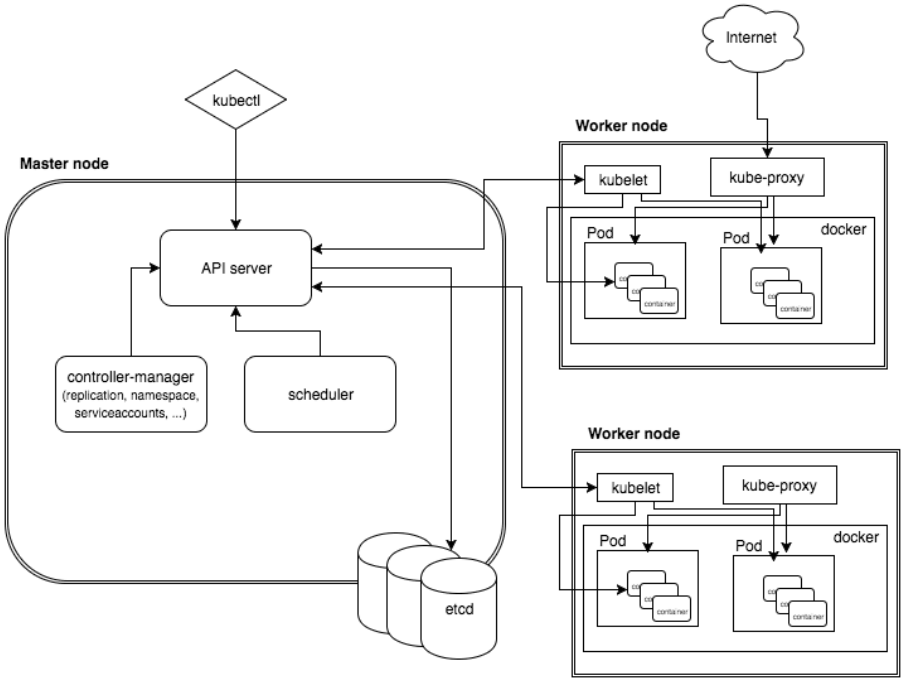
   This component runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

   These controllers include:

   - **Node Controller**: Responsible for noticing and responding when nodes go down.
   - **Replication Controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.
   - **Endpoints Controller**: Populates the Endpoints object (that is, joins Services & Pods).

Kubernetes (Worker) Nodes Components

Worker node are the servers that perform work by running containers. Following are the key components of the Node server which are necessary to communicate with master components, configuring the container networking, and running the actual workloads assigned to them.



1. **Kubelet:** This is the main contact point for each node and is responsible for **communication with the master** component to receive commands and work. The Kubelet takes a set of PodSpecs (YAML or JSON object that describes the Pod) that are provided through ApiServer and ensures that the containers described in those PodSpecs are running and healthy.
2. **Container Runtime:** The container runtime on each node is the component that runs the containers defined in the workloads submitted to the cluster. This can be docker or anything equivalent to it (containerd or CRI-O). They need to comply with a standard called **Kubernetes CRI**.
3. **Kubernetes Proxy Service:** This is a proxy service which runs on each node and helps in making services available to the end users/clients. It is responsible for ensuring network traffic is routed properly to internal and external services as required and is based on the rules defined by **network policies** in kube-controller-manager and other custom controllers.

   Client -> Kube Proxy Service -> Service -> Pod -> Container -> app
4. **cAdvisor**: Container advisor is a **resource monitoring agent** and provides information on resource usage and performance characteristics of the running containers. It collects, aggregates, processes, and exports information about running containers to scheduler.

Kubernetes Objects

- Kubernetes Pods
- Replication Controllers and ReplicaSets
- Deployments
- Services
- StatefulSets
- DaemonSets
- Jobs and Cron Jobs
- Volumes and Persistent Volumes
- ConfigMaps and Secrets

Kubectl (Utility):

Is the CLI: command line interface for running commands against the Kubernetes cluster

kubectl <operation> <object type> <resource name> <optional flags>

Kubernetes Pods

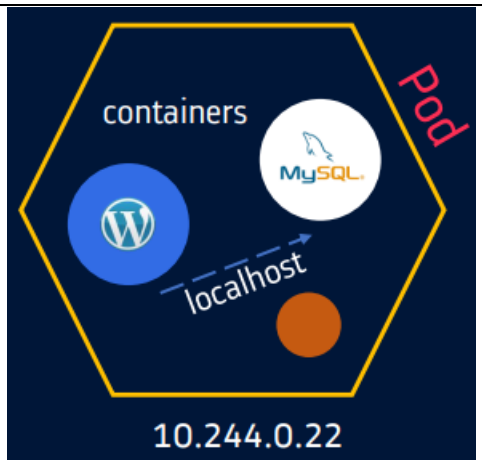
- A *Pod* is the **basic building block** of Kubernetes–the **smallest and simplest unit** in the Kubernetes object model that you **create or deploy**. A Pod represents a running process on your cluster.
- Each Pod has **one or more** application **containers** (such as Docker) and includes **shared storage** (volumes), **IP address** and information about how to run them.
- Querying a pod returns a data structure that contains information about containers and its metadata.
- Pods aren't intended to be treated as durable entities. They won't survive scheduling failures, node failures, or other evictions, such as due to lack of resources, or in the case of node maintenance.

Multi Container Pods:

- **Sidecar pattern**
  - With a sidecar, you run a second container in a pod whose job is to take action and support the primary container.
  - Logging is a good example, where a sidecar container sends logs from the primary container to a centralized logging system.

Pod Networking

- Each Pod is assigned a unique IP address.
- Any container running within a Pod will share the Node's network with any other container in the same Pod.
- Containers *inside a Pod* can communicate with one another using **localhost**.



Characteristics of a Pod

- All the containers for a Pod will be run on a same Node. Pod cannot be split across nodes.
- When a Pod is scale out, all the containers within it are scaled as a group.
- All containers in the Pod can access the shared volumes, allowing those containers to share data.
- A Pod has an explicit lifecycle and will always remain in a node in which it was started.
- Pods don't have a managed lifecycle, if they die, they will not be recreated.

Replication Controller

- Kubernetes encourages **Desired State** deployment.
- Controllers are responsible for managing the pod lifecycle. It is responsible for making sure that the **specified number of pod replicas** are running at any point of time. They take care of Node failure.
- It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.
- It's **obsolete** and replaced with ReplicaSet.

ReplicaSet

- It is associated with a Pod and indicates **how many instances of that Pod** should be running within the cluster.
- The key difference between the replica set and the replication controller is, the **replication controller** only supports **name equality-based selector** whereas the replicaset also supports **set-based selector**, a type of selection that enables a set of pods to be grouped so that the entire set is available for an operation defined in the ReplicaSet controller.
- The recommendation is to use replicasets along with higher-level controllers such as deployment.

Limitation of ReplicaSet is you can't change the Pod template - we cannot change the container image.

Deployments

- The deployment controller wraps around and extends the ReplicaSet controller.

- The Deployment instructs Kubernetes on how to **create and update instances** of your application. Once you've created a Deployment, the Kubernetes master **schedules** mentioned application instances onto individual Nodes in the cluster.
- Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces it. **This provides a self-healing mechanism to address machine failure or maintenance.**
- Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller. They have the capability to update the **replicaset** and are also capable of rolling back to the previous version.

StatefulSets

- StatefulSets were designed to solve the problems associated with data loss when pods crash.
- The StatefulSets object also has a unique and stable hostname that can be queried through DNS. Pods get a consistent naming scheme that is ordered. For example, pod-0, pod-1, pod-2, etc.
- Works like a deployment, but provides guarantees about the order and uniqueness of pods.
- Stateful sets can be configured to mount a **persistent storage volume**. So even **if a pod crashes**, **data is preserved on persistent data storage**.
- Useful when you have a group of servers that work together and need to know each others' names ahead of time. For example, **ElasticSearch** cluster.

DaemonSets

- A *DaemonSet* ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them.
- As nodes are removed from the cluster, those Pods are garbage collected.
- Deleting a DaemonSet will clean up the Pods it created.
- Useful for system level resources such as monitoring, logging, etc.
- This is how the pods on the worker nodes run, such as the kube-proxy and kubelet.

Service

- Services enable communication between various components within and outside of the application.
- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic.
- It provides an abstraction through to your Pods agnostics of the specific instances that are running.

- Discovery and routing among dependent Pods (such as the frontend and backend components in an application) is handled by Services. Basically, service enables loose coupling between Microservice in our application.

Ingresses

An ingress is a layer 7 HTTP load balancer. Ingresses are Kubernetes resources that expose one or more services to the outside world. An ingress provides externally visible URLs to services and load-balance traffic with SSL termination. This resource is useful in scenarios where rolling out such services is not possible or is expensive. Ingresses can also be used to define **network routes** between namespaces and pods in conjunction with network policies. They are managed using ingress controllers, which can limit requests, redirect URLs, and control access.

Jobs

Jobs are a Kubernetes resources that create one or more pods and ensure that they run until they **succeed**. Jobs are ideal for tasks that run and achieve a goal, and then stop.

As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete.

Cron Jobs

One CronJob object is like one line of a *crontab* (cron table) file. It runs a job periodically on a given schedule, written in Cron format.