lam-mithran / LWM-Kubernetes Public

1 star 14 forks Branches Tags Activity

Star Notifications

Code Issues Pull requests Actions Projects Security Insights

main 1 Branch 0 Tags Go to file Go to file Code

lam-mithran Uploaded Kubernetes part 9 ReadME 336b55c · 2 months ago

| | | |
|---|---|---|
| README.md | Uploaded Kubernetes part 9 ReadME | 2 months ago |

# 🚀 Kubernetes Part 1 – minikube & KOps Installation

This README serves as a **complete hands-on guide** for Kubernetes learning. It includes YAML manifests and command-line usage to help you practice kubernetes related topics.

## 📚 What You'll Learn

- ✅ How to create a Production Grade Cluster using **kOps**
- ✅ How to create a Local **minikube** Cluster

## 🔧 Getting Started with kOps on AWS

[Click me](#) for Official Documentation of kOps

### 1️⃣ Install Kops for Linux

```
curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/relea:
chmod +x ./kops
sudo mv ./kops /usr/local/bin/
```

### 2️⃣ Install kubectl for Linux

```
curl -Lo kubectl https://dl.k8s.io/release/$(curl -s -L https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

### 3️⃣ Setup IAM user

In order to build clusters within AWS we'll create a dedicated IAM user for kops. This user requires API credentials in order to use kops. Create the user, and credentials, using the AWS console.

The kops user will require the following IAM permissions to function properly:

```
AmazonEC2FullAccess
AmazonRoute53FullAccess
AmazonS3FullAccess
IAMFullAccess
AmazonVPCFullAccess
AmazonSQSFullAccess
AmazonEventBridgeFullAccess
```

### 4️⃣ Create S3 Bucket

In order to store the state of your cluster, and the representation of your cluster, we need to create a dedicated S3 bucket for kops to use. This bucket will become the source of truth for our cluster configuration. In this guide we'll call this bucket ==**lwm-kubernetes-bucket**==, but you should add a custom prefix as bucket names need to be unique.

### 5️⃣ Prepare local environment

You should record the SecretAccessKey and AccessKeyID in the returned JSON output, and then use them below:

```
# configure the aws client to use your new IAM user
aws configure           # Use your new access and secret key here
aws iam list-users      # you should see a list of all your IAM users here

# Create ssh-keys for kOps to use
ssh-keygen -t rsa -b 4096 -f ~/.ssh/kops-ssh-key
kops create secret --name <your-cluster-name> sshpublickey admin -i ~/.ssh/kops-ssh-key.pub

# Because "aws configure" doesn't export these vars for kops to use, we export them now
export AWS_ACCESS_KEY_ID=$(aws configure get aws_access_key_id)
export AWS_SECRET_ACCESS_KEY=$(aws configure get aws_secret_access_key)
export NAME=myfirstcluster.k8s.local
export KOPS_STATE_STORE=s3://prefix-example-com-state-store
```

For a gossip-based cluster, make sure the name ends with ==**k8s.local.**==

### 6️⃣ kOps Commands

```
kops create cluster --name=${NAME} --cloud=aws --zones=ap-southeast-1a
kops edit cluster --name ${NAME}
kops update cluster --name ${NAME} --yes --admin
kops validate cluster
kops delete cluster --name ${NAME} --yes
```

## 🔧 Getting Started with minikube on AWS

[Click me](#) for Official Documentation of minikube

### 1️⃣ Installation

Create a Linux EC2 VM with minimum

- 2 CPUs or more
- 2GB of free memory
- 20GB of free disk space

```
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

### 2️⃣ Start your cluster

```
minikube start
```

### 3️⃣ Interact with your cluster

```
minikube kubectl -- get pod -A
```

You can also make your life easier by adding the following to your shell config

```
alias kubectl="minikube kubectl --"
kubectl get pod
```

## 📞 Contact Us

**Phone:** +91 91500 87745

### 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 2 – YAML Reference Guide

In Part 2, you will learn how to create **Namespaces, Pods, ReplicaSets, DaemonSets.** Learn about kubectl cheat codes and both imperative and declarative way of deploying kubernetes Objects

## 📚 What You'll Learn

- ✅ Creating and using **Namespaces**
- ✅ Defining **Pods** with one or more containers
- ✅ Using **ReplicaSets** to maintain pod availability
- ✅ Deploying **DaemonSets** to run a pod on each node

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 1️⃣ Namespace: qa

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: qa
```

### 2️⃣ Pod: myfirstpod (Single Container)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myfirstpod
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
```

### 3️⃣ Pod: mysecondpod (Multi-Container: httpd + jenkins)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysecondpod
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
  - name: cont2
```

```yaml
    image: jenkins/jenkins
    ports:
    - containerPort: 8080
```

## 4️⃣ Pod: mythirdpod (Multi-Container: httpd + nginx)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mythirdpod
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
  - name: cont2
    image: nginx
    ports:
    - containerPort: 80
```

## 5️⃣ Pod: myfirstpod in dev Namespace

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myfirstpod
  namespace: dev
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
```

## 6️⃣ ReplicaSet: lwm-replica (5 Replicas)

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: lwm-replica
spec:
  # modify replicas according to your case
  replicas: 5
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: cont1
        image: httpd
```

## 7️⃣ DaemonSet: lwm-daemon

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: lwm-daemon
spec:
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
```

```
      - name: cont1
        image: httpd
```

### 📞 Contact Us

**Phone:** +91 91500 87745

### 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 3 – YAML Reference Guide

In **Part 3**, you will learn how to manage applications using **Deployments** and expose them to users and other services using **Kubernetes Services**.

## 📚 What You'll Learn

- ✅ Understand what a Deployment is and how it manages Pods
- ✅ Create Deployments with different update strategies (RollingUpdate vs Recreate)
- ✅ Expose Pods and Deployments using various Service types:
  - 🔹 **NodePort** – for accessing Pods externally via a fixed port on the node
  - 🔹 **ClusterIP** – for internal Pod-to-Pod communication
  - 🔹 **LoadBalancer** – for external access using cloud provider's load balancer
- ✅ Demonstrate how **selectors and labels** work in real-world scenarios
- ✅ Connect Pods internally and externally using appropriate Service types
- ✅ Expose applications like **Jenkins** via NodePort
- ✅ Test and validate connections using port mappings

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 1️⃣ Basic Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment
spec:
  replicas: 5
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: cont1
        image: httpd
```

## 2️⃣ Deployment with Recreate Strategy

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment-re
spec:
  strategy:
    type: Recreate
  replicas: 4
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: cont1
        image: httpd
```

## 3️⃣ Exposing a Pod with NodePort Service

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: webserver
  labels:
    app: httpd
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-np-service
spec:
  type: NodePort
  selector:
    app: httpd
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30001
```

## 4️⃣ Exposing Jenkins Pod with NodePort Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-np-service
spec:
  type: NodePort
  selector:
    youtube: lwm
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30002
---
apiVersion: v1
kind: Pod
metadata:
  name: jenkins-pod
  labels:
    youtube: lwm
spec:
  containers:
  - name: cont1
```

```yaml
        image: jenkins/jenkins
        ports:
        - containerPort: 8080
```

### 5 Pod-to-Pod Connection via ClusterIP Service

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    colour: black
spec:
  containers:
  - name: cont1
    image: httpd
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  name: pod2
  labels:
    colour: pink
spec:
  containers:
  - name: cont1
    image: nginx
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: my-cip-service
spec:
  type: ClusterIP
  selector:
    colour: pink
  ports:
    - port: 9999
      targetPort: 80
```

### 6 Exposing Deployment with LoadBalancer Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lb-service
spec:
  type: LoadBalancer
  selector:
    colour: blue
  ports:
    - port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment
spec:
  replicas: 5
  selector:
    matchLabels:
      colour: blue
  template:
    metadata:
      labels:
        colour: blue
    spec:
      containers:
      - name: cont1
        image: httpd
```

### 7️⃣ Showcasing Selector and Labels with Deployment and Pod

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment
spec:
  replicas: 5
  selector:
    matchLabels:
      colour: blue
  template:
    metadata:
      labels:
        colour: blue
    spec:
      containers:
      - name: cont1
        image: httpd
---
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    colour: blue
spec:
  containers:
  - name: cont1
    image: nginx
    ports:
    - containerPort: 80
```

### 📞 Contact Us

**Phone:** +91 91500 87745

### 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 4 – YAML Reference Guide

In **Part 4**, you will learn how to securely manage user access and permissions inside your EKS cluster using:

## 📚 What You'll Learn

- ✅ `aws-auth` ConfigMap to add IAM users and roles
- ✅ Role-Based Access Control (RBAC) for namespace-specific and cluster-wide access
- ✅ Create `Role`, `RoleBinding`, `ClusterRole`, and `ClusterRoleBinding`
- ✅ Map IAM users to Kubernetes RBAC groups
- ✅ Use **IRSA (IAM Roles for Service Accounts)** to allow Pods to access AWS services like S3 securely
- ✅ Hands-on live demo of a Pod accessing S3 using IRSA

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 1️⃣ aws-auth ConfigMap – Adding IAM Users

```yaml
apiVersion: v1
data:
  mapRoles: |
    - groups:
      - system:bootstrappers
      - system:nodes
      rolearn: arn:aws:iam::992382429239:role/eks-node-group-role
      username: system:node:{{EC2PrivateDNSName}}
  mapUsers: |
    - userarn: arn:aws:iam::992382429239:user/dev-user
      username: dev-user
      groups:
        - dev-group
    - userarn: arn:aws:iam::992382429239:user/readonly-user
      username: readonly-user
      groups:
        - readonly-group
kind: ConfigMap
metadata:
  name: aws-auth
  namespace: kube-system
```

### 2️⃣ Dev User Role and RoleBinding (Namespace-scoped Access)

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: dev-role
  namespace: dev-namespace
rules:
- apiGroups: ["", "apps", "batch"]
  resources: ["pods", "services", "deployments", "jobs"]
  verbs: ["get", "list", "create", "update", "delete", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-binding
  namespace: dev-namespace
subjects:
- kind: Group
  name: dev-group
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-role
  apiGroup: rbac.authorization.k8s.io
```

### 3️⃣ ReadOnly User ClusterRole and ClusterRoleBinding

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: readonly-role
rules:
- apiGroups: ["", "apps", "batch"]
  resources: ["pods", "services", "deployments", "jobs"]
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: readonly-binding
subjects:
- kind: Group
  name: readonly-group
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
```

```
    name: readonly-role
    apiGroup: rbac.authorization.k8s.io
```

## 4️⃣ Pod Accessing S3 Using IRSA

🔹 **Step 1: Enable OIDC Provider for Your EKS Cluster**

```
eksctl utils associate-iam-oidc-provider \
  --region ap-southeast-1 \
  --cluster <cluster-name> \
  --approve
```

🔹 **Step 2: Create IAM Role with Trust Policy for Pod**

🔹 **Step 3: Create Kubernetes ServiceAccount**

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: s3-access
  namespace: default
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::<account-id>:role/EKS_S3_IRSA_Role
```

🔹 **Step 4: Deploy a Pod Using the Service Account**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: s3-reader
spec:
  serviceAccountName: s3-access
  containers:
  - name: awscli
    image: amazonlinux
    command: ["/bin/sh"]
    args: ["-c", "yum install -y aws-cli && aws s3 ls"]
```

🔹 **Step 5: Verify Access to S3 from Pod**

```
2025-06-01 00:00:00 lwm-terraform-bucket
2025-06-01 00:00:00 lwm-kubernetes-bucket
```

## 📞 Contact Us

**Phone:** +91 91500 87745

## 💬 Ask Your Doubts

Join our **Discord Community**
👉 Click here to connect

## 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 Watch Now

# 🚀 Kubernetes Part 5 – YAML Reference Guide

In **Part 5**, you will learn advanced pod scheduling techniques in Kubernetes to control how and where your Pods run within your EKS cluster.

## 📚 What You'll Learn

- ✅ **Node Selector** – Assign Pods to specific nodes using simple labels
- ✅ **Node Affinity** – Fine-grained control over Pod placement using label expressions
- ✅ **Pod Affinity & Anti-Affinity** – Co-locate or separate Pods based on labels and topology
- ✅ **Taints and Tolerations** – Ensure only specific Pods are scheduled on tainted nodes
- ✅ Use **requiredDuringSchedulingIgnoredDuringExecution** and understand its impact
- ✅ Real-world examples with **multiple tolerations** (e.g., green, blue)
- ✅ Hands-on live demo with YAML files showing each scheduling strategy in action

## 🛠️ Cluster Setup: Three Worker Nodes in EKS

We will create three EKS worker nodes with custom labels and taints.

### ✅ Step-by-Step

1. Launch EKS cluster with 3 managed node groups (use console)
2. After nodes are ready, label and taint the nodes using kubectl: (in video the taint and labels are added in console you can also use the below commands)

```
# Label node 1 as general
kubectl label node <node-name-1> node-type=general

# Label and taint node 2 as high-cpu with taint colour=green
kubectl label node <node-name-2> node-type=high-cpu
kubectl taint node <node-name-2> colour=green:NoSchedule

# Label and taint node 3 as high-memory with taint colour=blue
kubectl label node <node-name-3> node-type=high-memory
kubectl taint node <node-name-3> colour=blue:NoSchedule
```

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 1️⃣ Basic Pod on Any Node

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

### 2️⃣ Pod with Required Node Affinity (high-memory)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-intensive-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-type
```

```
            operator: In
            values:
            - high-memory
  containers:
  - name: memory-app
    image: nginx
```

### 3️⃣ Pod with Preferred Node Affinity (high-memory)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: memory-intensive-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: node-type
            operator: In
            values:
            - high-memory
  containers:
  - name: memory-app
    image: nginx
```

### 4️⃣ Pod with Toleration for Taint (colour=green:NoSchedule)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cpu-intensive-pod
spec:
  tolerations:
  - key: "colour"
    operator: "Equal"
    value: "green"
    effect: "NoSchedule"
  containers:
  - name: cpu-app
    image: nginx
```

### 5️⃣ Deployment Tolerating Green Taint

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: green-app-deployment
  labels:
    app: green-app
spec:
  replicas: 5
  selector:
    matchLabels:
      app: green-app
  template:
    metadata:
      labels:
        app: green-app
    spec:
      tolerations:
      - key: "colour"
        operator: "Equal"
        value: "green"
        effect: "NoSchedule"
      containers:
      - name: green-container
        image: nginx
        ports:
        - containerPort: 80
```

### 6️⃣ Deployment Tolerating Both green and blue Taints

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: multicolor-app
  labels:
    app: multicolor-app
spec:
  replicas: 8
  selector:
    matchLabels:
      app: multicolor-app
  template:
    metadata:
      labels:
        app: multicolor-app
    spec:
      tolerations:
      - key: "colour"
        operator: "Equal"
        value: "green"
        effect: "NoSchedule"
      - key: "colour"
        operator: "Equal"
        value: "blue"
        effect: "NoSchedule"
      containers:
      - name: multicolor-container
        image: nginx
        ports:
        - containerPort: 80
```

### 7️⃣ Deployment with Node Affinity and Toleration

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: compute-app-deployment
  labels:
    app: compute-app
spec:
  replicas: 5
  selector:
    matchLabels:
      app: compute-app
  template:
    metadata:
      labels:
        app: compute-app
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: node-type
                operator: In
                values:
                - high-cpu
      tolerations:
      - key: "colour"
        operator: "Equal"
        value: "green"
        effect: "NoSchedule"
      containers:
      - name: compute-container
        image: nginx
        ports:
        - containerPort: 80
```

### 8️⃣ Pod with Label for Pod Affinity Targeting

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

### 9️⃣ Pod Affinity (Schedule with bestfriendz Pods)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: httpd-pod
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            group: bestfriendz
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: cont1
    image: httpd
```

### 1️⃣ 0️⃣ Pod Anti-Affinity (Avoid bestfriendz Pods)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: enemy-pod
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchLabels:
            group: bestfriendz
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: cont1
    image: httpd
```

## 📞 Contact Us

**Phone:** +91 91500 87745

## 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

## 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 6 – YAML Reference Guide

In **part 6**, you'll explore how to persist and share data between Pods using different types of volumes. You'll also understand how to dynamically and statically provision storage in AWS with EBS and EFS.

## 📚 What You'll Learn

- ✅ Difference between `emptyDir`, `hostPath`, `PersistentVolume` (PV) and `PersistentVolumeClaim` (PVC)
- ✅ Use `StorageClass` for dynamic provisioning of **EBS** and **EFS**
- ✅ Understand how `ReadWriteOnce` and `ReadWriteMany` access modes affect Pod mounting
- ✅ Share data between containers using `emptyDir`
- ✅ Mount host machine files into Pods using `hostPath`
- ✅ Use **EBS** volumes with static and dynamic provisioning
- ✅ Use **EFS** for shared, scalable, and dynamic file storage
- ✅ Full hands-on examples with YAML

### 🛠️ Cluster Setup: Enable the Add-on with Pod Identity (via Console)

- 1️⃣ Go to EKS > Your Cluster > Add-ons
- 2️⃣ Click Create Add-on
- 3️⃣ Choose aws-ebs-csi-driver & aws-efs-csi-driver
- 4️⃣ Select latest version
- 5️⃣ For IAM Role, choose:
- 6️⃣ Pod Identity (recommended)
- 7️⃣ Select or create the IAM role: AmazonEKS_EBS_CSI_DriverRole, AmazonEKS_EFS_CSI_DriverRole

# 📦 Kubernetes Volumes – Quick Notes

## ⚠️ Pods are Ephemeral!

==When a Pod is deleted, everything inside it is lost — unless it's backed by a volume.==

## 📁 Volume Types

| Volume Type | Description |
|---|---|
| `emptyDir` | Temporary storage shared between containers in a Pod. Deleted when the Pod is deleted. |
| `hostPath` | Mounts a file or directory from the host node into the Pod. **Not recommended** for production use. |
| **Persistent Volumes** | Backed by real storage from cloud or local systems. Persist beyond Pod lifecycle. |
| ├─ EBS | Amazon Elastic Block Store — supports **RWO** access. |
| └─ EFS | Amazon Elastic File System — supports **RWX** access. |

## ⚙️ Provisioning Modes

| Mode | Description |
|---|---|
| **Static** | Volume is pre-created manually (e.g., via AWS Console) |
| **Dynamic** | Volume is created **automatically** using a `StorageClass` |

## 🔓 Access Modes

| Access Mode | Description | Example Volume |
|---|---|---|
| `ReadWriteOnce` (RWO) | One node can read/write at a time | EBS |
| `ReadOnlyMany` (ROX) | Many nodes can read, but none can write | EFS (rare) |
| `ReadWriteMany` (RWX) | Many nodes can read/write simultaneously | EFS |

| Access Mode | Description | Example Volume |
|---|---|---|
| `ReadWriteOncePod` | Only a single pod (on any node) can access the volume | Rare use case |

## ♻️ Reclaim Policies

| Policy | Description |
|---|---|
| `Retain` | Manual cleanup required. Volume and data are retained. |
| `Recycle` | Performs basic scrub ( `rm -rf /volume/*` ) — deprecated in many environments |
| `Delete` | Deletes the associated storage resource automatically (e.g., EBS volume) |

## 🔁 volumeBindingMode

| Mode | Behavior |
|---|---|
| `WaitForFirstConsumer` | Volume is created **only when Pod is scheduled** (prevents cross-AZ issues) |
| `Immediate` | Volume is created **as soon as PVC is created**, even before pod uses it |

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 1️⃣ Sharing Data Between Containers Using `emptyDir`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example
spec:
  containers:
    - name: writer-cont1
      image: busybox
      command: ["/bin/sh", "-c", "echo 'Hello from writer' > /data/message.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: shared-data
    - name: reader-cont2
      image: busybox
      command: ["/bin/sh", "-c", "cat /data/message.txt && sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: shared-data
  volumes:
    - name: shared-data
      emptyDir: {}
```

### 2️⃣ Accessing Host Files Using hostPath

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-etc-example
spec:
  containers:
    - name: etc-reader
      image: busybox
      command: ["/bin/sh", "-c", "ls /host/etc && sleep 3600"]
      volumeMounts:
        - mountPath: /host/etc
```

```yaml
          name: host-etc
    volumes:
      - name: host-etc
        hostPath:
          path: /etc
          type: Directory
```

## 3️⃣ Dynamic EBS Provisioning Using StorageClass and PVC

```yaml
# StorageClass for EBS
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
---
# PVC for EBS
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
  storageClassName: ebs-sc
---
# Pod using the dynamic EBS volume
apiVersion: v1
kind: Pod
metadata:
  name: ebs-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - mountPath: "/data"
      name: ebs-volume
  volumes:
  - name: ebs-volume
    persistentVolumeClaim:
      claimName: ebs-pvc
```

## 4️⃣ Static EBS Volume with Manual PV and PVC

```yaml
# Static PersistentVolume (replace with actual EBS Volume ID)
apiVersion: v1
kind: PersistentVolume
metadata:
  name: static-ebs-pv
spec:
  capacity:
    storage: 3Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: manual
  csi:
    driver: ebs.csi.aws.com
    volumeHandle: vol-xxxxxxxxxxxxx     # Replace EBS Volume id
---
# PersistentVolumeClaim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: static-ebs-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: manual
```

```yaml
  resources:
    requests:
      storage: 3Gi
---
# Pod using the static EBS volume
apiVersion: v1
kind: Pod
metadata:
  name: ebs-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts:
    - mountPath: "/data"
      name: ebs-volume
  volumes:
  - name: ebs-volume
    persistentVolumeClaim:
      claimName: static-ebs-pvc
```

## 5️⃣ Dynamic EFS Volume Provisioning with StorageClass and PVC

```yaml
# StorageClass for EFS
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap
  fileSystemId: fs-0f8431a1f00634456  # Replace with your EFS ID
  directoryPerms: "700"
  gidRangeStart: "1000"
  gidRangeEnd: "2000"
  basePath: "/dynamic_provisioning"
---
# PVC for EFS
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
  storageClassName: efs-sc
---
# Pod using EFS volume
apiVersion: v1
kind: Pod
metadata:
  name: efs-test-pod
spec:
  containers:
    - name: app
      image: busybox
      command: [ "/bin/sh", "-c", "echo EFS volume is working > /data/hello.txt && sleep 3600" ]
      volumeMounts:
        - name: efs-volume
          mountPath: /data
  volumes:
    - name: efs-volume
      persistentVolumeClaim:
        claimName: efs-pvc
```

## 📞 Contact Us

**Phone:** +91 91500 87745

## 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚦 Kubernetes Part 7 – Probes, Init Containers, Sidecars & Environment Variables

## 📚 What You Will Learn

In **Day 7**, you'll learn how Kubernetes manages **application health**, **startup sequences**, and **shared responsibilities** between containers. This session is packed with **live examples** and advanced techniques.

✅ Understand how **Liveness**, **Readiness**, and **Startup** probes work
✅ Configure probes to detect failures, slow startups, and readiness checks
✅ Use **Init Containers** for setup tasks before app starts
✅ Implement the **Sidecar pattern** to enhance or monitor your main container
✅ Inject configuration using **Environment Variables**, **ConfigMaps**, and **Secrets**
✅ Mount secrets as files and access securely

## 🔧 How to Use

👉 You can copy all YAMLs below into a file like `main.yaml` and run:

```
kubectl apply -f main.yaml
```

### 🧪 1️⃣ Pod example with All Three Probes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: probe-demo
spec:
  containers:
  - name: demo-app
    image: httpd:latest
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 3
    startupProbe:
      httpGet:
        path: /startup
        port: 8080
      failureThreshold: 30
      periodSeconds: 10
```

## ⏱️ Probe Timing Example (Understanding Failure/Success)

Let's say you have this probe configuration:

```yaml
initialDelaySeconds: 15
periodSeconds: 30
failureThreshold: 4
successThreshold: 2
timeoutSeconds: 10
```

| Time | Event | Result |
|---|---|---|
| 2:00:00 PM | Pod Started | |
| 2:00:15 PM | 1st Check (Fail) | ❌ |
| 2:00:45 PM | 2nd Check (Fail) | ❌ |
| 2:01:15 PM | 3rd Check (Fail) | ❌ |
| 2:01:45 PM | 4th Check (Fail) | ❌ Pod Restart |
| 2:02:15 PM | 5th Check (Success) | ✅ |
| 2:02:45 PM | 6th Check (Success) | ✅ (Healthy again) |

> 💡 Note: The probe starts only after the initialDelaySeconds. If failureThreshold is reached with consecutive failures, the container will be restarted. After restart, the container must pass successThreshold consecutive checks to be marked as Ready.

### 🧪 2️⃣ Liveness Probe with NGINX

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 5
      failureThreshold: 3
```

### 🧪 3️⃣ Liveness Probe with Exec (BusyBox)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-busybox
spec:
  containers:
  - name: busybox
    image: busybox:latest
    command: ["sh", "-c", "touch /tmp/healthy && sleep 3600"]
    livenessProbe:
      exec:
        command: ["cat", "/tmp/healthy"]
      initialDelaySeconds: 5
      periodSeconds: 10
      failureThreshold: 3
```

### 🏁 4️⃣ Init Container Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: app
    image: busybox
```

```yaml
    command: ['sh', '-c', 'echo "App started!" && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'echo "Initializing..."; sleep 5']
```

## 🤝 5️⃣ Sidecar Pattern – Logging Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ['sh', '-c', 'echo "Writing log..." && while true; do echo log >> /shared/log.txt; sleep 5; done']
    volumeMounts:
    - name: shared-logs
      mountPath: /shared

  - name: log-watcher
    image: busybox
    command: ['sh', '-c', 'tail -f /shared/log.txt']
    volumeMounts:
    - name: shared-logs
      mountPath: /shared

  volumes:
  - name: shared-logs
    emptyDir: {}
```

## 🌿 6️⃣ Using Environment Variables

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-demo
spec:
  containers:
  - name: demo
    image: busybox
    command: ["sh", "-c", "echo $ENV_MSG && sleep 3600"]
    env:
    - name: ENV_MSG
      value: "Hello from LearnWithMithran Youtube!"
```

## ⚙️ 7️⃣ ConfigMap and Secret as envFrom

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLR: blue
  APP_MODE: prod
---
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  Channel: TGVhcldpdGhNaXRocmFuCg==
  Owner: TWl0aHJhbgo=
---
apiVersion: v1
kind: Pod
metadata:
  name: simple-webcolor-pod
spec:
  containers:
  - name: simple-webcolor-pod
    image: httpd
```

```
      ports:
      - containerPort: 80
      envFrom:
      - configMapRef:
          name: app-config
      - secretRef:
          name: app-secret
```

### 🔐 8️⃣ Secrets as Environment Variable and Mounted File

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: demo-secret
data:
  password: dXBkYXRlZDQ1Ngo=
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
  - name: demo
    image: busybox
    command: ["/bin/sh", "-c", "while true; do echo ENV: $PASSWORD; echo FILE: $(cat /etc/secret/password); sleep 10; done"]
    env:
    - name: PASSWORD
      valueFrom:
        secretKeyRef:
          name: demo-secret
          key: password
    volumeMounts:
    - name: secret-vol
      mountPath: /etc/secret
      readOnly: true
  volumes:
  - name: secret-vol
    secret:
      secretName: demo-secret
```

### 📞 Contact Us

**Phone:** +91 91500 87745

### 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 8 – AWS EKS ALB Ingress Controller

## 📚 What You Will Learn

In **Day 8**, you'll learn how to configure advanced Kubernetes networking and traffic routing using the **AWS ALB Ingress Controller** and **Helm**. This session helps you build real-world multi-team infrastructure in an Amazon EKS cluster.

✅ Install and configure eksctl and Helm for production-grade clusters ✅ Deploy the AWS Load Balancer Controller using Helm and IAM integration ✅ Create and expose applications using Kubernetes Ingress ✅ Set up path-based routing with a shared ALB across teams ✅ Deploy workloads in isolated namespaces (e.g., team-a, team-b) ✅ Use annotations to control Ingress behavior, grouping, and routing ✅ Understand the difference between LoadBalancer and Ingress services ✅ Deploy multi-container apps and manage traffic securely and efficiently

## ❗ Why Not Use LoadBalancer Services Alone?

Using Kubernetes Service of type LoadBalancer works well for exposing single services, but it has major limitations:

### ⚠️ Drawbacks of LoadBalancer Service

- 🚫 One ALB per Service → costly in production



- 🧱 No support for path-based or host-based routing
- 🧍 Harder to manage for multiple teams/projects
- 🔐 Difficult to apply centralized security policies

## 🧪 Example Issue with LoadBalancer Services

```yaml
apiVersion: v1
kind: Service
metadata:
  name: lb-service-1
spec:
  type: LoadBalancer
  selector:
    colour: blue
  ports:
    - port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment-1
spec:
  replicas: 5
  selector:
    matchLabels:
      colour: blue
  template:
    metadata:
      labels:
        colour: blue
    spec:
      containers:
      - name: cont1
        image: httpd
---
apiVersion: v1
kind: Service
metadata:
  name: lb-service-2
spec:
  type: LoadBalancer
  selector:
    colour: black
  ports:
    - port: 80
      targetPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: lwm-deployment-2
spec:
  replicas: 5
  selector:
    matchLabels:
      colour: black
  template:
    metadata:
      labels:
        colour: black
    spec:
      containers:
```

```
    - name: cont1
      image: nginx
```

## ✅ Why Use Ingress with ALB Controller?

Ingress with ALB controller provides:

- ✅ Single ALB for multiple apps (shared ALB)
- 🔁 Path-based routing (/app1, /app2, etc.)
- 👫 Team-level separation via namespaces & ingress group
- 💸 Cost savings – fewer ALBs created
- 🔐 Centralized control over routing & TLS termination
- ☁️ Better AWS-native integration via Load Balancer Controller

> ✅ Recommendation: Use Ingress + AWS Load Balancer Controller for scalable, secure, and cost-efficient EKS routing.

## ⚙️ Prerequisites to Install ALB Ingress Controller

- AWS CLI configured
- IAM user with appropriate EKS permissions
- `kubectl` installed
- `eksctl` and `helm` installed

### 🧱 eksctl Setup

```
# Install eksctl
ARCH=amd64
PLATFORM=$(uname -s)_$ARCH

curl -sLO "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl_$PLATFORM.tar.gz"
curl -sL "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl_checksums.txt" | grep $PLATFORM | sha256sum --chec

tar -xzf eksctl_$PLATFORM.tar.gz -C /tmp && rm eksctl_$PLATFORM.tar.gz
sudo install -m 0755 /tmp/eksctl /usr/local/bin && rm /tmp/eksctl
```

### 🔐 OIDC and IAM Setup

```
eksctl utils associate-iam-oidc-provider \
  --region=ap-southeast-1 \
  --cluster=LearnWithMithran2 \
  --approve
```

### 📥 Download the IAM policy:

[Click Me for IAM Json Policy]

Create the IAM policy in AWS as: `AWSLoadBalancerControllerIAMPolicy2`

### 📥 Create service account:

```
eksctl create iamserviceaccount \
  --cluster LearnWithMithran2 \
  --region ap-southeast-1 \
  --namespace kube-system \
  --name aws-load-balancer-controller2 \
  --attach-policy-arn arn:aws:iam::992382429239:policy/AWSLoadBalancerControllerIAMPolicy2 \
  --approve
```

### 📦 Helm Setup

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

### 📡 Add and update repo:

```
helm repo add eks https://aws.github.io/eks-charts
helm repo update
```

### 🚀 Install AWS ALB Ingress Controller

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=LearnWithMithran \
  --set region=ap-southeast-1 \
  --set vpcId=vpc-020d3da5a9615c591 \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller2
```

## 🌐 Ingress Setup: Basic Echo App

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
    spec:
      containers:
      - name: hello
        image: hashicorp/http-echo
        args: ["-text=Hello from app"]
        ports:
        - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: hello-service
spec:
  selector:
    app: hello
  ports:
  - port: 80
    targetPort: 5678
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-ingress
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
spec:
  rules:
  - http:
      paths:
      - path: /LWM
        pathType: Prefix
        backend:
          service:
            name: hello-service
            port:
              number: 80
```

## 👥 Multi-Team Namespaces with Shared ALB

### Create Namespaces

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: team-a
---
apiVersion: v1
kind: Namespace
metadata:
  name: team-b
```

### Team A

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-a
  namespace: team-a
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-a
  template:
    metadata:
      labels:
        app: app-a
    spec:
      containers:
      - name: app-a
        image: hashicorp/http-echo
        args: ["-text=Hello from App A LWM"]
        ports:
        - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: app-a-svc
  namespace: team-a
spec:
  selector:
    app: app-a
  ports:
  - port: 80
    targetPort: 5678
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-a-ingress
  namespace: team-a
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: shared-alb
spec:
  rules:
  - http:
      paths:
      - path: /a
        pathType: Prefix
        backend:
          service:
            name: app-a-svc
            port:
              number: 80
```

### Team B

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-b
  namespace: team-b
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app-b
  template:
    metadata:
      labels:
        app: app-b
    spec:
      containers:
      - name: app-b
        image: hashicorp/http-echo
        args: ["-text=Hello from App B LWM"]
        ports:
        - containerPort: 5678
---
apiVersion: v1
kind: Service
metadata:
  name: app-b-svc
  namespace: team-b
spec:
  selector:
    app: app-b
  ports:
  - port: 80
    targetPort: 5678
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-b-ingress
  namespace: team-b
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: shared-alb
spec:
  rules:
  - http:
      paths:
      - path: /b
        pathType: Prefix
        backend:
          service:
            name: app-b-svc
            port:
              number: 80
```

## 📌 Notes

- Ingress class should match your ALB controller: kubernetes.io/ingress.class: alb
- Shared ALB group enables multi-team routing via paths
- Ensure VPC ID and IAM policies are correctly set in the helm install step

### 📞 Contact Us

**Phone:** +91 91500 87745

### 💬 Ask Your Doubts

Join our **Discord Community**
👉 [Click here to connect]

### 📺 Explore More Learning

Subscribe to our **YouTube Channel** – *Learn With Mithran*
🎯 [Watch Now]

# 🚀 Kubernetes Part 9 – Jobs, CronJobs, StatefulSets, HPA, and ExternalName Services

## 📚 What You Will Learn

In **Day 9**, we deep dive into *Kubernetes workload types and resource management*. These examples are hands-on and ideal for mastering real-world use cases with **Jobs, CronJobs, StatefulSets, ExternalName, Resource Requests/Limits, and Horizontal Pod Autoscaling (HPA)**.

✅ Create one-time Jobs and scheduled CronJobs ✅ Understand StatefulSet and Headless Services (with and without volumes) ✅ Simulate FQDN with ExternalName services ✅ Apply resource requests and limits to containers ✅ Configure HPA using the Kubernetes Metrics Server ✅ Deploy multi-replica applications using Deployments and Services ✅ Run test pods and simulate DNS-based access

## 🛠️ Kubernetes Job and CronJob – One-time and Recurring Tasks

Jobs and CronJobs help run short-lived or recurring background tasks in Kubernetes.

### 🔨 What is a Job?

A Job creates one or more pods and ensures that a specific task runs to completion. Once the task finishes successfully, the Job is considered complete.

✅ Use when you need to:

- Run a one-time task
- Perform data migration or setup scripts
- Process a batch job or backup operation

#### 🔧 Example YAML – Job

```
# One-time Job
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-job
spec:
  template:
    spec:
      containers:
      - name: hello
        image: busybox
        command: ["echo", "Hello from Kubernetes Job"]
      restartPolicy: Never
  backoffLimit: 2
```

### ⏰ What is a CronJob?

A CronJob creates Jobs on a schedule, like a traditional Linux cron. It's ideal for repeating tasks such as log rotation, cleanup, backups, etc.

✅ Use when you need to:

- Run a task at specific time intervals
- Automate recurring operations
- Perform periodic cleanups or reports

#### 🔧 Example YAML – CronJob

```
# Recurring CronJob
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cleanup-task
```

```yaml
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: cleanup
            image: busybox
            command: ["sh", "-c", "echo Cleaning temp files at $(date)"]
          restartPolicy: OnFailure
```

### 🆚 Jobs vs CronJobs – Comparison Table

| Feature | Job | CronJob |
|---|---|---|
| **Run Type** | One-time execution | Recurring on schedule |
| **Scheduling** | Immediate or manual | CRON-based (e.g., every 5 mins) |
| **Use Case** | Backup, data migration, processing one file | Daily reports, cleanup tasks |
| **Pod Lifecycle** | Runs once then terminates | Triggers Job objects at intervals |
| **Built-in Retry** | ✅ Yes ( `backoffLimit` ) | ✅ Yes, inherited from Job spec |

## ⚔️ Deployments vs StatefulSets

Kubernetes provides different workload types for managing application lifecycles. Two of the most common are Deployments and StatefulSets.

### ✅ When to Use Deployments

- Your application is stateless
- You want easy horizontal scaling
- Any pod can serve any request (e.g., web servers, REST APIs, frontend apps)
- Persistent storage is not required or shared storage is enough

### ✅ When to Use StatefulSets

- Each pod needs a stable identity
- You want stable persistent storage per pod
- Pods need stable DNS names
- You require ordered, graceful deployment, scaling, or deletion
- Use cases include databases (MySQL, MongoDB), queues, Kafka, Zookeeper

### 🔍 What's the Difference?

| Feature | Deployment | StatefulSet |
|---|---|---|
| **Pod Name** | Dynamic, auto-generated (e.g., `app-xyz123` ) | Stable and predictable ( `app-0` , `app-1` ) |
| **Pod Identity** | All pods are identical (no unique identity) | Each pod has a **unique, stable identity** |
| **Storage** | Shared or ephemeral storage | Dedicated persistent volume per pod |
| **Network Identity (DNS)** | Shared service FQDN (e.g., `svc.default.svc.cluster.local` ) | Unique FQDN per pod (e.g., `pod-0.svc.default.svc.cluster.local` ) |
| **Scaling Behavior** | All pods are interchangeable | Pods are created **sequentially** and deleted in **reverse order** |
| **Use Case** | Stateless applications | Stateful applications |

## 🧱 StatefulSet With Headless Service (Without Volumes)

```yaml
# Headless Service
apiVersion: v1
kind: Service
metadata:
  name: nginx-headless
```

```yaml
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
  - port: 80
    name: http
---
# StatefulSet
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: "nginx-headless"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

## 🧱 Deployment With ClusterIp Service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deploy
spec:
  replicas: 5
  selector:
    matchLabels:
      colour: blue
  template:
    metadata:
      labels:
        colour: blue
    spec:
      containers:
      - name: cont1
        image: httpd
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: httpd-svc
spec:
  selector:
    colour: blue
  ports:
  - port: 80
    targetPort: 80
```

## 🔄 StatefulSet vs Deployment – FQDN Behavior and Identity

When running microservices in Kubernetes, how services are addressed (FQDN) matters for stable communication, especially in distributed systems like databases or stateful workloads.

| Feature | Deployment | StatefulSet |
|---|---|---|
| **Pod Name** | Random (e.g., `httpd-deploy-5f89dfd9cc-abcde` ) | Fixed (e.g., `nginx-0` , `nginx-1` , `nginx-2` ) |
| **ClusterIP** | Assigned (e.g., `10.100.247.159` ) | `None` (Headless Service) |

| Feature | Deployment | StatefulSet |
|---|---|---|
| Pod IP | Dynamic (changes on restart) | Dynamic (still can change, but FQDN is stable) |
| FQDN Format | `httpd-svc.default.svc.cluster.local` | `nginx-0.nginx-headless.default.svc.cluster.local` , `nginx-1...` |
| Stable DNS Identity | ❌ No stable per-pod DNS | ✅ Stable and addressable via predictable FQDN |
| Use Case | Stateless apps (e.g., web servers) | Stateful apps (e.g., DBs, queues) |

## 📘 Example: FQDN Usage

```
# Deployment
FQDN → httpd-svc.default.svc.cluster.local
Pod IP → Changes on restart (e.g., 172.31.4.118)

# StatefulSet
FQDNs →
- nginx-0.nginx-headless.default.svc.cluster.local
- nginx-1.nginx-headless.default.svc.cluster.local
- nginx-2.nginx-headless.default.svc.cluster.local

Pod IP → May change, but name-based resolution is stable
```

## 🧪 Testing DNS FQDN from a dummy Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dns-test
spec:
  containers:
  - name: test
    image: curlimages/curl
    command: ["sleep", "3600"]
```

## 📦 StatefulSet With Persistent Volumes ==(need PV & PVC)==

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-headless
spec:
  clusterIP: None
  selector:
    app: redis
  ports:
  - port: 6379
    name: redis
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis-headless"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis
        ports:
```

```
        - containerPort: 6379
        volumeMounts:
        - name: redis-data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: redis-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

# 🌐 Kubernetes ExternalName Service – Access External Resources via DNS

## 🔍 What is an ExternalName Service?

An ExternalName service is a special type of Kubernetes Service that allows pods inside the cluster to access an external service (outside the cluster) using an internal DNS name.

Instead of routing traffic to internal pods or endpoints, it returns a CNAME (DNS alias) pointing to an external FQDN.

## 🔧 Example Use Cases

✅ Allow applications to access:

- An external database (e.g., RDS, MongoDB Atlas)
- A legacy app running outside Kubernetes (e.g., on EC2)
- Third-party APIs or SaaS systems (e.g., Google, Stripe)

### 🛠️ Sample YAML – ExternalName to EC2 Endpoint

```
# EC2-hosted app
apiVersion: v1
kind: Service
metadata:
  name: lwm-svc
spec:
  type: ExternalName
  externalName: ec2-52-77-254-161.ap-southeast-1.compute.amazonaws.com
```

### 🔧 Sample YAML – ExternalName to Public Website (Google)

```
# Google ExternalName (for test/demo only)
apiVersion: v1
kind: Service
metadata:
  name: google-svc
spec:
  type: ExternalName
  externalName: www.google.com
```

# ⚙️ Kubernetes Resource Requests and Limits – Manage Pod Resource Usage

## 🔍 What Are Requests and Limits?

Kubernetes allows you to control how much CPU and memory a container can request and use through requests and limits. This helps ensure fair resource allocation, avoid overconsumption, and enable auto-scaling.

## 🧠 Key Concepts

| Term | Meaning |
|---|---|
| **Request** | The **minimum** amount of CPU/memory guaranteed to the container |
| **Limit** | The **maximum** amount of CPU/memory the container is allowed to use |

| Term | Meaning |
| --- | --- |
| **Eviction** | If a pod exceeds limits or system is under pressure, it may be evicted |
| **Throttling** | CPU usage above limit is throttled (not evicted) |

🔧 **Example YAML – Resource Request and Limit**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-rl
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        cpu: "100m"
        memory: "128Mi"
      limits:
        cpu: "200m"
        memory: "256Mi"
```

💡 **CPU Units Explained**

| Value | Meaning |
| --- | --- |
| `1` | 1 full CPU core |
| `500m` | 0.5 CPU core |
| `100m` | 0.1 CPU core |

💡 **Memory Units**

Common formats: Mi, Gi, M, G

Example: 128Mi = 128 Mebibytes

### ⚙️ What Happens at Runtime?

- If the node runs out of resources:
  - Pods may be evicted if they exceed memory limits
  - CPU usage will be throttled if it exceeds CPU limits
- Requests help the scheduler decide where to place pods

### 🧪 Real-World Use Case

| Scenario | Request | Limit |
| --- | --- | --- |
| Small API | 100m CPU / 128Mi RAM | 200m CPU / 256Mi RAM |
| Heavy Worker | 500m CPU / 512Mi RAM | 1 CPU / 1Gi RAM |
| Memory-intensive Batch Job | 256Mi RAM | 1Gi RAM |

## 📈 HPA vs VPA – Kubernetes Auto Scaling Demystified

In Kubernetes, auto-scaling ensures that your application has the right number of resources and replicas based on real-time demand. The two main types of autoscalers are:

- **Horizontal Pod Autoscaler (HPA)** – scales the number of pod replicas
- **Vertical Pod Autoscaler (VPA)** – adjusts the resource requests/limits for each pod

### 🔄 What is HPA (Horizontal Pod Autoscaler)?

HPA automatically adjusts the number of pods in a Deployment, ReplicaSet, or StatefulSet based on observed CPU/memory usage or custom metrics.

### ✅ Use when:

- You want to scale out/in based on traffic/load
- Your app is stateless
- Metrics are available via Metrics Server

#### 🔧 Example YAML – HPA

```yaml
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
      - name: php-apache
        image: registry.k8s.io/hpa-example
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
          requests:
            cpu: 200m
---
# Service
apiVersion: v1
kind: Service
metadata:
  name: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
---
# HPA (Metrics Server Required)
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

### 🧱 What is VPA (Vertical Pod Autoscaler)?

VPA automatically adjusts the CPU and memory requests/limits of containers in a pod to better match the actual usage.

### ✅ Use when:

- You want to optimize resource usage per pod
- Your app does not scale well horizontally
- You need to reduce over-provisioning

#### 🔧 Modes in VPA:

| Mode | Behavior |
|---|---|
| `Off` | Just monitor; no action |

| Mode | Behavior |
|---|---|
| `Auto` | Actively updates pod resources |
| `Initial` | Only sets values on pod creation |

**Example YAML – VPA**

```yaml
apiVersion: autoscaling.k8s.io/v1
```

### Releases

No releases published

### Packages

No packages published