Learn Kubernetes from Scratch

- Nasi Chaudhari (Cloud Champ)

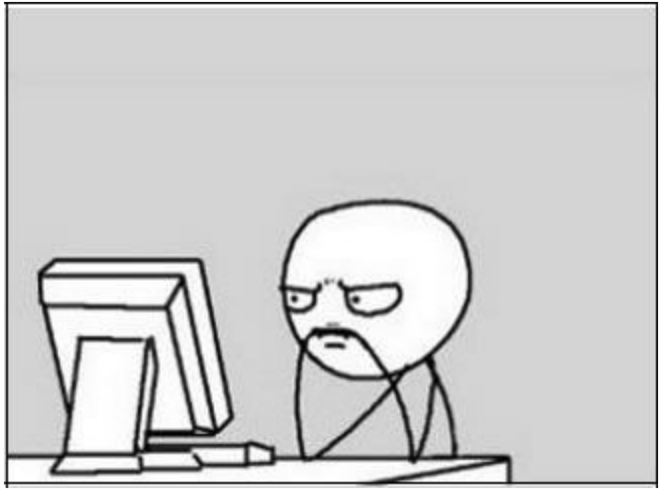
Thought of the Day

“Give a man a Container and you keep him busy for a day; Teach a man Kubernetes and you keep him busy for a lifetime.”

- Kelsey Hightower

What are we covering??

- What is Kubernetes
- How to start with Kubernetes
- Kubernetes Architecture
- Master Node components
- Worker Node Components
- Kubernetes API objects
- Scenario Based Kubernetes Questions



Follow this Repository

Github repository for the workshop:

KCD Boston Workshop: Hands-On Kubernetes Scenarios

Welcome to the **KCD Boston Workshop** repo! This workshop is built for practical Kubernetes experience. It is **not** focused on certifications but on solving real-world Kubernetes problems and learning how to find answers using the official Kubernetes documentation.

We will use KillerKoda for hands-on demos. Each folder in the `scenarios/` directory contains one practical scenario, including a description, solution, and optional interactive demo.

🔧 Workshop Objectives

- Gain practical, hands-on experience with Kubernetes
- Learn how to solve real problems using documentation
- Understand core components like etcd, networking, RBAC, and more
- Build confidence navigating the Kubernetes ecosystem

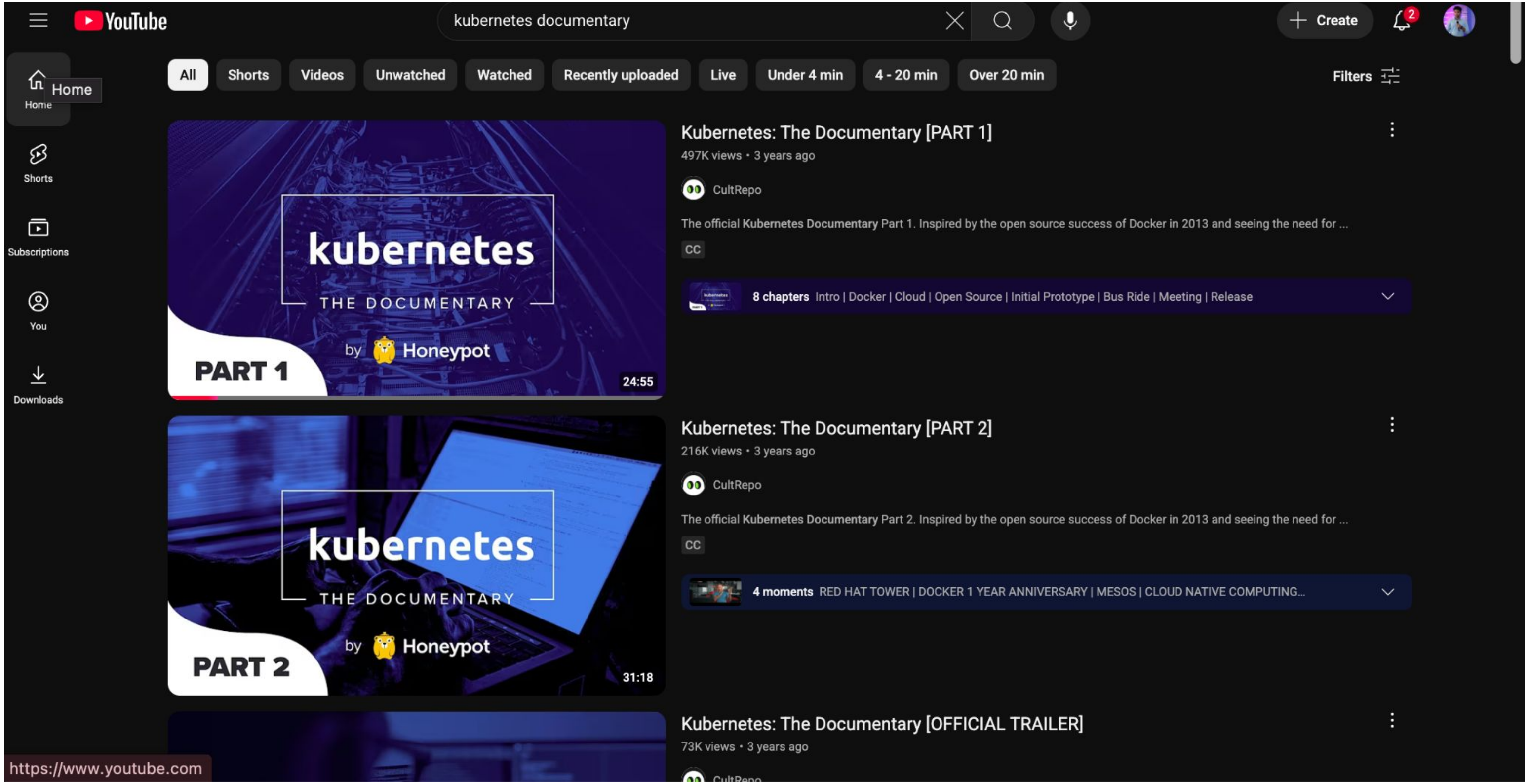
About Me:

-

Who Am I?

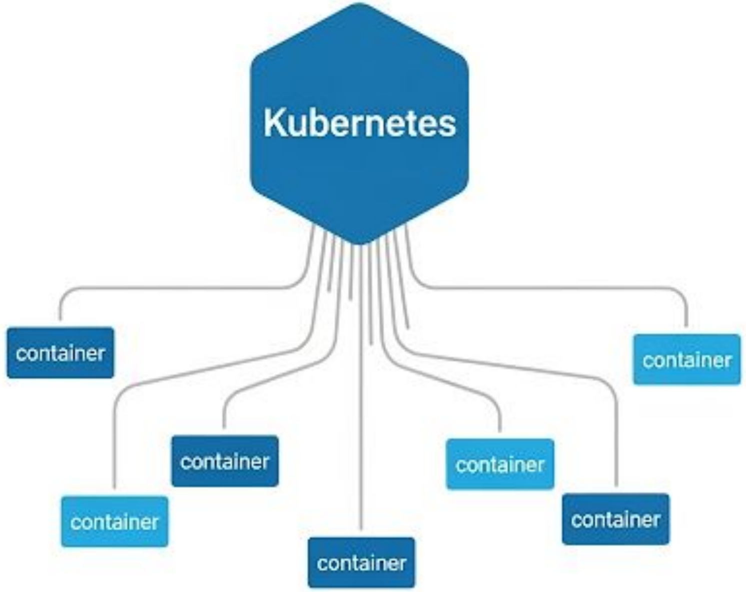
- DevOps Engineer/Consultant/Youtuber
- Kubernetes Certified (CKA/CKAD/KCNA/KCSA)
- Docker Captain
- Hashicorp Ambassador

Before I start



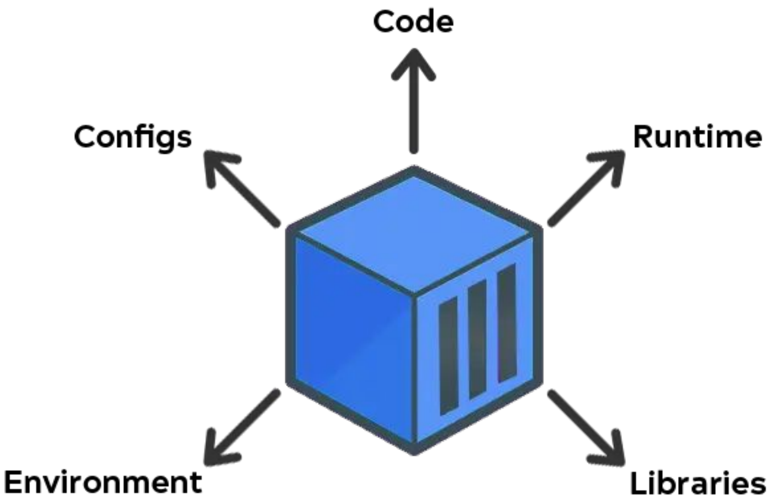
What is Kubernetes?

- Open Source Container Orchestrator.



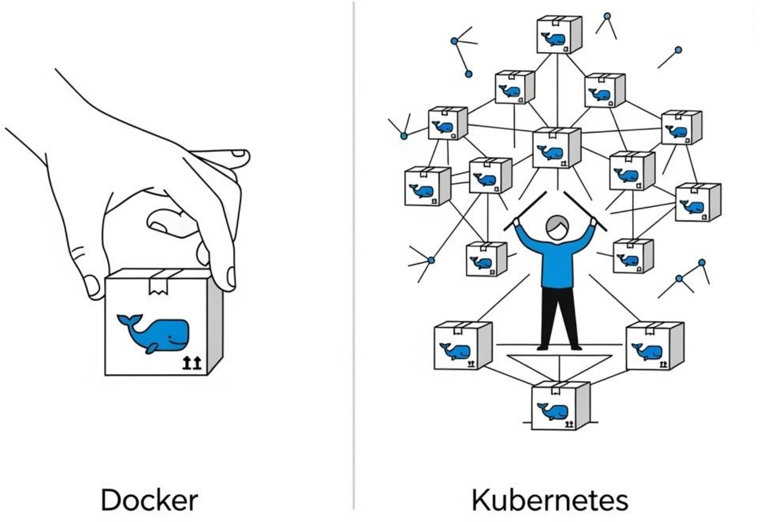
What is a Container?

- A **container** is a small package that bundles an app with everything it needs, so it runs the same anywhere.

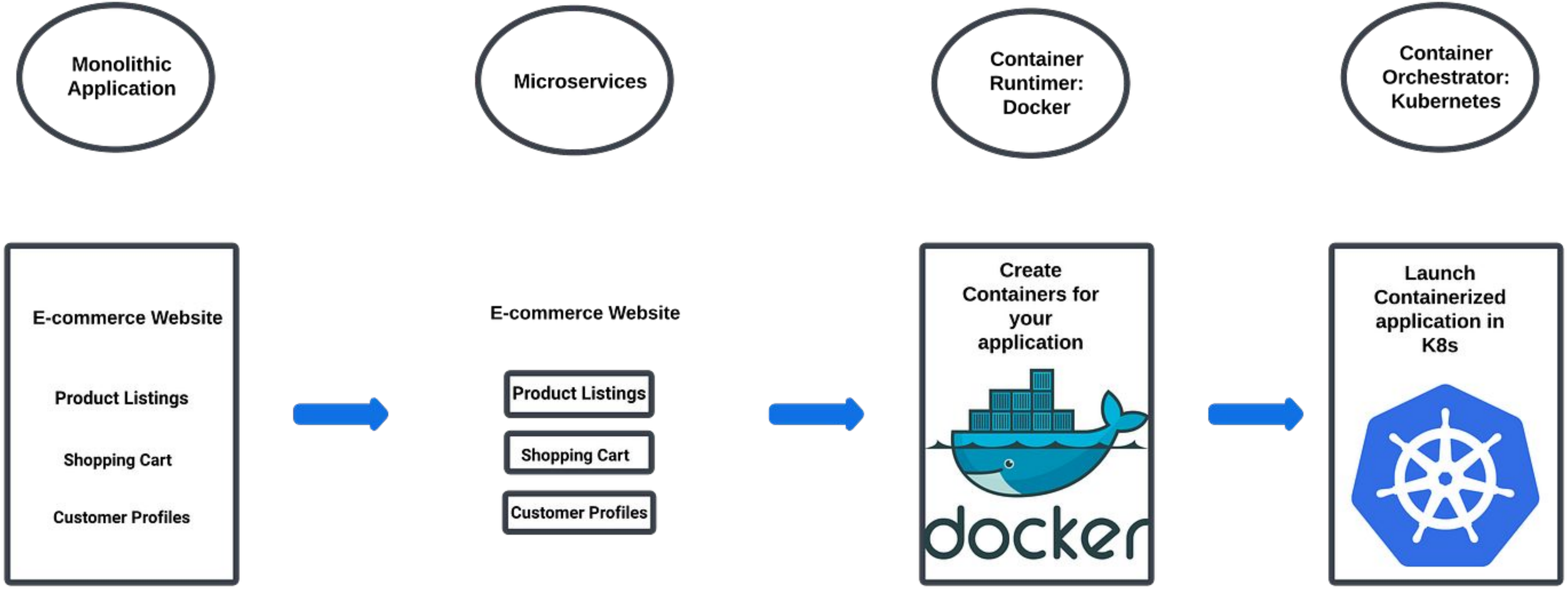


Docker vs Kubernetes

- **Docker** → creates & runs containers.
- **Kubernetes** → manages lots of containers at sc



Workflow


Monolithic Application
Microservices
Container Runtimer: Docker
Container Orchestrator: Kubernetes
E-commerce Website
Product Listings
Shopping Cart
Customer Profiles
E-commerce Website
Product Listings
Shopping Cart
Customer Profiles
Create Containers for your application
docker
Launch Containerized application in K8s

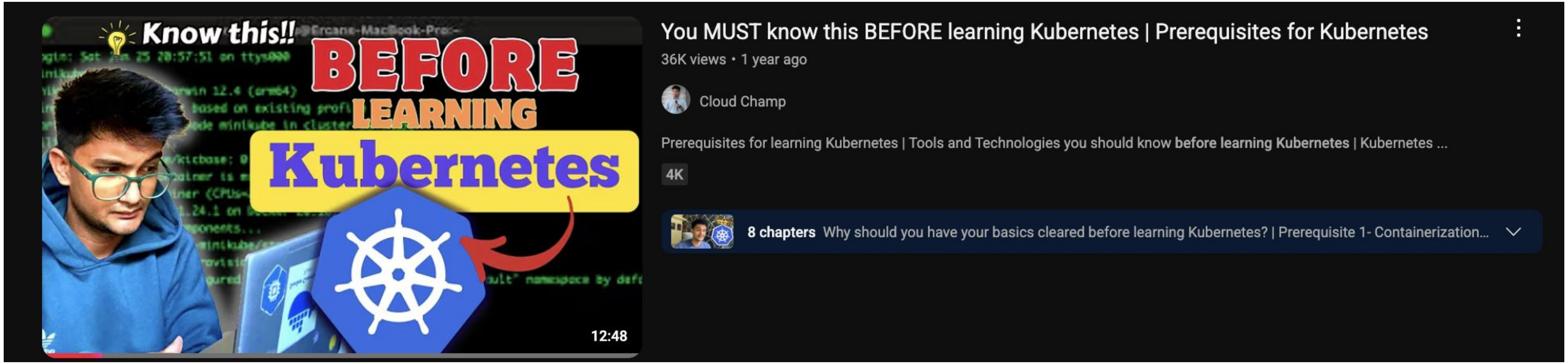
What is Kubernetes?

- Container Orchestrator
- Helps with container:
    - Deployment
    - Scaling
    - Management

Other Container orchestrators

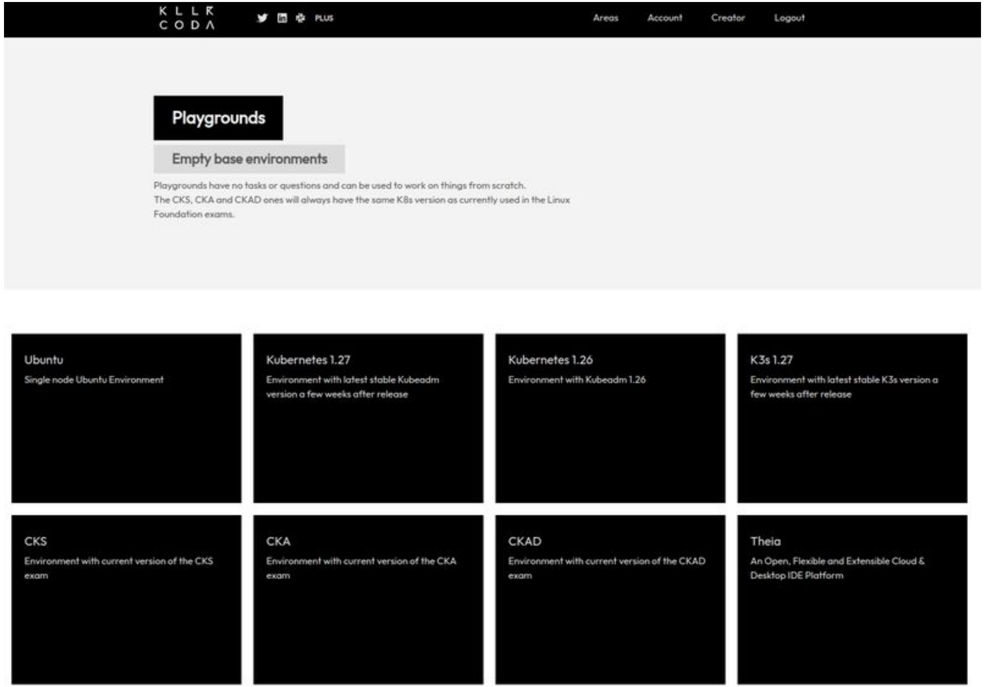
Things to know before learning Kubernetes

- Docker
- YAML
- kubectl

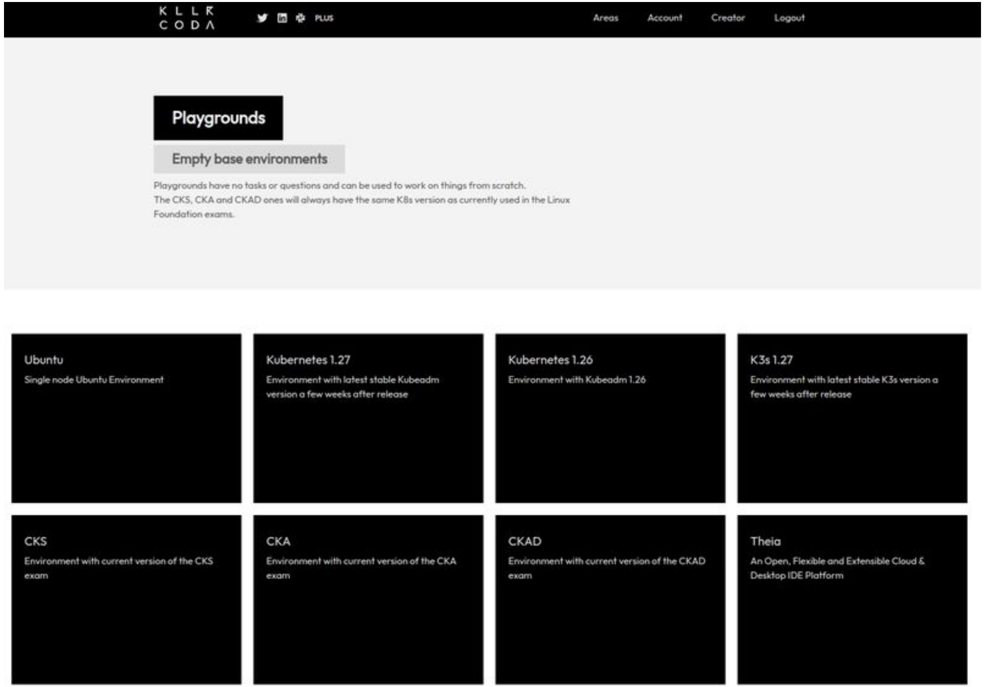


How to run a Kubernetes Cluster

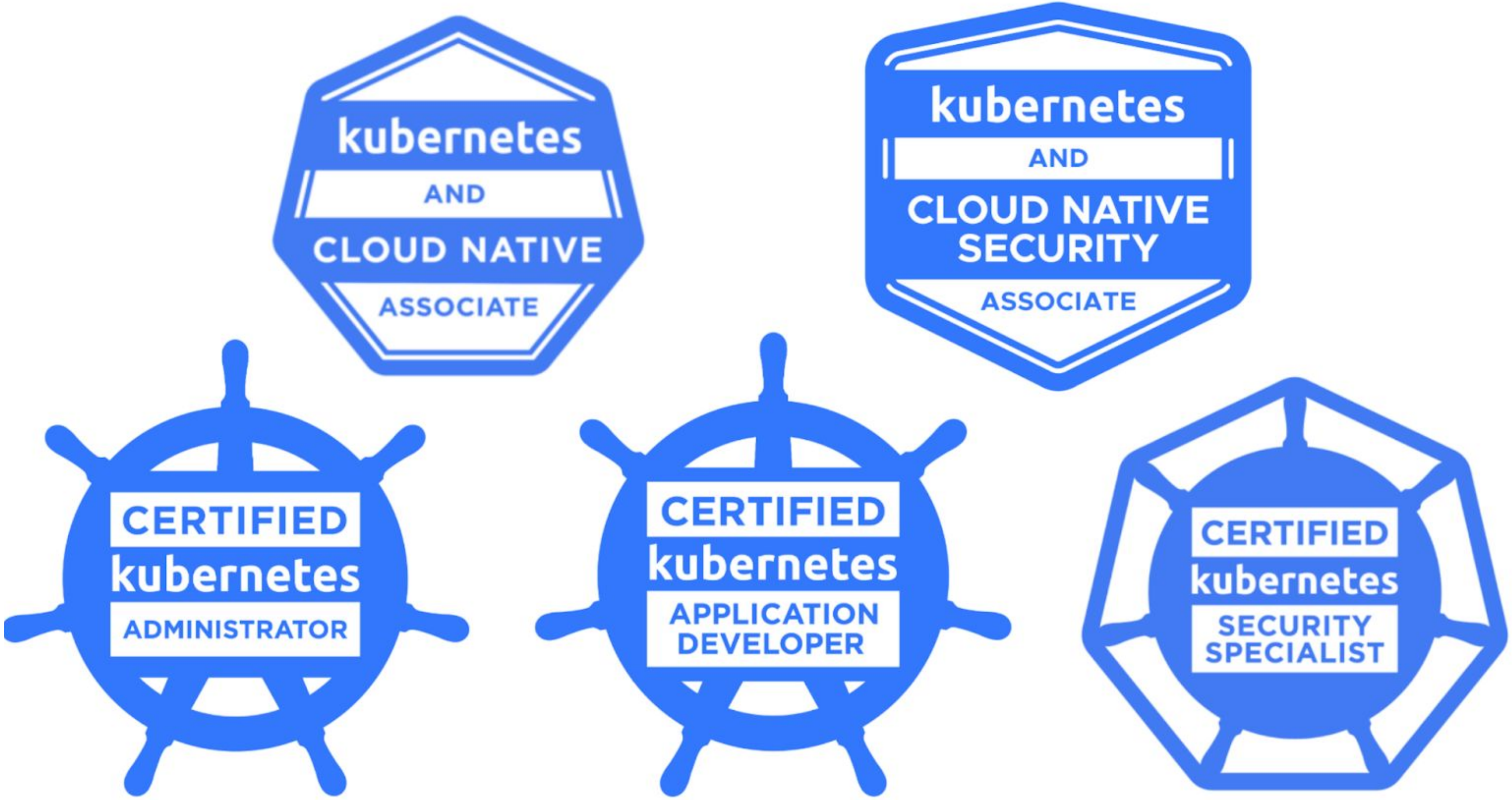
- Cloud: EKS | GKE | AKS
- Local: minikube | kind | kubeadm
- Free: Killercoda | Play-with-K8s



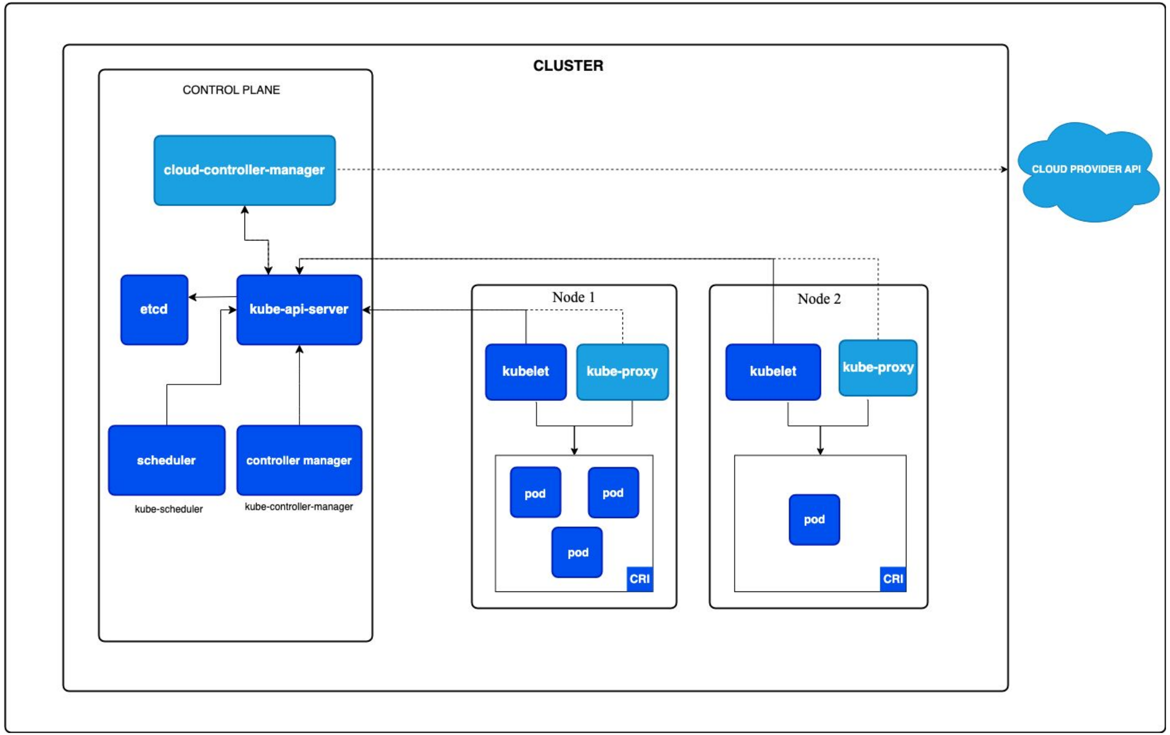
KillerCoda


KLLR CODA
PLUS
Areas
Account
Creator
Logout
Playgrounds
Empty base environments
Playgrounds have no tasks or questions and can be used to work on things from scratch.
The CKS, CKA and CKAD ones will always have the same K8s version as currently used in the Linux Foundation exams.
Ubuntu
Single node Ubuntu Environment
Kubernetes 1.27
Environment with latest stable Kubeadm version a few weeks after release
Kubernetes 1.26
Environment with Kubeadm 1.26
K3s 1.27
Environment with latest stable K3s version a few weeks after release
CKS
Environment with current version of the CKS exam
CKA
Environment with current version of the CKA exam
CKAD
Environment with current version of the CKAD exam
Theia
An Open, Flexible and Extensible Cloud & Desktop IDE Platform

Kubernetes Certs

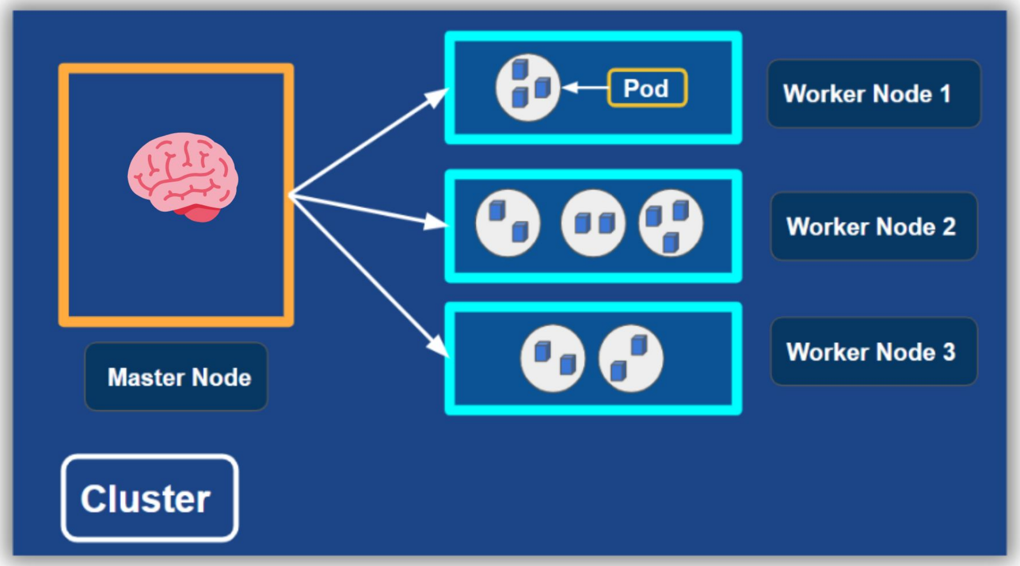


Kubernetes Architecture



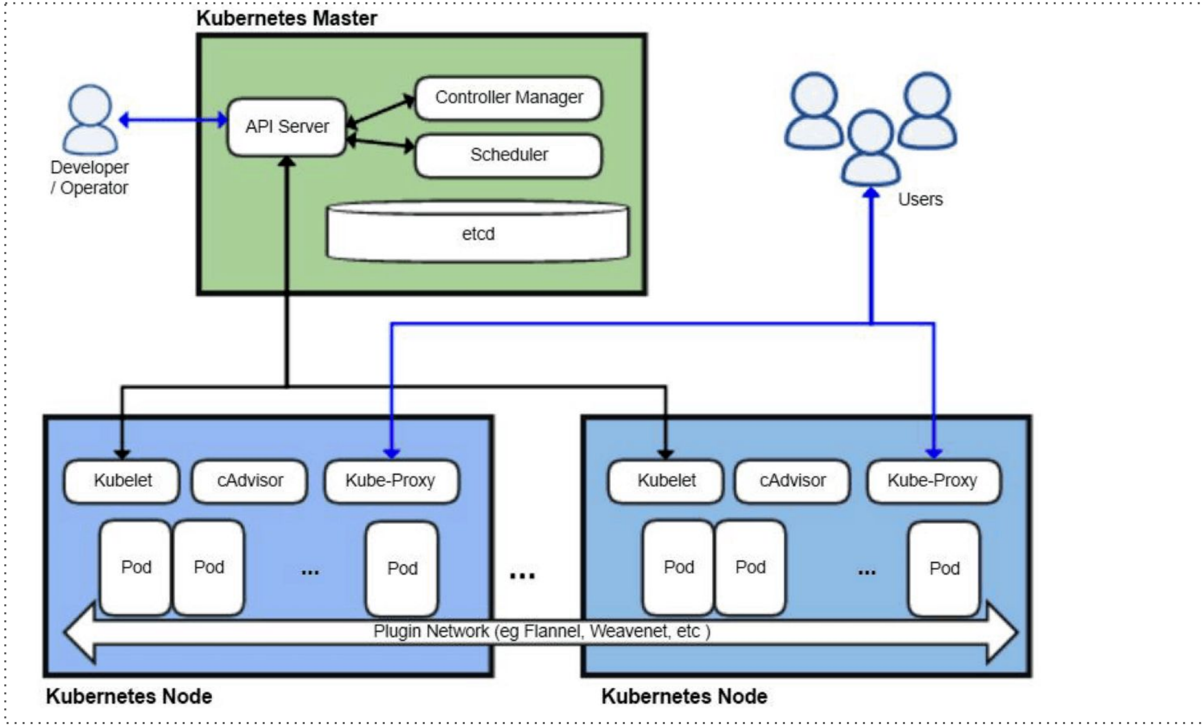
Kubernetes Architecture

- Master Node (Control Plane)
- Worker Node



Master Node

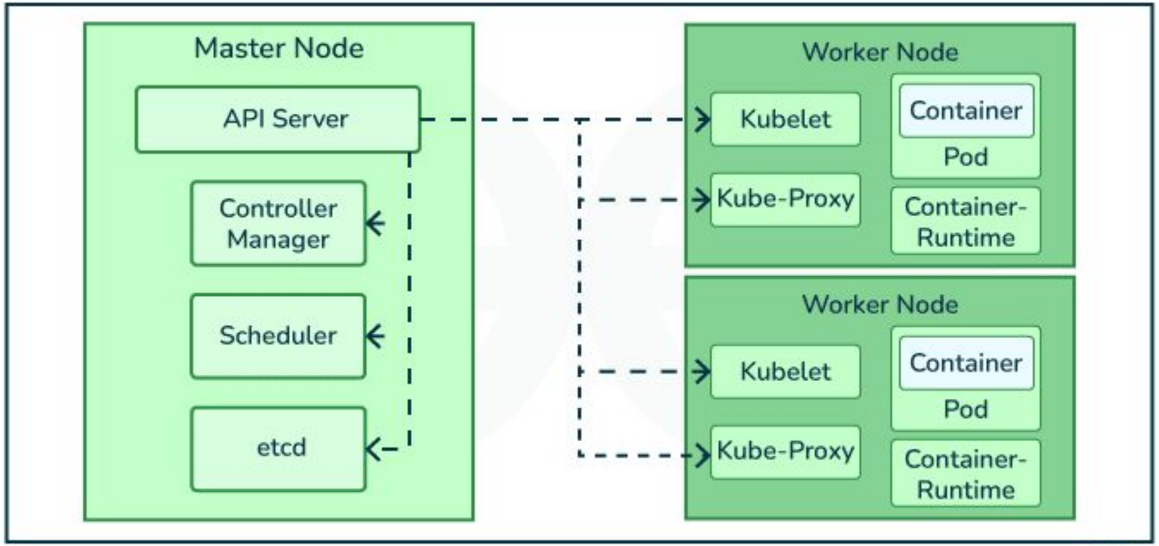
- Manages the entire cluster and is crucial for cluster-wide decisions.



Worker Node

- Receives instructions from the control plane

- Execute the actual workloads, Host containers that run applications.



Master Node Components

1. API Server
2. Controller Manager
3. Scheduler
4. etcd.

Worker Node Components

1. kubelet
2. Container runtime
3. kube-proxy

Kube API Server

- Central control point of Kubernetes
- Exposes Kubernetes API **(via kubectl, API calls)**
- Validates requests & updates cluster state in etcd

👉 Acts as the gateway to the cluster

ETCD

- Distributed, reliable key–value store
- Stores all cluster data (configuration, secrets, state, metadata)
- Acts as the single source of truth for the cluster
- Used by the API Server to persist the cluster state
- Provides high availability with leader election & data replication
- Must be backed up & secured (critical for disaster recovery)

👉 If etcd is lost, the entire cluster state is lost.

kube-scheduler

- Assigns Pods to Nodes based on resource requirements and constraints
- Watches for unscheduled Pods via the API Server
- Considers:
  - Resource availability (CPU, memory, GPU, etc.)
  - Affinity/Anti-Affinity rules
  - Taints & Tolerations
  - Node selectors / labels

👉 Scheduler decides where a Pod runs, not how it runs.

kube-controller-manager

- Runs controller processes that regulate the cluster's state
- Each controller watches the cluster state (via API Server) and takes action to match the desired state → actual state

🔑 Key Controllers:

- Node Controller → Detects node failures & updates node status
- Replication Controller → Ensures desired number of Pod replicas
- Endpoints Controller → Populates Endpoints objects (Services ↔ Pods mapping)
- Service Account & Token Controller → Manages default accounts & API tokens
- Job Controller → Manages Jobs & ensures Pods complete successfully

👉 Think of it as the automation brain of Kubernetes: "Make reality match the desired spec."

cloud-controller-manager (Optional):

- Runs cloud-specific controllers separately from core controllers
- Manages interaction with the cloud provider (AWS, GCP, Azure, etc.)

- Key responsibilities:
  - Node lifecycle management (detect node changes in cloud)
  - Route management (update network routes)
  - Load balancer management (provision cloud LBs)
  - Service account & token management for cloud resources
- Talks to API Server and cloud provider APIs, not directly to etcd

👉 Ensures Kubernetes can run across multiple cloud environments efficiently.

Worker Node Components:

1. kubelet
2. kube-proxy
3. Container runtime

kubelet

- Agent running on each node
- Watches PodSpecs from the API Server and ensures containers are running
- Reports node and Pod status back to the API Server
- Manages container lifecycle via container runtime (Docker, containerd, etc.)
- Handles health checks (liveness/readiness probes)

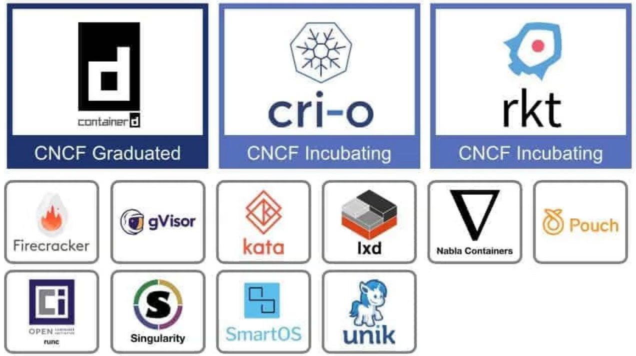
👉 Think of Kubelet as the node-level manager that keeps Pods running as intended.

k-proxy

- Network proxy running on each node
- Maintains network rules to allow Pods to communicate with Services
- Supports service discovery & load balancing across Pods
- Can operate in iptables, IPVS, or userspace mode
- Watches Service and Endpoint objects via API Server

👉 Think of Kube-Proxy as the traffic manager that routes requests to the correct Pod.

Container runtimess



- Software that runs containers on each node
- Kubelet uses the Container Runtime Interface (CRI) to interact with it
- Popular runtimes: containerd, CRI-O, Docker (deprecated in K8s)
- Responsible for:
  - Pulling container images
  - Starting/stopping containers
  - Managing container execution environment

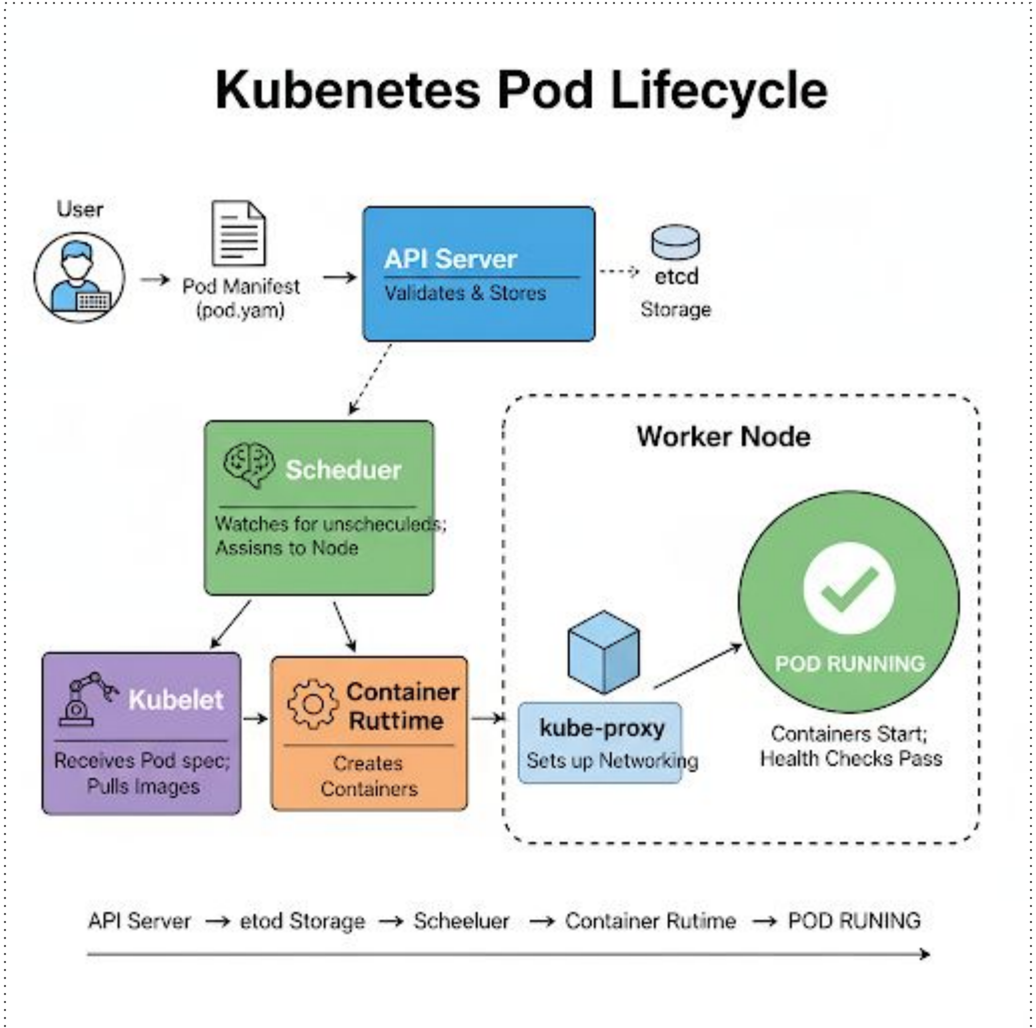
👉 It’s the engine that actually runs the containers inside Pods.s

Pod

- Smallest deployable unit in Kubernetes
- Abstraction over one or more containers
- Containers in a Pod share:
  - Network namespace (same IP, ports)
  - Storage volumes
- Used for running tightly coupled applications (e.g., sidecar pattern)
- Typically managed by higher-level controllers (Deployments, DaemonSets, Jobs)

👉 Think of a Pod as a wrapper around containers with shared resources.

What Happens when you create a POD?



What Happens when you create a POD?

1. kubectl/API request → User submits Pod spec to the API Server.
2. Validation & storage → API Server validates and stores the Pod object in etcd.
3. Scheduler → Watches for unscheduled Pods, assigns the Pod to a suitable Node.
4. kubelet on Node → Receives the Pod spec, pulls required container images.
5. Container runtime (e.g., containerd, Docker, CRI-O) → Creates containers from Pod spec.
6. kube-proxy → Sets up networking rules and services for Pod communication.
7. Pod ready → Containers start, liveness/readiness probes check health, Pod becomes Running.

👉 In short: API Server → etcd → Scheduler → Kubelet → Container Runtime → kube-proxy → Pod Running

Kubernetes API Objects/Components

Kubernetes API Objects

Kubernetes has many objects, grouped into categories.

They define desired state → what apps run, how they connect, and how resources are managed.

Kubernetes API objects are grouped into 4 main categories:

1. Workload → Pods, Deployments, StatefulSets (run your apps)
2. Service & Networking → Services, Ingress, NetworkPolicy (connect apps)
3. Config & Storage → ConfigMap, Secret, PersistentVolume (store data & configs)
4. Cluster → Node, Namespace, Role, RoleBinding (manage cluster resources & access)

1. Workload Objects (define running apps)

These define what applications run and how they run.

Examples:

- Pod → The smallest unit, runs one or more containers.
- Deployment → Manages stateless apps, handles scaling & updates.
- StatefulSet → Runs apps that need stable IDs & storage (e.g., databases).
- DaemonSet → Ensures a Pod runs on all (or some) Nodes, useful for monitoring/logging agents.

Pods

- Smallest deployable unit in Kubernetes.
- Can have one or more containers.
- Containers in a Pod share:
    - IP address
    - Volumes (storage)
    - Network namespace
- Pods are ephemeral → replaced when killed.
- Often created by higher-level controllers (Deployment, StatefulSet)

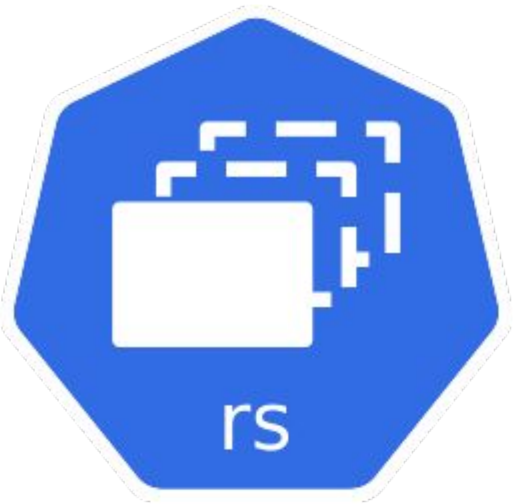
Pods

Example Manifest:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

ReplicaSet

- Ensures a set number of **identical Pods** are running.
- Recreates Pods if they fail.
- Rarely used directly → mostly managed by Deployments.



ReplicaSet

Example Manifest:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

replicas =3

Deployment

- Most common object for stateless apps.
- Manages ReplicaSets and Pods.
- Provides:
  - Rolling updates (gradual rollout).
  - Rollback (revert to previous version).
- Used for web servers, APIs, etc.

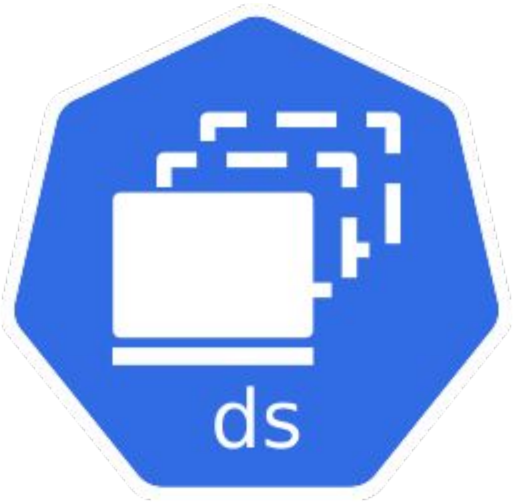
Deployment

Example Manifest:

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

DaemonSet

- Runs **one Pod per Node** automatically.
- Good for:
    - Logging agents (Fluentd).
    - Monitoring agents (Prometheus Node Exporter).
    - Networking agents (CNI plugins).
- Updates propagate across all nodes.



DaemonSet

Example Manifest:

Plain Text

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: my-daemonset
spec:
  selector:
    matchLabels:
      app: monitoring
  template:
    metadata:
      labels:
        app: monitoring
    spec:
      containers:
      - name: node-exporter
        image: prom/node-exporter
```

Service

- Exposes apps to outside world.
- Provides a stable endpoint (IP + DNS name).
- Types:
    - ClusterIP (default, inside cluster).
    - NodePort (exposed on each node’s IP).
    - LoadBalancer (uses cloud provider LB).
- Handles Pod IP changes automatically.

Service

Example Manifest:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Ingress

- Exposes HTTP/HTTPS services externally.
- Supports:
  - Host-based routing (app.example.com).
  - Path-based routing (/api → service1).
- Needs an Ingress Controller (NGINX, Traefik).

Ingress

Example Manifest:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

NetworkPolicy

- Defines rules for which Pods can communicate.
- Ingress and Egress

NetworkPolicy

Example Manifest:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
```

ConfigMap

- Stores non-sensitive config data (env vars, properties).
- Can be mounted as:
  - Environment variables.
  - Files inside Pods.
- Keeps config outside container image.

ConfigMap

Example Manifest:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: default
  labels:
    app: myapp
  annotations:
    description: "Configuration for myapp"
data:
  APP_MODE: "production"
  APP_PORT: "8080"
```

Secret

- Stores sensitive data (passwords, tokens, certs).
- Values stored in base64.
- Used like ConfigMaps but for secure data.
- Kubernetes can integrate with external secret managers

(Vault, AWS Secrets Manager).

Secret

Example Manifest:

Plain Text

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=   # base64 for "admin"
  password: cGFzc3dvcmQ= # base64 for "password"
```

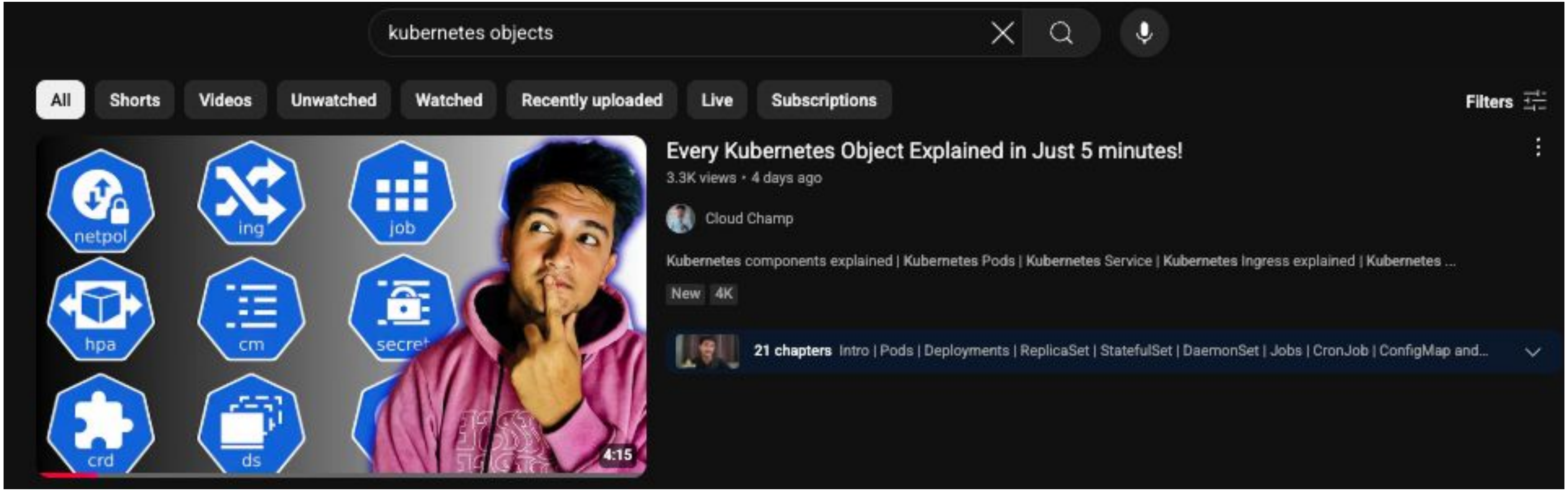
Role & RoleBinding

- Role: permissions in a namespace.
- RoleBinding: assigns Role to a user or ServiceAccount.

- **ClusterRole** → Permissions at the **cluster level** (or across all namespaces).

- **ClusterRoleBinding** → Assigns a ClusterRole to a **user, group, or ServiceAccount**.

And many more…..



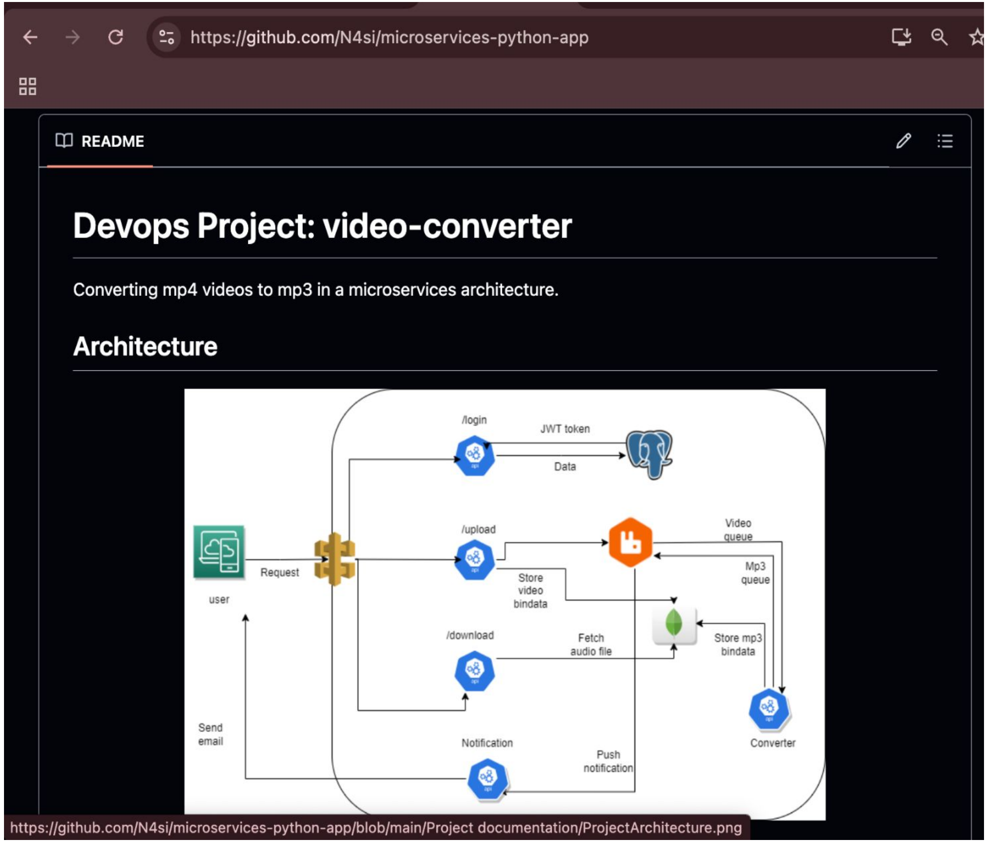
Hands on Scenarios

- Kubernetes Cluster version Upgrade
- Etcd backup and restore
- Role Based Access Control
- Taints and tolerations

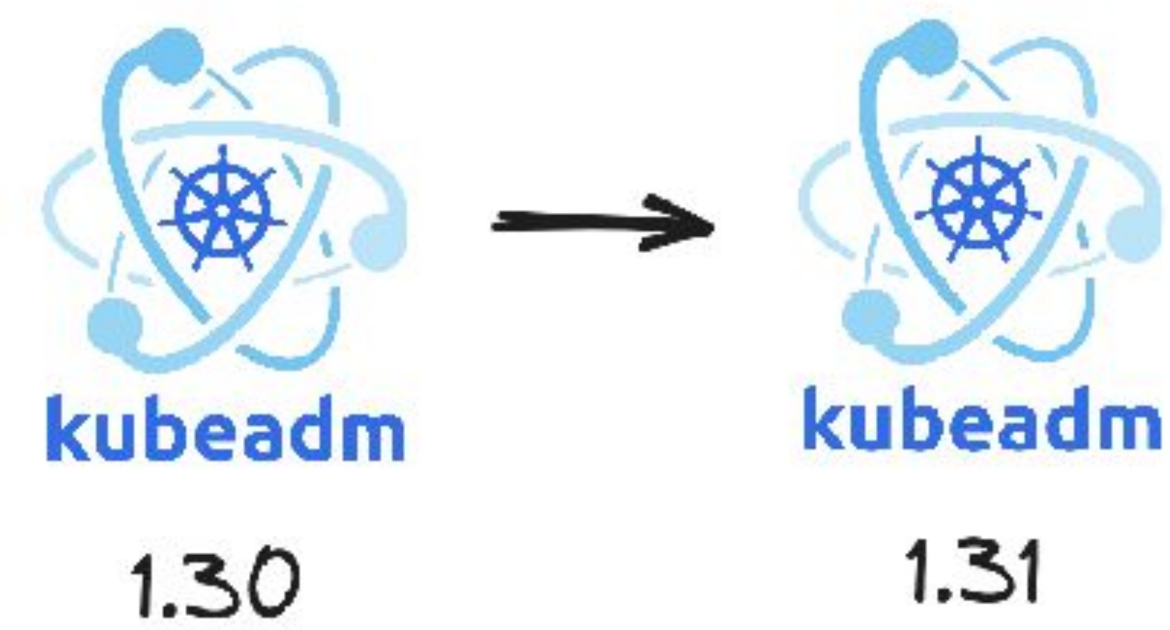
Please pose for a selfie

Valuable resource

Microservices App on K8s



Kubernetes Upgrade



Kubernetes upgrade

1. First, check the health of your cluster to ensure all nodes are ready and system pods are running correctly.

2. Next, review the available Kubernetes versions and select the target version for your upgrade.

3. Upgrade the control plane by updating kubeadm and applying the upgrade to the control plane components.

4. After that, upgrade the control plane’s kubelet and kubectl and restart the kubelet to apply changes.

5. For each worker node, drain the node, upgrade kubeadm and the node components, update kubelet and kubectl, and then uncordon the node.

6. Finally, verify that all nodes reflect the new version and that workloads are running as expected

7. As a best practice, always take a backup of etcd before upgrading production clusters, upgrade the control plane first, and validate workloads after the upgrade.

Etcd Backup

1. First, set the ETCD client to use the appropriate API version.

2. Take a snapshot of the etcd data as a backup, ensuring you use the proper certificates and keys for authentication.

3. Verify that the snapshot was created successfully and is intact.

4. To simulate a disaster, you can delete a critical component, such as the kube-proxy daemonset.

5. Confirm that the deletion was successful by checking the current state of the daemonsets.

6. Restore the snapshot to a new data directory to recover the cluster state.

7. Update the etcd manifest to point to the restored data directory, replacing the old data path with the new one.

8. Wait for etcd to restart, which usually takes a few minutes, and ensure that it comes back online successfully.

9. Finally, verify that the deleted component, like the kube-proxy daemonset, has been restored and is running correctly.

10. Your etcd backup and restore workflow is now complete.

Taints & Tolerations

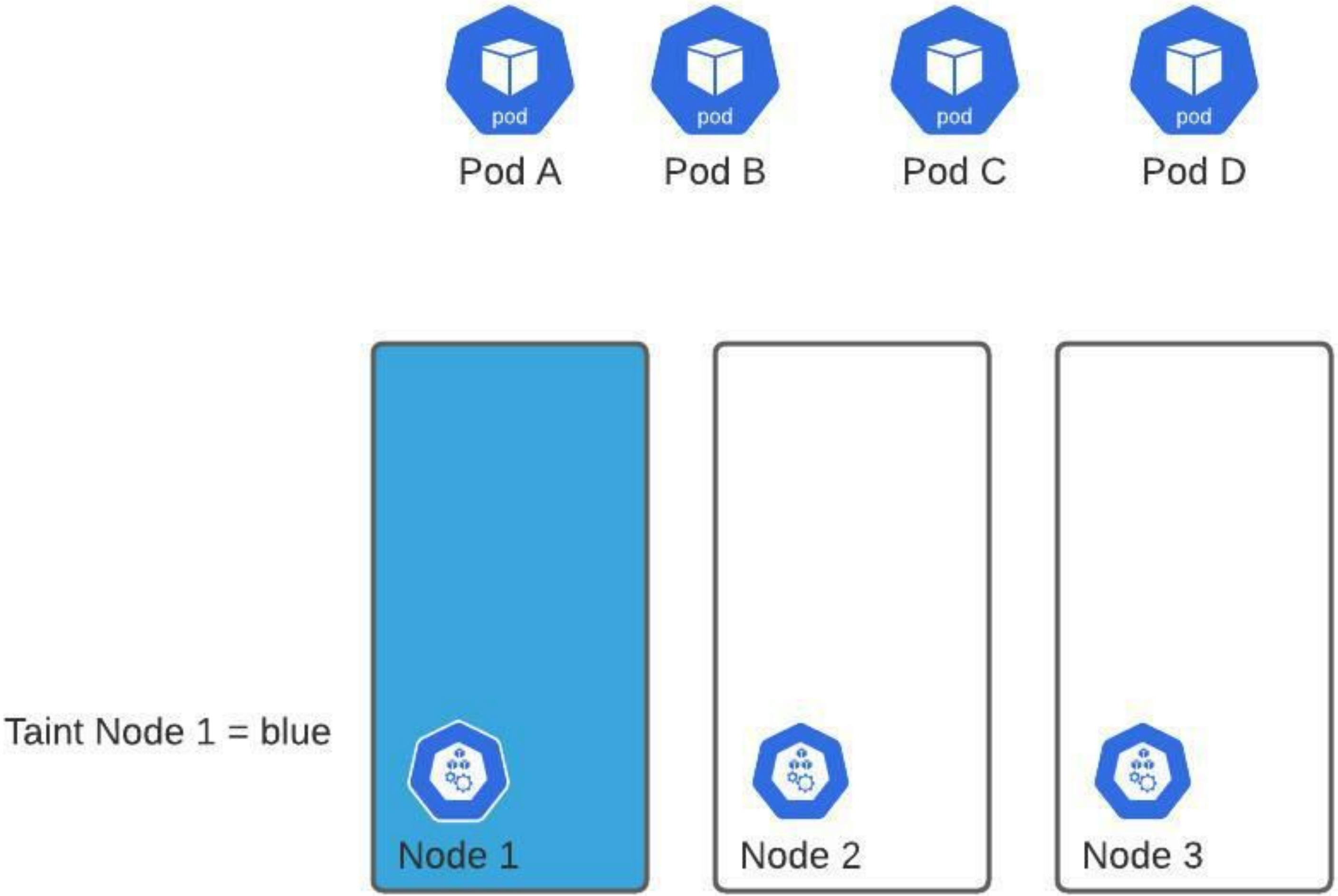
1. First, inspect the node to see if any taints are applied. A taint prevents pods from being scheduled onto the node unless the pod has a matching toleration.

2. Next, update the pod specification to include a toleration that matches the key, value, and effect of the node’s taint. This allows the pod to be considered for scheduling on that node.

3. Deploy the pod with the updated specification. Kubernetes will now allow it to run on the tainted node because the toleration satisfies the taint requirements.

4. Finally, verify that the pod is running and is scheduled to the intended node.

5. Remember that tolerations allow pods to be scheduled on tainted nodes but do not guarantee placement. To ensure a pod lands on a specific node, combine tolerations with nodeSelector or node affinity.


pod
pod
pod
pod
Pod A
Pod B
Pod C
Pod D
Taint Node 1 = blue
Node 1
Node 2
Node 3

NetworkPolicy

1. First, define a NetworkPolicy for the target application pods using a label selector. This ensures the policy applies only to the intended pods.

2. Specify the types of traffic the policy will control, typically Ingress for incoming traffic and Egress for outgoing traffic.

3. For ingress, allow traffic from all pods as well as from pods with a specific trusted label. This ensures that only desired pods can communicate with the application pods.

4. For egress, allow traffic to all pods, ensuring that the application can reach other services as needed.

5. Once the policy is applied, verify that it exists and test connectivity to confirm that ingress and egress rules are enforced according to the defined selectors.

Rbac

1. First, define a Role within a namespace that specifies which resources and actions are allowed, such as creating service accounts.

2. Next, bind this Role to a user or entity using a RoleBinding so that they can perform the actions defined in the Role.

3. Verify that the user or entity has the expected permissions by checking whether they can perform the allowed actions.

4. Create a service account within the namespace to represent an automated or system identity.

5. Assign a predefined ClusterRole to the service account using a RoleBinding to grant it read or write access to resources within the namespace.

6. Finally, confirm that the service account has the expected permissions to interact with the resources according to the assigned roles.

7. This process demonstrates how to define roles, bind them to users or service accounts, and verify access in a Kubernetes cluster using RBAC.

Thank You