**Introduction to Kubernetes**

𝐖𝐡𝐚𝐭 𝐢𝐬 𝐊𝐮𝐛𝐞𝐫𝐧𝐞𝐭𝐞𝐬? Kubernetes is an open-source container 𝐨𝐫𝐜𝐡𝐞𝐬𝐭𝐫𝐚𝐭𝐢𝐨𝐧 platform that automates deploying, scaling, and managing applications in containers.

𝐇𝐢𝐬𝐭𝐨𝐫𝐲: Kubernetes was initially developed by Google to solve the challenges of managing containerized applications at scale. Google had an internal system called 𝐁𝐨𝐫𝐠 that handled container orchestration. Project 7 🚀 When a group of engineers started to work on open-sourcing a more streamlined, scalable orchestration platform based on their learnings from Borg, they code-named it "Project 7. Project 7 evolved into a robust and open-source-ready platform; it was officially named "Kubernetes.

𝐖𝐡𝐲 𝐢𝐬 𝐢𝐭 𝐄𝐬𝐬𝐞𝐧𝐭𝐢𝐚𝐥 𝐢𝐧 𝐌𝐨𝐝𝐞𝐫𝐧 𝐃𝐞𝐯𝐎𝐩𝐬?

Containers are a good and easy way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. In docker we used docker swarm for this. but any how docker has drawbacks

1.) 𝐄𝐟𝐟𝐢𝐜𝐢𝐞𝐧𝐭 𝐑𝐞𝐬𝐨𝐮𝐫𝐜𝐞 𝐔𝐬𝐚𝐠𝐞 ⚙️: Kubernetes optimizes resource use, helping teams save on hardware and cloud costs.

2.) 𝐄𝐧𝐯𝐢𝐫𝐨𝐧𝐦𝐞𝐧𝐭 𝐂𝐨𝐧𝐬𝐢𝐬𝐭𝐞𝐧𝐜𝐲: It provides a consistent platform for development, testing, and production, ensuring smooth deployments.

3.) 𝐒𝐜𝐚𝐥𝐚𝐛𝐢𝐥𝐢𝐭𝐲 📈: Automatically scales applications up or down to handle load changes efficiently.

4.) 𝐒𝐞𝐥𝐟-𝐇𝐞𝐚𝐥𝐢𝐧𝐠: Detects and replaces failed containers, maintaining uptime.

5.) 𝐒𝐞𝐜𝐮𝐫𝐢𝐭𝐲 & 𝐂𝐨𝐦𝐩𝐥𝐢𝐚𝐧𝐜𝐞 🔐: Kubernetes offers built-in security features, like Role-Based Access Control (RBAC) and Secrets management.

**Cluster:** A Kubernetes cluster is a collection of nodes (machines) where Kubernetes manages and orchestrates containerized applications. A cluster represents a Kubernetes deployment as a whole, combining both worker nodes (where applications run) and a control plane (which manages and monitors the cluster).

**Problems with docker**

**1. Single Host Limitation**

Docker runs containers on a single host unless orchestrated otherwise. If you run 100 containers on one host and one of them starts consuming too much memory or CPU, it can affect other containers — for example, the 50th or 99th — because all containers share the same underlying host resources. There's no isolation or prioritization unless you manually configure resource limits. If a container doesn't get the necessary resources, it may crash and restart repeatedly, impacting overall stability.

**2. Lack of Native Auto-Healing**

If someone manually stops or kills a container, Docker itself doesn’t have built-in auto-healing mechanisms. This means the application running inside that container becomes unavailable unless someone manually restarts it or a custom script is in place. Docker by itself does not ensure service continuity in the event of container failure.

**3. No Auto-Scaling**

Docker does not provide automatic scaling. Suppose your application is deployed on a host with 4 CPUs and 4 GB RAM and can handle 10,000 users. During peak times, like a festival season, if traffic surges to 100,000 users, the host will be overwhelmed. Without the ability to scale out dynamically across more resources or machines, the application will crash due to overload.

**4. Not Enterprise-Ready Out-of-the-Box**

Docker is lightweight and ideal for simple workloads. However, it lacks several features expected in enterprise environments, such as:

• Auto-scaling and self-healing

• Native support for load balancing

• Security features like firewall rules

• Service discovery and API gateways

To address these gaps, you need a more comprehensive platform — and that’s where Kubernetes comes in.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**How Kubernetes Solves These Problems**

1. Clustering by Default Kubernetes organizes infrastructure into a cluster, which is a group of nodes (physical or virtual machines). These nodes share resources and workloads can be distributed across them, solving the single-host limitation.

2.Resource Management Kubernetes allows you to define resource limits and requests per container (CPU/memory), ensuring fair allocation and preventing any single container from starving others.

3.Auto-Healing Kubernetes constantly monitors the health of pods (which run containers). If a container crashes or is killed, Kubernetes automatically restarts it. If a node fails, the workloads are rescheduled on healthy nodes — no manual intervention needed.

4.Auto-Scaling Kubernetes supports horizontal pod autoscaling, which automatically increases or decreases the number of pod replicas based on CPU usage or custom metrics. It can also support cluster autoscaling, which adds or removes nodes based on demand.

5. **Enterprise Features Kubernetes supports:**

 Built-in load balancing across pods

 Network policies (firewalls)

 Ingress controllers and API gateways

 Secrets management

 Rolling updates and rollbacks

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Conclusion**

While Docker is excellent for building and running containerized applications, it’s not enough for running them reliably at scale, especially in production environments. Kubernetes complements Docker by offering powerful features like clustering, self-healing, auto-scaling, and enterprise-grade networking and security.

In short, Docker helps you run containers; Kubernetes helps you run containers at scale.

**The Heart of Kubernetes: Understanding Its Architecture ⚙️**

1️. Master Node (Control Plane)

The Master Node is the brain of the Kubernetes cluster. It manages the entire cluster and ensures that everything runs smoothly. Here are its key components:

⏺️ Cloud Controller Manager (CCM)

Handles interactions with cloud providers (e.g., AWS, Azure, GCP). It ensures that Kubernetes can leverage cloud-specific features like load balancers or storage.

⏺️ API Server:

Acts as the gateway to the Kubernetes cluster. All communication with the cluster (e.g., deploying applications, scaling pods) goes through the API Server.

⏺️ etcd:

A highly reliable distributed key-value store that holds all the configuration data for the cluster. Think of it as the database for Kubernetes.

⏺️ Scheduler:

Decides where to run your application pods by allocating them to worker nodes based on resource availability and other constraints.

⏺️ Controller:

Ensures that the desired state of the cluster matches the actual state. For example, if a pod fails, the controller will restart it.

**2️. Worker Nodes**

Worker nodes are where your applications actually run. Each worker node contains the necessary tools to manage and execute containers. Here are its key components:

⏺️ Kubelet:

The primary agent on each worker node. It ensures that the pods defined in the cluster specification are running correctly. It communicates with the API Server to report the status of pods and nodes.

⏺️ Kube-proxy:

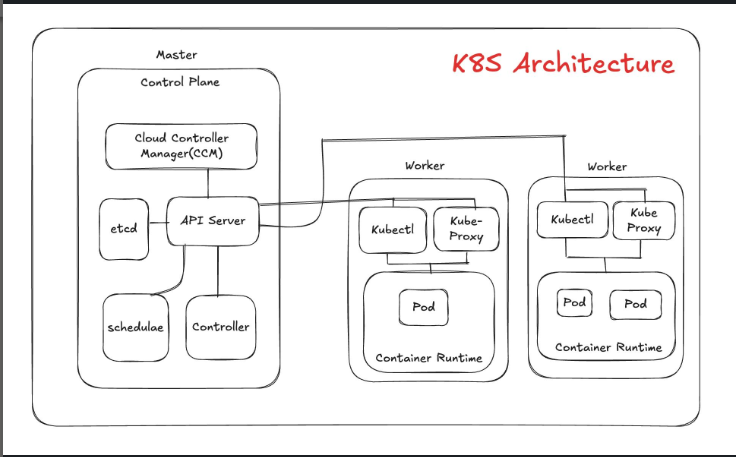
Manages network rules and enables communication between services within the cluster. It ensures that traffic reaches the correct pods.

⏺️ Container Runtime:

The software responsible for running containers. Common container runtimes include Docker, containerd, or CRI-O.

⏺️ Pods:

The smallest deployable units in Kubernetes. Pods can contain one or more tightly coupled containers. They are the building blocks of your applications.



In Kubernetes, nodes can be added or removed based on workload demands. Adding more worker nodes allows the cluster to run more applications or manage higher workloads, enhancing scalability.

**POD:**

It is a smallest unit of deployment in K8's.

It is a group of containers.

Pods are ephemeral (short living objects) Mostly we can use single container inside a pod but if we required, we can create multiple containers inside a same pod.

when we create a pod, containers inside pods can share the same network namespace, and can share the same storage volumes .

While creating pod, we must specify the image, along with any necessary configuration and resource limits.

K8's cannot communicate with containers, they can communicate with only pods. We can create this pod in two ways, 1. Imperative(command) 2. Declarative (Manifest file)

A Pod is the smallest and most basic deployable unit in Kubernetes. It represents a single instance of a running process in the Kubernetes cluster. While containers are at the heart of modern application deployment, Kubernetes introduces Pods to provide an abstraction that manages one or more containers cohesively.

**Why does Kubernetes use Pods?**

Kubernetes could manage containers directly, but Pods provide:

1.Logical Grouping Containers in a Pod form a logical unit of deployment and scaling.

2.Enhanced Collaboration Shared storage and networking simplify interactions between tightly coupled containers.

3.Unified Resource Management Kubernetes can allocate resources, handle health checks, and manage container lifecycles more effectively

by grouping them into Pods.

**Key Features of a Pod**

✒ Encapsulation: A Pod encapsulates one or more containers.

Shared resources for those containers, including:

Networking: All containers in a Pod share the same network namespace.

Storage: Volumes are shared among the containers within a Pod.

✒ Lifecycle Management: Kubernetes manages Pods rather than individual containers. This ensures containers in a Pod are always co-located, coscheduled, and run in a tightly coupled manner.

✒Atomic Deployment Unit: If a Pod fails, Kubernetes does not repair it but replaces it with a new Pod instance. Pods are designed to be ephemeral.

**Components of a Pod**

➡️ Containers: Most Pods run a single container, but you can run multiple containers in a single Pod if they are tightly coupled.

➡️Shared Network Namespace: Containers in the same Pod share:

IP Address: Assigned at the Pod level.

Ports: Containers within a Pod communicate via localhost.

➡️Volumes: Storage shared among containers in the Pod.

**Pod Lifecycle**

1.Pending: The Pod is created but not yet running. This happens while Kubernetes schedules the Pod to a node.

2.Running: The Pod is successfully scheduled and all containers are running or in the process of starting.

3.Succeeded: All containers in the Pod have terminated successfully (exit code 0).

4.Failed: At least one container in the Pod has terminated with a non-zero exit code.

5.Unknown: The state of the Pod cannot be determined.

**POD CREATION:**

IMPERATIVE:

The imperative way uses kubectl command to create pod.

This method is useful for quickly creating and modifying the pods.

SYNTAX: kubectl run pod\_name --image=image\_name

COMMAND: kubectl run pod-1 --image=nginx

kubect1. : command line tool

run : action

pod-1 : name of pod

nginx : name of image

**KUBECTL:**

**DECLARATIVE:**

The Declarative tive way we need to create a Manifest file in YAML Extension.

This file contains the desired state of a Pod.

It takes care of creating, updating, or deleting the resources.

This manifest file need to follow the yaml indentation.

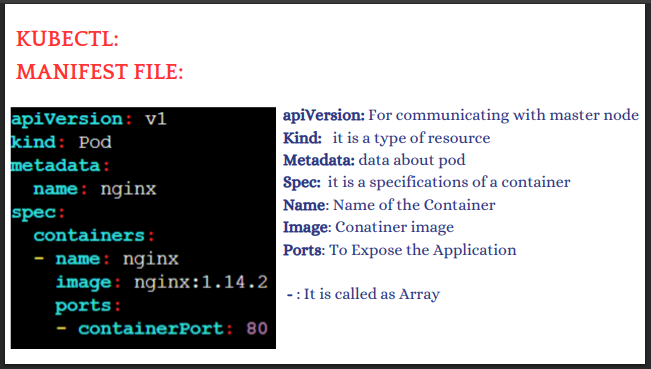
YAML file consist of KEY-VALUE Pair.

Here we use create or apply command to execute the Manifest file.

SYNTAX: kubectl create/apply -f file\_name

CREATE: if you are creating the object for first time we use create only.

APPLY: if we change any thing on files and changes need to apply the resources.



**Kubernetes Services**

Before getting started with Services in Kubernetes, the first question comes in our mind is “Why do we need Services?”. Suppose we have a website where we have 3 pod replicas of front-end and 3 pod replicas of backend.

These are the following scenarios we have to tackle:

1.How would the front-end pods be able to access the backend pods?

2.If the front-end pod wants to access the backend pod to which replica of the backend pod will the requests be redirected. Who makes this decision?

3.As the IP address of the pods can change, who keeps the track of the new IP addresses and inform this to the front-end pods?

4.As the containers inside the pods are deployed in a private internal network, which IP address will the users use to access the front-end pods?

To overcome the above mentioned cases the services object is created.

Services enables loose coupling between microservices in our application. It enables communication between various components within and outside of the application.

In Kubernetes, Services are an abstraction that define a logical set of Pods and a policy for accessing them. Since Pods in Kubernetes are ephemeral (can be created and destroyed dynamically), their IP addresses are not static. This is where Services come in—to provide stable networking and access to the Pods.

**Why Do We Need Services in Kubernetes?**

Pods Are Ephemeral: Pods are temporary and can be destroyed or recreated for reasons like scaling, updates, or failures. Each new Pod gets a different IP address, making direct communication with Pods unreliable.

Stable Communication: Services provide a consistent way to access the Pods, regardless of changes in the underlying Pods or their IPs.

Load Balancing: Services distribute network traffic among multiple Pods.

Discovery: They simplify service discovery by acting as a single access point to a group of Pods.

**How Services Work Internally?**

Label Selector: The Service identifies the set of Pods it manages using label selectors.

Endpoints Object: Kubernetes creates an Endpoints object, which keeps track of the IPs of Pods matching the Service's selector.

Service Proxying: Kubernetes uses kube-proxy to handle traffic to the Service and forward it to the appropriate Pod.kube-proxy uses methods like iptables or IPVS for load balancing.

**Key Concepts of Services**

Stable Endpoint: A Service gives a consistent IP address (called a ClusterIP) and DNS name that remains constant, even as the underlying Pods change.

Discovery and Load Balancing: Services allow clients to discover and communicate with the right Pods. They automatically distribute traffic among the Pods using label selectors and load-balancing mechanisms.

Selector and Labels: A Service uses selectors to identify which Pods it should target. Pods with labels that match the selector will automatically become part of the Service.

| **Aspect** | **Container** | **Pod** | **Deployment** |
| --- | --- | --- | --- |
| **Definition** | A lightweight, standalone executable software package. | The smallest deployable unit in Kubernetes. | A Kubernetes controller that manages the lifecycle of pods. |
| **Purpose** | Runs a single application/process. | Encapsulates one or more containers with shared storage/network. | Manages ReplicaSets to ensure desired number of pods are running. |
| **Scope** | OS-level virtualization of an app and its dependencies. | Groups containers that must work together. | Orchestrates pods across nodes with scalability and version control. |
| **Kubernetes Role** | Not directly managed by Kubernetes. | Core unit of deployment in Kubernetes. | Abstracts and automates pod management. |
| **Lifecycle** | Managed by container runtime (e.g., Docker, containerd). | Managed by Kubernetes (via controllers like ReplicaSet). | Managed by Kubernetes and defines update and rollback strategies. |
| **Isolation** | Isolated via namespaces and cgroups. | Shares resources within the pod (e.g., IP, volumes). | Manages multiple pods, possibly across multiple nodes. |
| **Networking** | Has its own IP in Docker, but not in Kubernetes. | Shares a single IP address across all containers in the pod. | No direct networking; manages pods with networking capabilities. |
| **Persistence** | Uses mounted volumes or temp storage. | Can share volumes among containers in a pod. | Can define persistent volumes as part of pod specs. |
| **Scaling** | Manual (using scripts/tools). | Not scalable by itself. | Supports horizontal scaling via replicas field. |
| **Rolling Updates** | Not applicable. | Not applicable. | Supports zero-downtime rolling updates and rollbacks. |
| **Use Case** | Running a single microservice or task. | Running tightly coupled containers (e.g., app + sidecar). | Managing and scaling stateless applications, maintaining desired state. |
| **Example** | Nginx container. | Pod with Nginx + Fluentd sidecar. | Deployment of 3 Nginx pods, auto-restarts on failure, rolling updates enabled. |

**What is a Kubernetes Namespace?**

A namespace in Kubernetes is a virtual cluster within a physical cluster. It allows you to divide cluster resources between multiple users, teams, or environments (like dev, test, and prod), helping with organization, access control, and resource management.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

🔹 W**hy Use Namespaces?**

• ✅ Environment Separation: Isolate dev, QA, and prod environments within the same cluster.

• ✅ Resource Management: Set resource limits (CPU/memory) per namespace using ResourceQuota.

• ✅ Access Control: Assign specific roles and permissions to users per namespace using Role-Based Access Control (RBAC).

• ✅ Avoid Name Conflicts: Two pods or services can have the same name if they are in different namespaces.

**Default Namespaces in Kubernetes**

1. default – Used when no namespace is specified.

2. kube-system – Used by Kubernetes system components.

3. kube-public – Readable by all users; mostly for cluster info.

4. kube-node-lease – For node heartbeats.

**When to Use Namespaces**

Use namespaces when:

• You want to separate environments (dev/test/prod).

• Multiple teams share a Kubernetes cluster.

• You need to apply different resource limits or policies per project/team.

If your cluster is small or used by a single team/project, you might not need namespaces at all beyond the default.

**Important commands:**

Create a Namespace:

kubectl create namespace my-namespace

List Namespaces:

kubectl get namespaces

Run a Pod in a Namespace

kubectl run my-pod --image=nginx --namespace=my-namespace

Set Context to a Namespace

kubectl config set-context --current --namespace=my-namespace

**Kubectl setup**

**STEP-1:** LAUNCH INSTANCE WITH T2.MICRO AND 20 GB SSD

**STEP-2: INSTALL AWS CLI**

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

unzip awscliv2.zip

sudo ./aws/install

TO CHECK VERSION: /usr/local/bin/aws --version

TO SET PATH: vim .bashrc

export PATH=$PATH:/usr/local/bin/

source .bashrc

aws –version

**STEP-3: INSTALL KOPS & KUBECTL**

Curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

wget https://github.com/kubernetes/kops/releases/download/v1.24.1/kops-linux-amd64

PERMISSIONS: chmod +x kops-linux-amd64 kubectl

MOVE FILES: mv kubectl /usr/local/bin/kubectl

MOVE FILES: mv kops-linux-amd64 /usr/local/bin/kops

TO SEE VERSION: kubectl version & kops version

**STEP**-4: CREATE IAM USER WITH ADMIN PERMISSIONS AND CONFIGURE IT IN ANY REGION WITH TABLE FORMAT

**STEP-5: CREATE INFRA SETUP**

TO CREATE BUCKET: aws s3api create-bucket --bucket musta.k8s.local --region us-east-1

TO ENABLE VERSION: aws s3api put-bucket-versioning --bucket musta.k8s.local --region us-east-1 --versioning-configuration Status=Enabled

EXPORT CLUSTER DATA INTO BUCKET: export KOPS\_STATE\_STORE=s3://musta.k8s.local

GENERATE-KEY: ssh-keygen

TO CREATE CLUSTER: kops create cluster --name musta.k8s.local --zones us-east-1a --master-size t2.medium --node-size t2.micro

TO SEE THE CLUSTER: kops get cluster

IF YOU WANT TO EDIT THE CLUSTER: kops edit cluster cluster\_name

TO RUN THE CLUSTER: kops update cluster --name musta.k8s.local --yes --admin

**STEP-6: CREATE PODS**

TO CREATE A POD: kops update cluster --name musta.k8s.local --yes --admin

TO CHECK THE PODS: kubectl get pods

TO CHECK THE POD IS RUNNING WHERE: kubectl get pods -o wide

TO CREATE A CONTIANER:

========================

vim pod-nginx.yml

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- image: nginx

name: nginx

================

TO DELETE EXISTING POD: kubectl delete pod nginx

TO CREATE A POD: kubectl create -f pod-nginx.yml

TO DEPLOY POD: we need to delete existing pod (kubectl delete pod nginx) & write the code for deployment

**Minikube set up**

UPDATE SERVER:

1 apt update -y

2 apt upgrade -y

INSTALL DOCKER:

3 sudo apt install curl wget apt-transport-https -y

4 sudo curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

INSTALL MINIKUBE:

5 sudo curl -LO

https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

6 sudo mv minikube-linux-amd64 /usr/local/bin/minikube

7 sudo chmod +x /usr/local/bin/minikube

8 sudo minikube version

INSTALL KUBECTL:

"9 sudo curl -LO ""https://dl.k8s.io/release/$(curl -L -s"

"https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"""

"10 sudo curl -LO ""https://dl.k8s.io/$(curl -L -s"

"https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"""

"11 sudo echo ""$(cat kubectl.sha256) kubectl"" | sha256sum --check"

12 sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

13 sudo kubectl version --client

14 sudo kubectl version --client --output=yaml

15 sudo minikube start --driver=docker --force