KUBERNETES

Basics

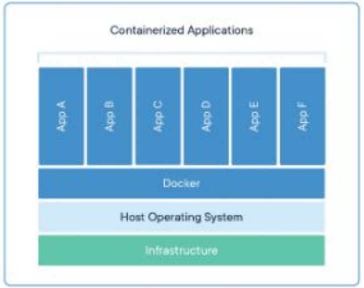
Gaurav Sharma

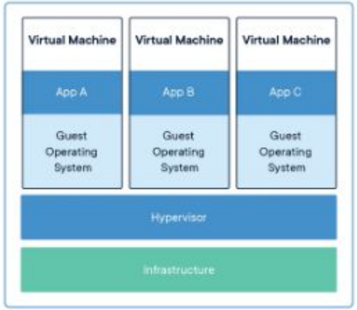
AGENDA

- What is a Container ?
- What is Docker?
- Traditional Vs Containerized Deployment
- What is Kubernetes and how it works ?
- Why you need Kubernetes ?
- Kubernetes Cluster Architecture
- Control Plane Vs Data Plane
- Master Node : Scheduler & Etcd
- Worker Node : Container Runtime Engine
- What is Pod ?
- How multiple containers are managed in a Pod?
- What is a Namespace ?
- What is a Replication Set ?
- What is a Deployment ?
- What is a Service ?
- Storage and Networking in Kubernetes
- Security in Kubernetes

What is a container ?

- Encapsulate applications and their dependencies, including runtime, libraries and settings, ensuring consistency across different environments
- Containers share the host OS kernel making them lightweight and resource-efficient compared to traditional virtual machines
- Provide process isolation, securing applications from interference and enhancing security

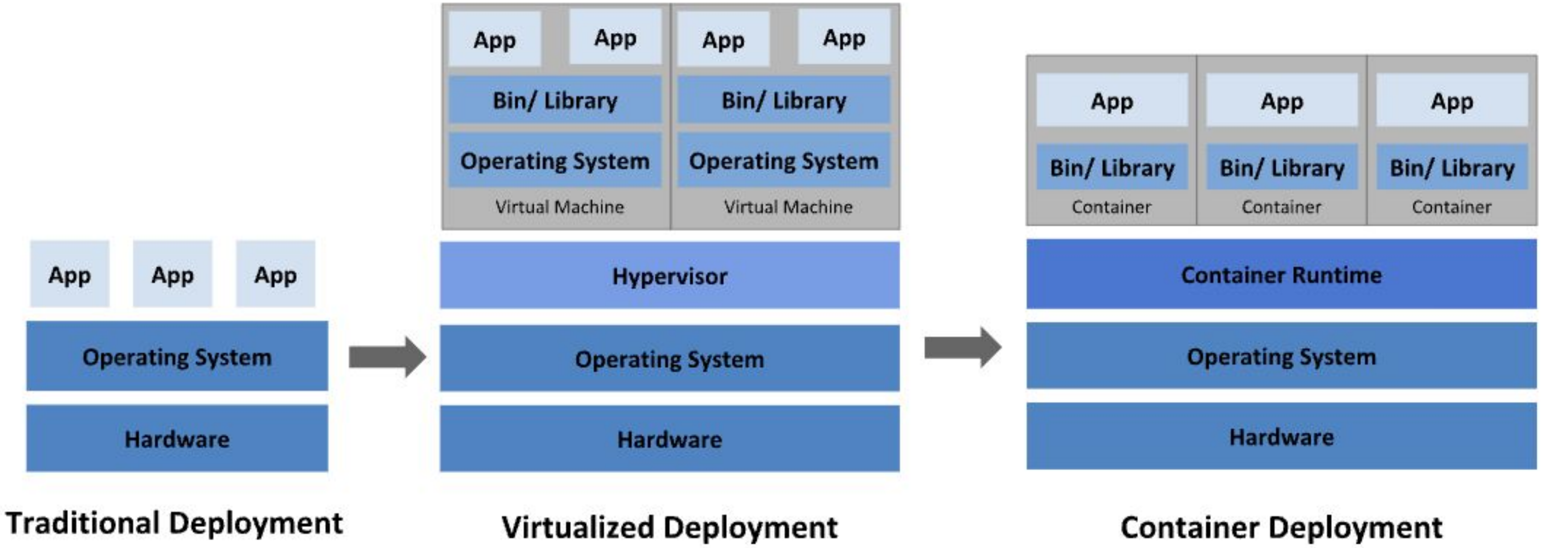




What is Docker ?

- Docker is an open platform for developing, shipping, and running applications.
- It enables you to separate your applications from your infrastructure so you can deliver software quickly
- It provides the ability to package and run an application in a loosely isolated environment called container
- With Docker, you can manage your infrastructure in the same way as you manage your applications
- Docker official documentation Link

Traditional vs Containerized Deployment

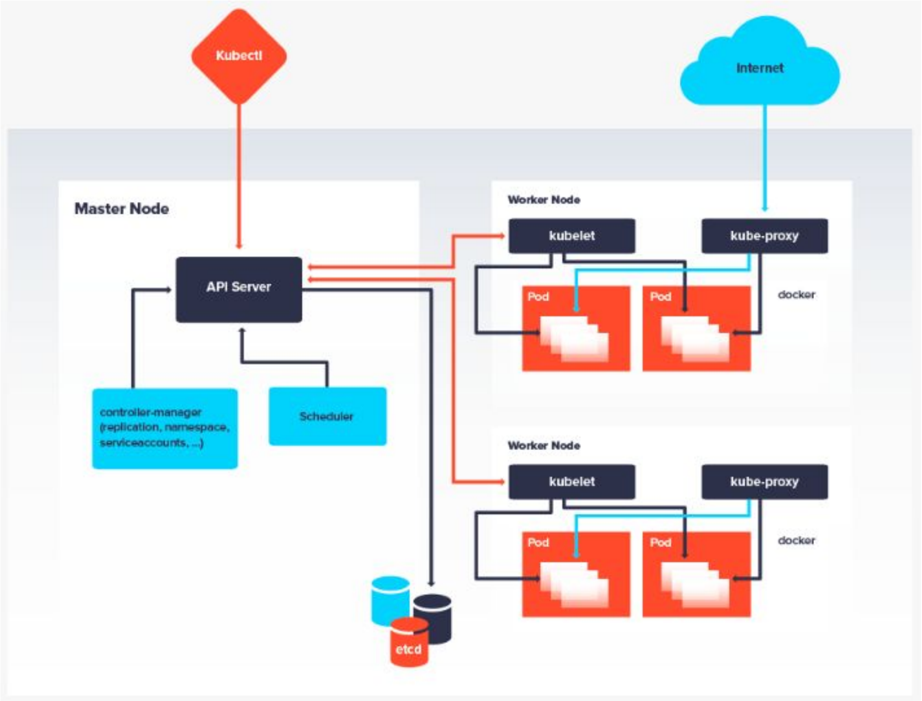


What is Kubernetes ?

- Word “Kubernetes” is originated from greek word which means helmsman/pilot
- Also called as K8s,K8s as an abbreviation results from counting the eight letters between the "k" and "s"
- Portable, extensible, open source platform for managing containerized workloads\services
- Orchestrator that schedules containers on a cluster and manages workloads to ensure they run as intended
- Facilitates both declarative configuration and automation
- K8s works by managing and coordinating containers across a cluster of machines

How Kubernetes works ?

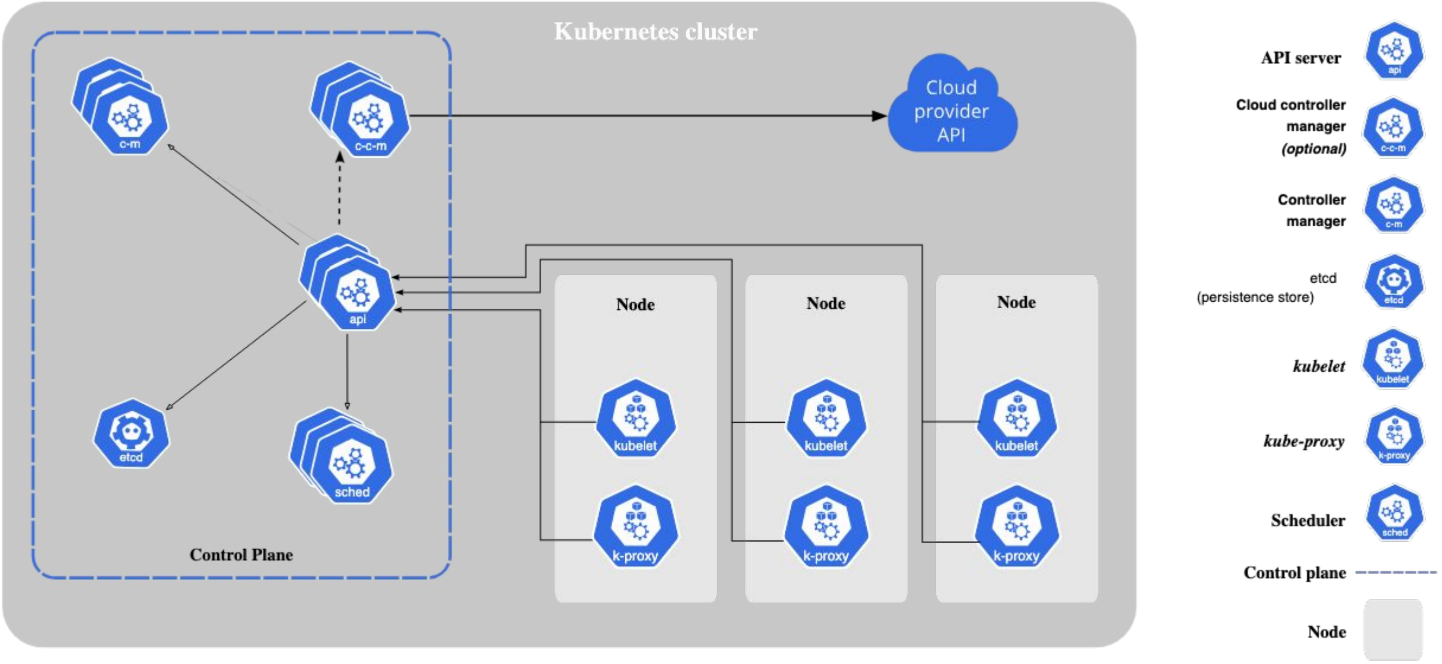
- Master Node functions as the central orchestrator, managing workloads and overseeing critical components such as the API Server, Controller Manager, Scheduler, and etcd
- Nodes serve as the execution environment for applications, receiving instructions and updates from the Master.
- Etcd component functions as a distributed key-value store, storing the cluster's state.
- Kubelet ensures container health and facilitates communication
- Kubernetes API Server validates requests and maintains the cluster's integrity in a cohesive and professional manner.



Why you need Kubernetes?

1. Service discovery & load balancing
2. Storage orchestration
3. Automated rollouts and rollbacks
4. Automatic bin packing
5. Self-healing
6. Secret & configuration management
7. Horizontal scaling

Kubernetes Cluster Architecture



Control Plane

kube-apiserver

- API server is the frontend for the K8s control plane that exposes the K8s API
- It is designed to scale horizontally— that is, it scales by deploying more instances.
- It validates and configures data for the api objects which include pods, services, replicationcontrollers, and others

etcd

- Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data

kube-scheduler

- Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on
- Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines

kube-controller-manager

- Control plane component that 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
- different types of controllers - Node controller, Job controller, Service account controller

cloud-controller -manager

- A Kubernetes control plane component that embeds cloud-specific control logic
- It runs controllers that are specific to your cloud provider
- It combines several logically independent control loops into a single binary that you run as a single process

Data Plane

Kubelet

- An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod
- It takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy
- Kubelet doesn't manage containers which were not created by Kubernetes

Kube-Proxy

- kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept
- It maintains network rules on nodes that allow network communication to the pods from network sessions inside or outside of the cluster
- It uses the operating system packet filtering layer if there is one and it's available

Container-Runtime

- A fundamental component that empowers Kubernetes to run containers effectively
- It is responsible for managing the execution and lifecycle of containers within the Kubernetes environment
- Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface)

Master Node

- High Availability
- Security
- Performance
- Monitoring
- Backup & Recovery
- Updates & Patching
- Scaling
- Documentation
- Networking
- Logging

Control Plane : Scheduler

- Kube-scheduler is the default scheduler for Kubernetes
- The scheduler finds feasible nodes for a pod and then runs a set of functions to score the feasible nodes
- The node with the highest score is selected among the feasible ones to run the Pod and the API server is then notified.This process is called binding
- Kube-scheduler selects a node for the pod in a 2-step operation: Filtering & Scoring
    - The filtering step finds the set of Nodes where it's feasible to schedule the Pod
    - In the scoring step, the scheduler ranks the remaining nodes to choose the most suitable Pod placement and assigns a score to each node that survived filtering
- There are two supported ways to configure the filtering and scoring behavior of the scheduler :
    - Scheduling Policies
    - Scheduling Profiles

Control Plane : Etcd

- Distributed Key-Value Store : Etcd is a distributed, consistent key-value store used for configuration management and service discovery in Kubernetes.
- Consistency and Reliability : It ensures strong consistency, allowing for reliable storage and retrieval of configuration data across a Kubernetes cluster.
- Raft Consensus Algorithm : Etcd employs the Raft consensus algorithm to maintain consistency and fault tolerance among distributed nodes.
- Critical Cluster Component : It serves as a critical component in a Kubernetes cluster, storing essential information about the cluster's state.
- API for Kubernetes : Etcd provides a reliable API that Kubernetes components use to store and retrieve critical configuration and state information.
- Secure Communication : Security is paramount, and etcd supports secure communication through encryption and access control mechanisms, ensuring the integrity of stored data.

Worker Node

- Resource Optimization
- Security Policies
- Scaling
- Networking
- Logging & Monitoring
- Backup and Recovery
- Container Lifecycle Management
- Updates & Patching
- Documentation
- Compatibility

Data Plane : Container Runtime Engine

Container Runtime Definition

A container runtime is the software responsible for running containers. Popular choices include Docker, containerd, and cri-o

Interoperability

Kubernetes supports multiple container runtimes, allowing flexibility in choosing the most suitable runtime for the environment

Container Lifecycle Management

The container runtime manages the entire lifecycle of containers, including creation, execution, and termination

Performance Optimization

Selecting an efficient container runtime contributes to optimized resource usage and improved application performance

Security Considerations

Container runtimes play a crucial role in enforcing security measures, such as isolation and access controls, to protect the host and other containers

Compatibility with Kubernetes

Ensuring compatibility between the container runtime and Kubernetes version is essential for seamless integration and operation within the cluster

Pod

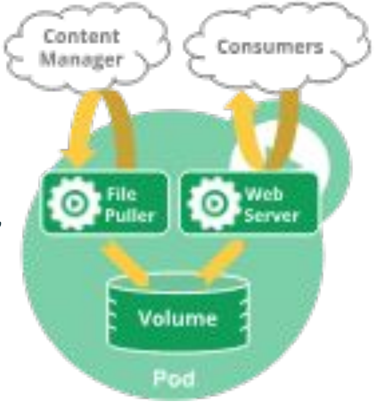
- Smallest deployable units of computing in k8s
- Group of one or more containers with shared storage and network resources
- Pods in a Kubernetes cluster are used in two main ways:
  - Pods that run a single container
  - Pods that run multiple containers that need to work together
- Pods are generally not created directly and are created using workload resources
- Containers in a Pod can share resources, dependencies, and communicate seamlessly.

Sample Pod defn yaml file

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

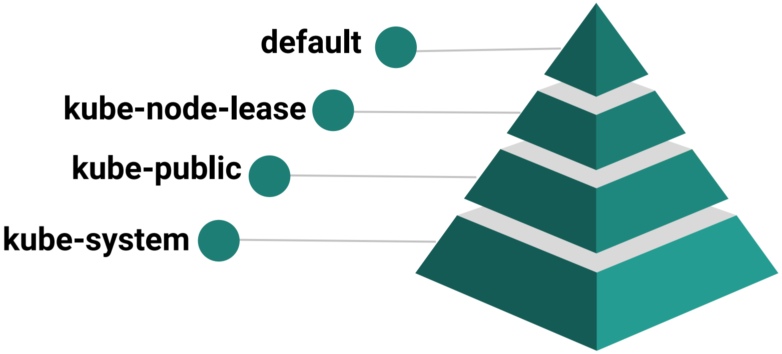
How Pods manage multiple containers ?

- Automatic Co-location and Scheduling: Pods streamline container management through automatic co-location on the same machine, promoting efficiency.
- Seamless Resource and Dependency Sharing: Containers within a Pod effortlessly share resources and dependencies, facilitating collaborative processes.
- Coordinated Communication: Containers can communicate and coordinate termination within a Pod, enhancing operational cohesion.
- Init Container Efficiency: Init containers, running before app containers by default, exemplify efficient process initiation.
- Init Container Restart Policy Control: Utilizing the Sidecar Containers feature gate enhanced control over restart policies for init containers, ensuring consistent operation.
- Networking and Storage Resources: Pods inherently offer shared networking and storage resources, further enhancing their ability to manage multiple containers effectively.



Namespace

- Namespace provides a mechanism for **isolating** groups of resources within a single cluster
- A way to divide cluster resources between multiple users (via **resource quota**)
- Intended for use in environments with many users spread across multiple teams, or projects
- **Cannot be nested** inside one another and each Kubernetes resource can only be in one namespace



Initial Namespaces

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

Sample Namespace defn yaml file

Replication Set

- A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time
- It is linked to its Pods via the Pods “metadata.ownerReferences” field
- It identifies new pods to acquire by using its selector
- It uses pod template to create new pods
- It is recommend to use Deployments instead of directly using ReplicaSets

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Sample Replicaset defn yaml file

Deployment

- Application Scaling : Deployments enable effortless scaling of applications by managing the deployment and scaling of Pods.
- Rolling Updates : Facilitates seamless rolling updates, ensuring continuous application availability during the update process.
- Rollback Capability : Offers easy rollback to previous versions in case of issues with the latest deployment.
- Declarative Configuration : Defined using declarative YAML, allowing easy configuration and version control.
- Load Balancing : Automatically provides load balancing across Pods to distribute traffic evenly.
- Self-healing : Monitors and ensures the desired state, automatically replacing failed Pods and maintaining application availability.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Sample Deployment defn yaml file

Services

- Abstraction to help expose groups of Pods over a network.
- Each Service object defines a logical set of endpoints
- The set of Pods targeted by a Service is usually determined by a selector that is defined
- Different service types :
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName
- Headless service : Designed for scenarios where direct communication with individual pods is necessary

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Sample Service defn yaml file

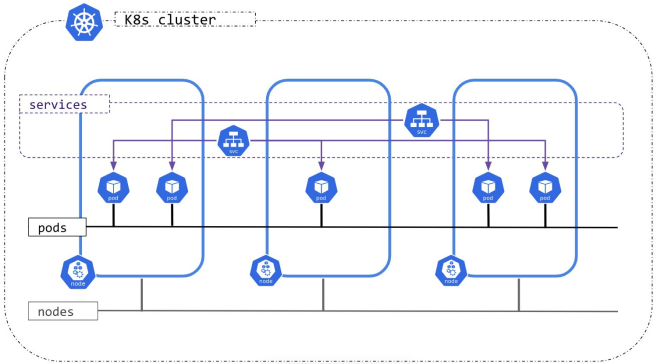
Storage

- Volumes
  - NFS: Network File System volume allows pods to share files across the network. Useful for sharing data between pods.
  - ConfigMap: Mounts configuration data as a volume, allowing pods to consume configuration files or environment variables.
  - Secret: Similar to ConfigMap but designed for storing sensitive information such as passwords or API keys.
  - PersistentVolume (PV): Represents physical storage in the cluster, decoupling it from individual pods. Can be dynamically provisioned.
  - PersistentVolumeClaim (PVC): Requests a specific amount of storage from a PersistentVolume. Binds with a matching PV.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

Sample Volume defn yaml file

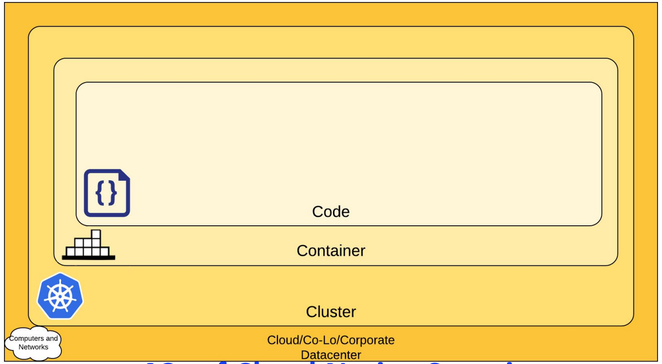
Networking



- Pod Networking:
  - Pods communicate with each other via a flat network within the cluster.
  - Each pod gets its unique IP address for inter-pod communication.
- Service Networking:
  - Services enable load balancing and provide a stable endpoint for accessing pods.
  - Cluster IP, NodePort, and LoadBalancer service types manage different networking scenarios.
- Ingress:
  - Manages external access to services, acting as an API gateway.
  - Routes external traffic to appropriate services based on rules and configurations.
- Network Policies:
  - Controls traffic between pods using defined policies.
  - Specifies which pods can communicate with each other based on labels and selectors.
- CNI Plugins:
  - Container Network Interface (CNI) plugins manage container networking.
  - Plugins handle tasks like IP address allocation, routing, and network isolation.
- DNS-Based Service Discovery:
  - Kubernetes uses DNS to enable service discovery.
  - Services and pods are assigned DNS names, allowing easy and dynamic resolution.

Security

Security in Kubernetes is a multi-faceted approach, involving access control, network segmentation, and secure handling of sensitive information. These concepts collectively contribute to a robust security posture within a Kubernetes cluster.



4Cs of Cloud Native Security

- Role-Based Access Control (RBAC) : Ensures fine-grained access control by defining roles and role bindings.It specifies what actions users, groups, or service accounts can perform within the cluster.
- Pod Security Policies : Defines security policies for pods, restricting privileges and access.
- Network Policies : Specifies how pods communicate with each other, enforcing rules on ingress and egress traffic
- Secrets and ConfigMaps : Manages sensitive information and configuration data securely. Kubernetes encrypts and manages secrets, preventing unauthorized access.

The End

Devops/SRE-DeepDive