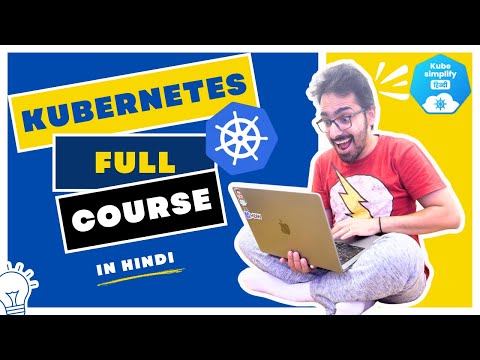
<https://www.youtube.com/watch?v=5NXmbV50IxE> -- Kubernetes Course in Hindi (13 Hours) | Full Hands-On Experience

[](https://www.youtube.com/watch?v=5NXmbV50IxE)

<https://www.youtube.com/watch?v=MGCF6slXG0w>

[](https://www.youtube.com/watch?v=MGCF6slXG0w)

ALMALinux -- Alma Linux is a free and open source linux distribution

Java Programming for Complete Begineers -- By in28Minutes official

Dr. Angela Yu, Developer and Lead Instructor - Udemy Instructor Partner

Terraform on Azure with IaC DevOps SRE | Real-World 25 Demos by Kalyan Reddy Daida

GITLAB CI --- How to integrate.

Ansible for automating tasks.

Monitoring in Linux -- last command is used.

<https://www.udemy.com/course/certified-kubernetes-administrator-with-practice-tests/?couponCode=24T4MT120424>

# **Certified Kubernetes Administrator (CKA) with Practice Tests**

By [**Mumshad Mannambeth**](https://www.udemy.com/user/mumshad-mannambeth/)

<https://www.udemy.com/course/aws-certified-devops-engineer-professional-hands-on/?couponCode=24T4MT120424>

# **AWS Certified DevOps Engineer Professional 2025 - DOP-C02**

<https://www.udemy.com/course/aws-certified-devops-engineer-professional-hands-on/?couponCode=24T4MT120424#instructor-1>

Admiralty is a Kubernetes plugin designed for **multi-cluster management**, allowing users to deploy applications across multiple Kubernetes clusters efficiently. It simplifies the process of managing workloads, traffic routing, and disaster recovery across clusters, making it particularly useful for organizations operating in diverse environments, including cloud and on-premises setups.

## **Key Features of Admiralty**

* **Multi-Cluster Deployment**: Deploy applications across various clusters seamlessly.
* **Traffic Optimization**: Ensures low latency and instant failover.
* **Centralized Management**: Provides a single control plane for managing user access and audit logs.
* **Flexible Architecture**: Supports both centralized and decentralized cluster topologies.
* **Disaster Recovery**: Facilitates active-active disaster recovery strategies.

## **Steps to Install and Configure Admiralty Plugin in Kubernetes**

### **Prerequisites**

1. Ensure you have a working Kubernetes cluster.
2. Install kubectl and helm on your local machine.
3. Have access to the Kubernetes API with appropriate permissions.

### **Installation Steps**

1. **Add the Admiralty Helm Repository**:

helm repo add admiralty oci://public.ecr.aws/admiralty/admiralty  
helm repo update

1. **Install Admiralty in Each Cluster**: Use the following command to install Admiralty in your desired clusters:

for CLUSTER\_NAME in <your-cluster-names>; do  
 sudo helm install admiralty oci://public.ecr.aws/admiralty/admiralty \  
 --kube-context kind-$CLUSTER\_NAME \  
 --namespace admiralty --create-namespace \  
 --version 0.16.0 --wait --debug  
done

Replace <your-cluster-names> with the actual names of your clusters.

1. **Set Up Cross-Cluster Authentication**:
   1. Create a Kubernetes service account in each workload cluster for the management cluster.
   2. Generate a token for this service account.
   3. Obtain a routable Kubernetes API address from the management cluster.
   4. Prepare a kubeconfig file using the token and address, then save this kubeconfig as a secret in the management cluster.
2. **Configure Workloads**: Annotate your pods with multicluster.admiralty.io/elect="" to enable workload distribution across clusters.
3. **Verify Installation**: Check that all components are running correctly by executing:

kubectl get pods -n admiralty

1. **Monitor and Manage**: Use kubectl commands to monitor the status of your deployments and manage workloads across clusters effectively.

By following these steps, you can successfully install and configure the Admiralty plugin in your Kubernetes environment, enabling efficient multi-cluster management and deployment capabilities

Calico is an open-source networking and network security solution designed for Kubernetes and other container orchestration platforms. It provides robust networking capabilities, including IP address management, network policies, and security features, making it a popular choice for organizations looking to enhance their Kubernetes environments.

## **Uses of Calico in Kubernetes**

* **Networking**: Calico enables a flat networking model where each pod receives its own IP address, allowing direct communication without the need for Network Address Translation (NAT).
* **Network Policies**: It supports Kubernetes network policies and extends them with additional features for fine-grained control over traffic between pods, enhancing security through microsegmentation.
* **Performance**: Calico can leverage eBPF (Extended Berkeley Packet Filter) or standard Linux networking for high-performance networking, suitable for large-scale deployments.
* **Multi-Tenancy**: It facilitates network segmentation in multi-tenant environments, allowing organizations to isolate workloads and enforce strict security policies.

## **Installation and Configuration of Calico in Kubernetes**

### **Prerequisites**

* A running Kubernetes cluster.
* Access to kubectl command-line tool.

### **Installation Steps**

1. **Download the Calico Manifest**: Use the following command to download the Calico manifest file:

curl <https://docs.projectcalico.org/manifests/calico.yaml> -O

1. **Apply the Manifest**: Deploy Calico by applying the downloaded manifest:

kubectl apply -f calico.yaml

1. **Verify Installation**: Check that the Calico node agent is running on all nodes:

kubectl get pods -n kube-system -l k8s-app=calico-node

1. **Create a Sample Deployment**: For demonstration, create a simple Nginx deployment:

kubectl create deployment nginx --image=nginx

1. **Expose the Deployment**: Expose the Nginx deployment as a ClusterIP service:

kubectl expose deployment nginx --port=80 --type=ClusterIP

1. **Verify Pod Networking**: Check that Calico has assigned an IP address to the Nginx pod and set up necessary routes:

kubectl get pods -l app=nginx -o wide

### **Example of Using Network Policies**

To demonstrate Calico's network policy capabilities, you can create a policy that restricts access to the Nginx pod:

1. **Define a Network Policy**: Create a YAML file named nginx-network-policy.yaml with the following content:

apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: deny-all-ingress  
 namespace: default  
spec:  
 podSelector:  
 matchLabels:  
 app: nginx  
 policyTypes:  
 - Ingress  
 ingress: []

1. **Apply the Network Policy**: Apply the policy to enforce restrictions on incoming traffic to the Nginx pod:

kubectl apply -f nginx-network-policy.yaml

This configuration establishes a basic setup of Calico in your Kubernetes environment, showcasing its networking capabilities and how to implement network policies for enhanced security.

Kubernetes with Kind

Etcd stores the data in Key-Value format. And stores the status of pods.

All the request comes to API Server. As scheduler contineously runs and check the status of nodes. It will check which node is used less. So he can transfer new request to him.

API-Gateway talks to etcd and tell them to create pods. Kubeproxy is used for running POD and also check its status. IP tables handle by kubeproxy.

Application runs inside PODs.

Deployments controlled by Deployment Controller. ReplicaSet controlled by Replicaset controller.

Install “Kind” on Windows for Kubernetes.

To install Kind (Kubernetes in Docker) on Windows and create a running pod with master-slave architecture, follow these detailed steps:

## **Step 1: Install Prerequisites**

1. **Install Docker**:
   1. Download Docker Desktop for Windows from the official Docker website.
   2. Follow the installation prompts, ensuring that WSL 2 is enabled if you are using Windows 10 Home.
2. **Install Chocolatey** (optional but recommended):
   1. Open PowerShell as an administrator and run:Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

## **Step 2: Install Kind**

### **Method 1: Using Chocolatey**

1. Open a new PowerShell window and run:choco install kind -y

### **Method 2: Direct Download**

1. In PowerShell, run the following command to download Kind:curl.exe -Lo kind-windows-amd64.exe <https://kind.sigs.k8s.io/dl/v0.14.0/kind-windows-amd64>
2. Move the executable to a directory in your PATH:Move-Item .\kind-windows-amd64.exe C:\some-dir-in-your-PATH\kind.exe

## **Step 3: Verify Installation**

Confirm that Kind is installed correctly by running:

kind --version

## **Step 4: Create a Kubernetes Cluster**

To create a Kubernetes cluster with Kind, execute:

kind create cluster

This command will create a single-node cluster named "kind" by default.

## **Step 5: Create a Pod**

To create a running pod, you can use a simple YAML configuration file. Create a file named pod.yaml with the following content:

apiVersion: v1  
kind: Pod  
metadata:  
 name: my-pod  
spec:  
 containers:  
 - name: my-container  
 image: nginx  
 ports:  
 - containerPort: 80

Then, apply this configuration to create the pod:

kubectl apply -f pod.yaml

## **Step 6: Create a Master-Slave Setup**

To set up a master-slave architecture (for example, using two pods), you can define a Deployment with replicas. Create a file named deployment.yaml with the following content:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: my-deployment  
spec:  
 replicas: 2 # This creates one master and one slave pod  
 selector:  
 matchLabels:  
 app: my-app  
 template:  
 metadata:  
 labels:  
 app: my-app  
 spec:  
 containers:  
 - name: my-container  
 image: nginx  
 ports:  
 - containerPort: 80

Apply this deployment configuration with:

kubectl apply -f deployment.yaml

## **Step 7: Verify Pods**

To check if your pods are running, use the following command:

kubectl get pods

You should see your master and slave pods listed.

## **Conclusion**

You have successfully installed Kind on Windows, created a Kubernetes cluster, deployed a pod, and set up a master-slave architecture using Kubernetes Deployments. This setup is ideal for local development and testing purposes.

IBM DevOps and Software Engineering Professional Certificate – Coursera

[IBM DevOps and Software Engineering Professional Certificate | Coursera](https://www.coursera.org/professional-certificates/devops-and-software-engineering?)

How to deploy envoy load balancer and also steps to confgure it with current kubernetes master and slaves.

Metrics Server Kubernetes ---

Metrics Server is a crucial component in Kubernetes that provides resource metrics for pods and nodes, primarily for autoscaling purposes. Here’s a comprehensive overview of why you should use it and how to set it up using a Helm chart.

## **Why Use Metrics Server?**

1. **Autoscaling Support**: Metrics Server collects resource metrics (CPU and memory) from Kubelets and exposes them via the Kubernetes API. This data is essential for the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) to dynamically scale workloads based on demand.
2. **Resource Efficiency**: It has a minimal resource footprint, typically requiring only about 1 milli-core of CPU and 2 MB of memory per node, making it suitable for clusters with up to 5,000 nodes.
3. **Real-time Metrics**: Metrics Server collects metrics every 15 seconds, allowing for timely adjustments to resource allocations as workloads change.
4. **Debugging Tool**: The metrics can also be accessed using the kubectl top command, which aids in debugging and monitoring cluster performance.

## **How to Create and Configure Metrics Server Using Helm**

### **Prerequisites**

* A running Kubernetes cluster (version 1.4 or later).
* Helm installed and configured to manage your Kubernetes applications.

### **Step-by-Step Installation**

#### **1. Add the Metrics Server Helm Repository**

Run the following command to add the official Metrics Server repository:

helm repo add metrics-server <https://kubernetes-sigs.github.io/metrics-server/>

#### **2. Install Metrics Server**

To install Metrics Server, execute:

helm install metrics-server metrics-server/metrics-server

If you need a specific version, specify it as follows:

helm install metrics-server metrics-server/metrics-server --version <version>

Replace <version> with your desired version number.

#### **3. Verify Installation**

Check the status of the installation:

helm ls -n default

You should see metrics-server listed with a status of deployed. To confirm that the pods are running, use:

kubectl get pods -n kube-system

#### **4. Configuration Options**

You may need to customize certain configurations based on your cluster setup. Common flags include:

* --kubelet-preferred-address-types: Adjusts the preferred address types used when connecting to nodes.
* --kubelet-insecure-tls: Disables TLS verification for Kubelet connections (useful in non-production environments).

To apply these configurations, create a values.yaml file with your desired settings or pass them directly in the command line during installation.

#### **5. Upgrading or Uninstalling**

To upgrade Metrics Server:

helm upgrade metrics-server metrics-server/metrics-server --version <new-version>

To uninstall it:

helm uninstall metrics-server -n kube-system

If you wish to delete the associated namespace, run:

kubectl delete ns kube-system

## **Conclusion**

Deploying and configuring Metrics Server in your Kubernetes cluster using Helm is straightforward and offers significant benefits for autoscaling applications. By following these steps, you can ensure that your cluster efficiently manages resources based on real-time usage data, enhancing performance and reliability.

HPA [Horizontal Pod AutoScaler Scalling] ---

Horizontal Pod Autoscaler (HPA) in Kubernetes is a powerful feature that automatically adjusts the number of pod replicas in a deployment based on observed metrics, such as CPU utilization or memory consumption. This capability is essential for maintaining application performance and resource efficiency in dynamic environments.

## **Why Use Horizontal Pod Autoscaler?**

1. **Dynamic Scaling**: HPA automatically scales the number of pods up or down based on real-time demand, ensuring that applications can handle varying workloads without manual intervention.
2. **Cost Efficiency**: By scaling resources according to actual usage, HPA helps avoid overprovisioning, which can lead to unnecessary costs in cloud environments.
3. **Improved Performance**: Applications can maintain optimal performance levels during peak loads by increasing the number of replicas, thus distributing the workload effectively.
4. **High Availability**: HPA contributes to application availability by ensuring that sufficient resources are allocated to meet user demand at all times.

## **How to Configure Horizontal Pod Autoscaler**

### **Prerequisites**

* A running Kubernetes cluster (version 1.6 or later).
* Metrics Server installed in your cluster to provide resource metrics.

### **Step-by-Step Configuration**

#### **1. Create a Deployment**

First, create a deployment that you want to scale. Save the following YAML configuration as app-deployment.yaml:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: sample-app  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 app: sample-app  
 template:  
 metadata:  
 labels:  
 app: sample-app  
 spec:  
 containers:  
 - name: sample-app-container  
 image: nginx  
 resources:  
 limits:  
 cpu: "500m"  
 requests:  
 cpu: "200m"

Apply the deployment using:

kubectl apply -f app-deployment.yaml

#### **2. Create the Horizontal Pod Autoscaler**

Now, create the HPA for your deployment. You can do this using the kubectl autoscale command:

kubectl autoscale deployment sample-app --cpu-percent=50 --min=1 --max=10

In this command:

* --cpu-percent=50 sets the target average CPU utilization across all pods.
* --min=1 specifies the minimum number of pod replicas.
* --max=10 sets the maximum number of pod replicas.

#### **3. Verify HPA Configuration**

To check if the HPA has been created successfully and is functioning as expected, run:

kubectl get hpa

This will display information about your HPA, including current and desired replica counts based on CPU utilization.

#### **4. Monitor and Test**

You can simulate load on your application to see how HPA responds. For example, you could use a load testing tool to increase traffic to your service and observe how the number of pods scales up or down accordingly.

### **Conclusion**

Configuring Horizontal Pod Autoscaler in Kubernetes allows your applications to dynamically adjust their resource allocation based on real-time metrics, enhancing both performance and cost efficiency. By following these steps, you can ensure that your Kubernetes workloads are resilient and responsive to changing demands.

## **Automated Sidecar Container in Kubernetes**

An **automated sidecar container** in Kubernetes is a design pattern that allows developers to extend the functionality of the main application container within a pod. Sidecar containers run alongside the primary container and share the same lifecycle, network, and storage resources. This setup enables various enhancements such as logging, monitoring, data synchronization, and proxying without modifying the main application code.

### **Key Features of Sidecar Containers**

1. **Shared Lifecycle**: Sidecar containers start and stop with the main application container.
2. **Resource Sharing**: They share the same network namespace, allowing communication via localhost, and can access shared storage volumes.
3. **Independence**: Although they share resources, sidecars can be managed independently, enabling updates or maintenance without affecting the primary application.

### **Use Cases for Sidecar Containers**

* **Logging and Monitoring**: Collect logs from the main container and forward them to a centralized logging system.
* **Data Synchronization**: Sync data between the main application and external databases or storage systems.
* **Service Discovery**: Act as a proxy for routing requests to backend services.

### **Implementing Automated Sidecar Containers**

To implement automated sidecar containers in Kubernetes, you can use the built-in support introduced in version 1.28. Here’s a step-by-step example of how to set up a sidecar container for logging purposes using Fluentd.

#### **Example: Setting Up a Sidecar Container for Logging**

1. **Create a ConfigMap for Fluentd Configuration**: First, create a ConfigMap to hold your Fluentd configuration.

apiVersion: v1  
kind: ConfigMap  
metadata:  
 name: fluentd-config  
data:  
 fluent.conf: |  
 <source>  
 @type forward  
 port 24224  
 </source>  
 <match \*\*>  
 @type stdout  
 </match>

1. **Define Your Pod with Sidecar Container**: Create a pod definition that includes both your main application container and the Fluentd sidecar.

apiVersion: v1  
kind: Pod  
metadata:  
 name: my-app-pod  
spec:  
 containers:  
 - name: my-app-container  
 image: my-app-image  
 ports:  
 - containerPort: 8080  
 volumeMounts:  
 - name: log-volume  
 mountPath: /var/log/myapp  
 - name: fluentd-sidecar  
 image: fluent/fluentd:v1.12-1  
 env:  
 - name: FLUENTD\_CONF  
 value: "fluent.conf"  
 volumeMounts:  
 - name: log-volume  
 mountPath: /var/log/myapp  
 volumes:  
 - name: log-volume  
 emptyDir: {}

1. **Deploy the Pod**: Apply the configuration to your Kubernetes cluster.

kubectl apply -f fluentd-config.yaml  
kubectl apply -f my-app-pod.yaml

1. **Verify Functionality**: Check if both containers are running and that Fluentd is collecting logs from your application.

kubectl get pods my-app-pod  
kubectl logs my-app-pod -c fluentd-sidecar

### **Conclusion**

Automated sidecar containers in Kubernetes provide an effective way to enhance application functionality while maintaining clean separation of concerns. By leveraging Kubernetes' built-in support for sidecars, developers can easily implement additional features like logging or monitoring without altering their main application logic. This approach not only simplifies deployments but also improves maintainability and scalability of applications running in Kubernetes environments.