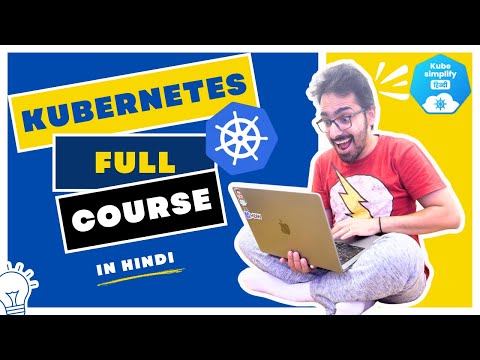
<https://www.youtube.com/watch?v=5NXmbV50IxE> -- Kubernetes Course in Hindi (13 Hours) | Full Hands-On Experience

[](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/)

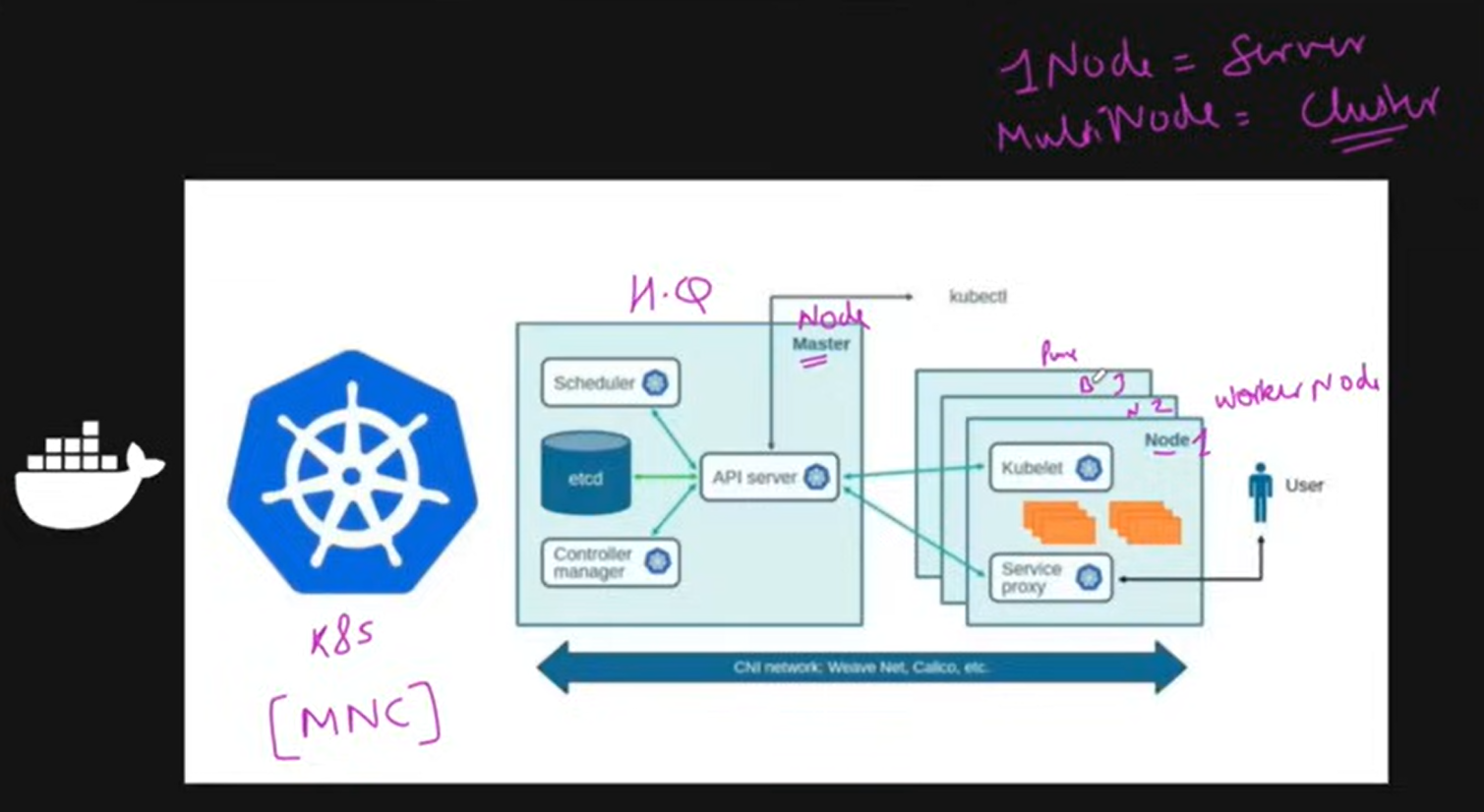
<https://www.udemy.com/course/aws-certified-devops-engineer-professional-hands-on/?couponCode=24T4MT120424>

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

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

<https://www.youtube.com/watch?v=W04brGNgxN4&t=1101s>

[](https://www.youtube.com/watch?v=W04brGNgxN4&t=1101s)



Kubernetes runs docker containers.

Master uses kubectl command to manage node and Worker uses kubelet command to manage pods.

Node means Server like Master or Worker. And combination of nodes is cluster.

API Server contains information of all activities happening on master and worker. It's like a communication gateway.

Scheduler service fulfills requirement of API Server. It helps to creates container in Worker node. To run the pod. In pod its run docker container.

Etcd – data stores in key value pair format. It stores status of all servers and pods.

Controller manager – check everything working correctly in master or worker or not. It checked expected state of pod.

Kubelet is manager at worker node. It provides information to API Server.

Server Proxy or Kube Proxy – To access pods from outside. Because of this user can access pods.

CNI Network example Calico, etc helps Pods, server to communicates with each other.

Kubectl [Kube Controller] -- From using this you can give instructions to all kubernetes pods and nodes.

Kubeadm – Setup installation of kubernetes.

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 continuously 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.

Here's a breakdown of the differences between the components kubeproxy, kubescheduler, calico, sidecars, kiali istio service and MTLS protocol:

* **kube-proxy**: A network proxy that runs on each node and manages network rules, facilitating communication between pods and external services. It is responsible for enabling network communication between Pods and external services.
* **kube-scheduler**: This component looks for Pods that are not yet bound to a node and assigns them to a suitable node.
* **Calico**: While not a core Kubernetes component, Calico is a networking and network security solution for containers. It can be integrated with Istio to provide a unified way to write security policies and implement restrictions without disturbing the entire system.
* **Sidecars**: Although not mentioned in the search results, a sidecar is a design pattern where a secondary container runs alongside a primary container within the same pod. It augments and enhances the main container, often handling tasks like logging, monitoring, or security.
* **Kiali**: A console for the Istio service mesh, used to visualize and manage the mesh. Kiali needs to retrieve Istio data and configurations, which are exposed through Prometheus, the Kubernetes API, and istiod. It queries the Kubernetes API to retrieve definitions for namespaces, services, deployments, pods, and other entities.
* **Istio**: A service mesh that integrates with Kubernetes to provide features like secure pod-to-pod communications, traffic management, and observability. Istio works by injecting a sidecar proxy (Envoy) onto each pod, which intercepts traffic. The service mesh is split into two main components: a data plane (Envoy proxies) and a control plane (Istiod).
* **Mutual TLS (mTLS)**: A security protocol enabled by default in Istio that provides mutual authentication between services. It ensures that both the client and server verify each other's identities before establishing a connection. Istio can configure mTLS to work in three modes: Permissive, Strict, and Disable. When mTLS is set to strict, workloads only accept mTLS traffic.

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.

In Kubernetes, the kind parameter is used to specify the type of Kubernetes resource that you are defining in your YAML configuration. Here are some of the most used values for the kind parameter:

### **Common kind Options in Kubernetes:**

1. **Pod**
   1. **Purpose:** Defines a single pod. A pod is the smallest deployable unit in Kubernetes that can contain one or more containers.
   2. Example: kind: Pod
2. **Deployment**
   1. **Purpose:** Manages a set of identical pods, ensuring that the desired number of replicas are running at all times. Supports rolling updates and rollback.
   2. Example: kind: Deployment
3. **StatefulSet**
   1. **Purpose:** Similar to a Deployment but provides stable network identities and persistent storage for each pod.
   2. Example: kind: StatefulSet
4. **DaemonSet**
   1. **Purpose:** Ensures that a copy of a pod is running on all (or selected) nodes in the cluster. Typically used for background services like logging agents or monitoring tools.
   2. Example: kind: DaemonSet
5. **ReplicaSet**
   1. **Purpose:** Ensures that a specified number of pod replicas are running. It is an advanced version of the Deployment but not usually used directly. Deployments use ReplicaSets internally.
   2. Example: kind: ReplicaSet
6. **Job**
   1. **Purpose:** A batch job that runs to completion. Used for tasks that are intended to run once and finish, such as database migrations or backups.
   2. Example: kind: Job
7. **CronJob**
   1. **Purpose:** Defines a cron job that runs periodically on a schedule. Useful for running jobs on a fixed time interval (e.g., backups or scheduled reports).
   2. Example: kind: CronJob
8. **Service**
   1. **Purpose:** Defines a logical set of pods and a policy to access them, typically by exposing an application to the network.
   2. Example: kind: Service
9. **ConfigMap**
   1. **Purpose:** Provides configuration data in key-value pairs that can be injected into pods as environment variables or volumes.
   2. Example: kind: ConfigMap
10. **Secret**
    1. **Purpose:** Stores sensitive data, such as passwords or tokens, in an encrypted form.
    2. Example: kind: Secret
11. **Namespace**
    1. **Purpose:** A way to organize and isolate Kubernetes resources within a cluster. Namespaces provide scope for names.
    2. Example: kind: Namespace
12. **Ingress**
    1. **Purpose:** Manages HTTP and HTTPS routing rules to services within the cluster. Typically used for exposing services to external traffic.
    2. Example: kind: Ingress
13. **PersistentVolume (PV)**
    1. **Purpose:** Represents a piece of storage in the cluster, provisioned by an administrator or dynamically via a storage class.
    2. Example: kind: PersistentVolume
14. **PersistentVolumeClaim (PVC)**
    1. **Purpose:** A request for storage by a user that can be satisfied by a PersistentVolume. This is used in conjunction with StatefulSets or Pods that require persistent storage.
    2. Example: kind: PersistentVolumeClaim
15. **NetworkPolicy**
    1. **Purpose:** Defines rules for controlling the communication between pods or services, allowing you to define what network traffic is allowed or blocked.
    2. Example: kind: NetworkPolicy
16. **HorizontalPodAutoscaler (HPA)**
    1. **Purpose:** Automatically scales the number of pods in a deployment, replica set, or stateful set based on CPU utilization or custom metrics.
    2. Example: kind: HorizontalPodAutoscaler
17. **PodDisruptionBudget (PDB)**
    1. **Purpose:** Ensures that a certain number of pods remain available during voluntary disruptions (e.g., rolling updates or draining nodes).
    2. Example: kind: PodDisruptionBudget
18. **Role**
    1. **Purpose:** Defines a set of permissions within a specific namespace, usually used in combination with RoleBindings.
    2. Example: kind: Role
19. **RoleBinding**
    1. **Purpose:** Grants the permissions defined in a Role or ClusterRole to a user or group of users (or service accounts).
    2. Example: kind: RoleBinding
20. **ClusterRole**
    1. **Purpose:** Similar to a Role, but it grants permissions across the entire cluster, not just a single namespace.
    2. Example: kind: ClusterRole
21. **ClusterRoleBinding**
    1. **Purpose:** Grants the permissions defined in a ClusterRole to users, groups, or service accounts across the entire cluster.
    2. Example: kind: ClusterRoleBinding
22. **Event**
    1. **Purpose:** Represents an event in Kubernetes, typically used for logging or debugging.
    2. Example: kind: Event
23. **LimitRange**
    1. **Purpose:** Defines the resource usage limits (e.g., memory, CPU) for a namespace.
    2. Example: kind: LimitRange
24. **ResourceQuota**
    1. **Purpose:** Enforces limits on the total resources (e.g., CPU, memory, storage) used within a namespace.
    2. Example: kind: ResourceQuota
25. **StorageClass**
    1. **Purpose:** Defines the storage options for dynamic provisioning of persistent volumes.
    2. Example: kind: StorageClass

### **Summary**

The kind parameter helps Kubernetes understand the type of resource you're defining, and there are many different kinds depending on the use case, from managing pods and services to configuring access control, storage, and resource limits.

Let me know if you need further clarification or examples of any specific kind!

**NOTE:** In Kubernetes, you can define multiple resources in a single YAML file. To do this, simply separate each resource with ---. Each resource will have its own kind field, specifying the type of resource being defined. This allows you to group related resources together in one file, making it easier to manage and deploy multiple components at once.

We can use “dependsOn” parameter for specifying the dependency on another pod.

Namespace is like a group. Default namespace is “kube-system”

Purpose of Template Label and Selector in yaml configuration of kubernetes?

Template Labels

Template labels are defined within the template section of a Kubernetes resource YAML file, specifically for objects like Deployments or StatefulSets. These labels serve several important functions:

1. Identification: Template labels help identify the pods created by the deployment. Each pod inherits the labels specified in the template, which allows for easy identification and organization of resources within the cluster.
2. Grouping: By assigning specific labels to pods, you can group them based on characteristics such as application name, environment (e.g., production, staging), or version. This grouping facilitates management tasks like monitoring and scaling.
3. Consistency: When you define template labels, it ensures that all pods created from that template share the same labels, maintaining consistency across your application instances.

Selectors

Selectors are used to define how Kubernetes identifies which pods belong to a particular deployment or service. They match the labels assigned to pods against specified criteria.

1. Pod Management: The selector defines which pods are managed by a deployment or service based on their labels. This is critical for ensuring that operations like scaling, updating, or deleting pods only affect the intended resources.
2. Dynamic Selection: Selectors allow for dynamic selection of pods based on their current state and labels, enabling Kubernetes to manage resources efficiently as conditions change.
3. Match Criteria: Selectors can use matchLabels or matchExpressions to specify criteria for selecting pods. This flexibility allows for more complex queries based on multiple label values or conditions.

How to check etcd, controller manager, scheduler, api gateway status and details in kubernetes?

Answer:

kubectl get pods –n kube-system | grep controller-manager

kubectl get pods –n kube-system | grep scheduler

kubectl get pods –n <api-gateway-namespace>

etcdctl endpoint status –write-out=table

CoreDNS --- Why CoreDNS is not involved in k8s architectural diagram?

Difference between Static Pods and Standalone Pods?

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 configure it with current Kubernetes master and slaves.

To deploy Envoy as a load balancer in a Kubernetes environment, follow these structured steps. This guide will cover the necessary configurations to set up Envoy with your Kubernetes master and worker nodes.

## Prerequisites

* A running Kubernetes cluster with access to kubectl.
* Basic understanding of Kubernetes concepts (pods, services, deployments).
* Docker installed for building the Envoy image.

## Steps to Deploy Envoy Load Balancer

## 1. Create a Headless Service

A headless service allows DNS queries to return the IP addresses of all pods backing the service, which is useful for Envoy's service discovery.

text

apiVersion: v1  
kind: Service  
metadata:  
 name: myapp-headless  
spec:  
 clusterIP: None # This makes it a headless service  
 selector:  
 app: myapp  
 ports:  
 - port: 80

## 2. Create the Envoy Configuration File

Create an envoy.yaml file that defines how Envoy should route traffic. Below is a basic example:

text

static\_resources:  
 listeners:  
 - name: listener\_0  
 address:  
 socket\_address: { address: 0.0.0.0, port\_value: 80 }  
 filter\_chains:  
 - filters:  
 - name: envoy.filters.network.http\_connection\_manager  
 config:  
 codec\_type: AUTO  
 stat\_prefix: ingress\_http  
 route\_config:  
 name: local\_route  
 virtual\_hosts:  
 - name: backend  
 domains: ["\*"]  
 routes:  
 - match: { prefix: "/" }  
 route:  
 cluster: myapp\_cluster  
 http\_filters:  
 - name: envoy.filters.http.router  
  
 clusters:  
 - name: myapp\_cluster  
 connect\_timeout: 0.25s  
 type: STRICT\_DNS  
 lb\_policy: ROUND\_ROBIN  
 load\_assignment:  
 cluster\_name: myapp\_cluster  
 endpoints:  
 - lb\_endpoints:  
 - endpoint:  
 address:  
 socket\_address: { address: myapp-headless, port\_value: 80 }

## 3. Build and Push the Envoy Docker Image

Use Docker to build the image containing your Envoy configuration.

bash

*# Create a Dockerfile for Envoy*  
echo "FROM envoyproxy/envoy:latest" > Dockerfile  
echo "COPY envoy.yaml /etc/envoy.yaml" >> Dockerfile  
echo 'CMD /usr/local/bin/envoy -c /etc/envoy.yaml' >> Dockerfile  
  
*# Build and push the image to your Docker registry*  
docker build -t your-docker-repo/myapp-envoy .  
docker push your-docker-repo/myapp-envoy

## 4. Deploy Envoy in Kubernetes

Create a deployment for Envoy using the Docker image you just built.

text

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: envoy-deployment  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 app: envoy  
 template:  
 metadata:  
 labels:  
 app: envoy  
 spec:  
 containers:  
 - name: envoy-container  
 image: your-docker-repo/myapp-envoy  
 ports:  
 - containerPort: 80

## 5. Expose Envoy as a LoadBalancer Service (Optional)

If you want to expose Envoy externally, create a LoadBalancer service.

text

apiVersion: v1  
kind: Service  
metadata:  
 name: envoy-loadbalancer  
spec:  
 type: LoadBalancer  
 ports:  
 - port: 80   
 targetPort: 80   
 selector:  
 app: envoy

## 6. Apply the Configurations to Kubernetes

Run the following commands to apply all configurations:

bash

kubectl apply -f headless-service.yaml *# Create headless service for your application.*  
kubectl apply -f envoy.yaml *# Deploy Envoy.*  
kubectl apply -f envoy-deployment.yaml *# Create deployment for Envoy.*  
kubectl apply -f envoy-loadbalancer.yaml *# Expose Envoy as a LoadBalancer (if needed).*

## Conclusion

After following these steps, you should have Envoy deployed as a load balancer in front of your application services in Kubernetes. You can further customize the envoy.yaml configuration to implement advanced features like rate limiting and circuit breaking as needed.

Difference in Work of MetaLB and Envoy in Kubernetes?

MetaLB and Envoy serve different purposes in a Kubernetes environment, each addressing specific needs related to load balancing and service communication.

## MetaLB

Purpose: MetaLB is primarily focused on providing LoadBalancer services in bare-metal Kubernetes clusters. By default, Kubernetes does not have a built-in way to manage LoadBalancer services in environments without cloud provider integrations. MetaLB fills this gap by enabling the creation of external IPs that can route traffic to services within the cluster.Key Features:

* Layer 2 and Layer 3 Load Balancing: It supports both L2 (ARP) and L3 (BGP) modes for distributing traffic.
* Integration with Kubernetes Services: It allows users to expose their services using the LoadBalancer type, which is essential for external access.
* Bare-Metal Focus: Specifically designed for on-premise deployments where cloud-native load balancing isn't available.

## Envoy

Purpose: Envoy is a high-performance edge and service proxy designed for microservices architectures. It excels in managing service-to-service communication, offering advanced features like traffic management, observability, and security.Key Features:

* Dynamic Service Discovery: Envoy can automatically discover services in a microservices architecture.
* Advanced Traffic Management: It provides capabilities such as load balancing, circuit breaking, retries, and rate limiting.
* Observability: Envoy includes extensive logging and tracing features to monitor application performance and health.
* Security Features: Built-in support for TLS and mTLS enhances the security of inter-service communications.

## Comparison Summary

|  |  |  |
| --- | --- | --- |
| **Feature** | **MetaLB** | **Envoy** |
| Primary Function | Load balancing for bare-metal (i.e. on-premise) | Service proxy for microservices |
| Layer Support | L2/L3 Load Balancing | Layer 7 (HTTP) load balancing |
| Use Case | Exposing services externally | Managing internal service communication |
| Traffic Management | Basic routing | Advanced traffic control |
| Observability | Limited | Extensive monitoring capabilities |
| Security Features | Basic support | Advanced security features |

In summary, while MetaLB provides essential load balancing capabilities for exposing services in bare-metal Kubernetes clusters, Envoy offers a robust solution for managing complex service interactions within a microservices architecture. They can be used together effectively; for instance, MetaLB can expose an Envoy service that handles internal traffic management and observability.

Kubernetes Layer of Operation and Layer Mapping with OSI Model

Kubernetes operates across multiple layers, which can be mapped to the OSI (Open Systems Interconnection) model. Understanding this mapping helps clarify how Kubernetes manages networking and communication within its architecture.

## Kubernetes Layers of Operation

Kubernetes can be conceptualized in several layers, including:

1. Control Plane: This is responsible for managing the Kubernetes cluster. It includes components like the API server, scheduler, and controller manager, which handle the overall state and configuration of the cluster.
2. Data Plane: This layer consists of worker nodes where application workloads (packaged as pods) run. It includes components such as kubelet and kube-proxy that manage pod operations and networking.
3. Application Layer: This layer deals with the actual applications running on Kubernetes, including deployment configurations and service definitions.
4. Networking Layer: This layer facilitates communication between pods and services, utilizing tools like CNI (Container Network Interface) for network connectivity.
5. Storage Layer: This layer manages data storage solutions within Kubernetes, including persistent volumes and claims.
6. Security Layer: This encompasses security features such as Role-Based Access Control (RBAC) and TLS certificates to secure communication and access.

## Mapping Kubernetes to the OSI Model

The OSI model consists of seven layers, each serving a specific role in network communication. Here’s how Kubernetes concepts align with these layers:

|  |  |  |
| --- | --- | --- |
| OSI Layer | Description | Kubernetes Mapping |
| Layer 1 | Physical Layer: Hardware infrastructure | Physical servers or cloud infrastructure hosting Kubernetes clusters |
| Layer 2 | Data Link Layer: Node-to-node data transfer | Network interfaces and virtualized networking in cloud environments |
| Layer 3 | Network Layer: Routing data between networks | Pod IP assignment, Services for routing traffic, kube-proxy for traffic management |
| Layer 4 | Transport Layer: End-to-end communication | TCP/UDP protocols used for pod communication, LoadBalancer services for distributing traffic |
| Layer 5 | Session Layer: Managing connections | Ingress controllers managing external access to services |
| Layer 6 | Presentation Layer: Data formatting | Not directly mapped; however, application-level protocols may apply |
| Layer 7 | Application Layer: End-user interface | Application workloads running in pods, Service definitions for accessing applications |

## Detailed Mapping

* Layers 1 & 2 (Physical and Data Link): These layers encompass the physical servers or cloud infrastructure where Kubernetes is deployed. In a cloud environment, this is often virtualized hardware.
* Layer 3 (Network): In this layer, each pod receives a unique IP address, allowing them to communicate effectively. Services abstract a set of pods and facilitate routing requests similarly to how DNS operates.
* Layer 4 (Transport): Kubernetes supports both TCP and UDP protocols for communication between pods. LoadBalancer services distribute incoming traffic across multiple pods to ensure high availability.
* Layers 5, 6 & 7 (Session, Presentation, Application): The Ingress resource manages external access to services, providing features like load balancing and SSL termination at the application layer.

## Conclusion

Mapping Kubernetes operations to the OSI model provides clarity on how various components interact within a cluster. Each layer serves distinct functions that contribute to the overall efficiency and effectiveness of application deployment and management in a cloud-native environment. Understanding these mappings can aid developers and operators in troubleshooting network issues and optimizing their Kubernetes deployments.

Purpose of using logstash in kubernetes ELK environment and it works on which layer of OSI Model and how to filter log details. And what is XPack Monitoring and Management. and how to configure it.

Logstash is a crucial component of the ELK (Elasticsearch, Logstash, Kibana) stack, particularly in a Kubernetes environment. Here’s an overview of its purpose, its operation within the OSI model, how to filter log details, and an explanation of X-Pack Monitoring and Management along with configuration steps.

## Purpose of Using Logstash in Kubernetes ELK Environment

1. Log Aggregation: Logstash collects logs from various sources (e.g., application logs, system logs) and aggregates them into a centralized location for easier management and analysis.
2. Data Transformation: It processes incoming log data through various filters, allowing for parsing, enrichment, and formatting, which transforms raw logs into structured data suitable for analysis.
3. Forwarding Logs: After processing, Logstash sends the transformed logs to Elasticsearch for indexing. This enables users to search and visualize their logs using Kibana.
4. Integration with Kubernetes: Deploying Logstash on Kubernetes allows for dynamic scaling and resource management. Kubernetes can automatically adjust the number of Logstash instances based on log volume and resource usage, ensuring efficient operation.

## OSI Model Layer Mapping

Logstash primarily operates at the Transport Layer (Layer 4) of the OSI model, managing the transport of log data over protocols such as TCP and UDP. It also interacts with the Application Layer (Layer 7) since it processes application-level data (logs) before sending it to Elasticsearch.

## Filtering Log Details

To filter log details in Logstash, you can utilize filter plugins in the Logstash configuration file. Here’s a basic example using the Grok filter:

text

input {  
 beats {  
 port => 5044  
 }  
}  
  
filter {  
 grok {  
 match => { "message" => "%{COMBINEDAPACHELOG}" }  
 }  
  
 # Example filtering: drop 404 responses  
 if [response] == "404" {  
 drop { }  
 }  
}  
  
output {  
 elasticsearch {  
 hosts => ["<http://elasticsearch:9200>"]  
 index => "logs-%{+YYYY.MM.dd}"  
 }  
}

## Explanation:

* Input Section: Configures Logstash to receive logs from Filebeat or other sources.
* Filter Section:
  + The grok filter parses incoming log messages.
  + The conditional statement checks for specific criteria (e.g., response code "404") and drops those entries.
* Output Section: Specifies where to send processed logs—in this case, to Elasticsearch.

## X-Pack Monitoring and Management

X-Pack Monitoring is an extension of the ELK stack that provides monitoring capabilities for Elasticsearch, Logstash, and Kibana. It enables users to track performance metrics, visualize cluster health, and receive alerts based on predefined conditions.

## Key Features:

* Cluster Health Monitoring: Provides insights into cluster performance and resource utilization.
* Alerting: Allows users to set up alerts based on specific metrics or thresholds.
* Management UI: Offers a user-friendly interface to manage settings and view metrics.

## Configuration Steps for X-Pack Monitoring:

1. Install X-Pack: Ensure that X-Pack is installed in your ELK stack components. For Elasticsearch, it comes pre-installed in the default distributions.
2. Enable Monitoring:
   * In elasticsearch.yml, add the following configuration:

text

xpack.monitoring.enabled: true  
xpack.monitoring.elasticsearch.hosts: ["<http://localhost:9200>"]

* + In logstash.yml, include:

text

xpack.monitoring.enabled: true  
xpack.monitoring.elasticsearch.hosts: ["<http://elasticsearch:9200>"]

* + For Kibana, update kibana.yml:

text

xpack.monitoring.enabled: true  
xpack.monitoring.ui.container.elasticsearch.enabled: true

1. Deploy with Helm (Optional): If using Helm to deploy your ELK stack on Kubernetes, you can enable monitoring during installation by setting appropriate values in your Helm chart configuration.
2. Access Monitoring UI: Once configured, you can access monitoring dashboards in Kibana under the "Monitoring" section to view cluster health and performance metrics.

By integrating Logstash into your Kubernetes ELK environment and enabling X-Pack Monitoring, you can effectively manage log data while gaining valuable insights into your system's performance.

## **What is Istio Service Mesh in Kubernetes?**

Istio is an open-source **service mesh** that provides a transparent layer for managing microservices communication within a distributed architecture, particularly in Kubernetes environments. It enables developers to secure, connect, and monitor microservices without requiring significant changes to the application code. Key features of Istio include:

* **Traffic Management**: Automatic load balancing, traffic routing, and fine-grained control over service interactions.
* **Security**: Mutual TLS encryption for secure service-to-service communication and robust identity-based authentication.
* **Observability**: Automatic metrics collection, logging, and tracing for monitoring service behavior.

Istio operates through a **control plane** (which manages the configuration and policies) and a **data plane** (which handles the actual traffic between services using Envoy proxies). **Istio provides details like the number of users currently accessing a pod.**

## **Steps to Integrate Istio into a Kubernetes Cluster**

Integrating Istio into an existing Kubernetes cluster involves several steps. Below is a streamlined guide to help you set up Istio:

### **Step 1: Install Istio**

1. **Download Istio**:
   1. Download the latest Istio release from the official website.
   2. Add the istioctl command-line tool to your system's PATH.
2. **Install Istio in your Kubernetes cluster**:
   1. Use the following command to install Istio with the demo profile:istioctl install --set profile=demo -y  
       This command installs the core components of Istio, including Istiod, ingress, and egress gateways.

### **Step 2: Prepare Namespace for Istio**

* Create a namespace for your application and enable automatic sidecar injection:kubectl create namespace my-app  
  kubectl label namespace my-app istio-injection=enabled  
   This ensures that Envoy proxies are automatically injected into your application pods [4][10].

### **Step 3: Deploy a Demo Application**

* Deploy a sample application to test your Istio installation. You can use a simple application like Bookinfo:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
name: bookinfo  
namespace: my-app  
spec:  
replicas: 3  
selector:  
 matchLabels:  
 app: bookinfo  
template:  
 metadata:  
 labels:  
 app: bookinfo  
 spec:  
 containers:  
 - name: bookinfo  
 image: docker.io/istio/examples-bookinfo-productpage-v1:latest  
 Apply this manifest using:kubectl apply -f bookinfo.yaml

### **Step 4: Set Up Ingress Gateway**

* Create an ingress gateway to expose your application externally. Define a Gateway and VirtualService like so: ```yaml apiVersion: networking.istio.io/v1alpha3 kind: Gateway metadata: name: bookinfo-gateway namespace: my-app spec: selector: istio: ingressgateway # use istio default gateway servers:
  + port: number: 80 name: http protocol: HTTP hosts:
    - "\*"

apiVersion: networking.istio.io/v1alpha3 kind: VirtualService metadata: name: bookinfo namespace: my-app spec: hosts:

* "\*" gateways:
* bookinfo-gateway http:
* match:
  + uri: prefix: /productpage route:
  + destination: host: productpage.default.svc.cluster.local # adjust as necessary port: number: 9080 ``` Apply these configurations using kubectl apply.

### **Step 5: Visualize Traffic with Kiali Dashboard**

* To visualize the service graph and monitor traffic, install Kiali by following its installation guide. Once installed, you can access Kiali via the ingress gateway.

This setup enables you to leverage Istio's capabilities for traffic management, security, and observability in your Kubernetes cluster effectively.

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.

To enable **autoscaling** for a Matrix server in Kubernetes, you can use Kubernetes' built-in scaling mechanisms like the **Horizontal Pod Autoscaler (HPA)** and **Cluster Autoscaler**, as well as advanced tools like the **Vertical Pod Autoscaler (VPA)** or event-driven autoscaling solutions. Here's a detailed guide:

## **1. Why Autoscaling for a Matrix Server?**

Matrix servers (e.g., Synapse) often experience fluctuating workloads due to:

* Increased user activity during peak hours.
* Federation with other servers, which can add unpredictable traffic.
* Resource-intensive tasks like message history synchronization and encryption.

Autoscaling ensures:

* Optimal resource utilization.
* Cost efficiency by scaling down during low usage.
* High availability and responsiveness during traffic spikes.

## **2. Types of Autoscaling in Kubernetes**

## **Horizontal Pod Autoscaler (HPA)**

* Scales the number of pods based on resource usage (e.g., CPU, memory).
* Ideal for handling increased traffic by adding more replicas of the Matrix server.
* Requires the **Kubernetes Metrics Server** to collect resource metrics.

Example HPA YAML configuration:

text

apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
 name: matrix-synapse-hpa  
 namespace: matrix  
spec:  
 scaleTargetRef:  
 apiVersion: apps/v1  
 kind: Deployment  
 name: synapse-deployment  
 **minReplicas: 2**  
 **maxReplicas: 10**  
 metrics:  
 - type: Resource  
 resource:  
 name: cpu  
 targetAverageUtilization: 70

## **Vertical Pod Autoscaler (VPA)**

* Adjusts the resource requests and limits of pods based on observed usage.
* Useful for optimizing pod performance without manual intervention.
* Requires installation of the VPA tool as it is not part of Kubernetes by default.

## **Cluster Autoscaler**

* Scales the number of nodes in a cluster based on pending pods that cannot be scheduled due to insufficient resources.
* Works in conjunction with HPA or VPA to ensure sufficient infrastructure is available.

## **Event-Driven Autoscaling**

* Tools like **KEDA** (Kubernetes Event-driven Autoscaler) can scale workloads based on custom events, such as queue length or API requests.

## **3. Steps to Set Up Autoscaling for a Matrix Server**

## **Step 1: Deploy Kubernetes Metrics Server**

The Metrics Server collects CPU and memory metrics required for HPA or VPA. Install it using Helm or YAML manifests:

bash

kubectl apply -f <https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml>

## **Step 2: Configure Horizontal Pod Autoscaler**

Create an HPA resource targeting your Synapse deployment. Ensure your deployment has resource requests/limits defined:

text

resources:  
 requests:  
 memory: "512Mi"  
 cpu: "500m"  
 limits:  
 memory: "1Gi"  
 cpu: "1000m"

Apply the HPA manifest to scale pods automatically based on CPU or memory usage.

## **Step 3: Enable Cluster Autoscaler**

Install and configure the Cluster Autoscaler to dynamically add/remove nodes when pods cannot be scheduled due to insufficient resources.

Example setup for managed Kubernetes services like AKS, GKE, or EKS:

bash

kubectl apply -f cluster-autoscaler.yaml

## **Step 4: Monitor and Tune**

Use monitoring tools like Prometheus, Grafana, or Datadog to observe autoscaling behavior and fine-tune thresholds.

## **4. Advanced Considerations**

1. **Federation Traffic**:
   1. Federation can cause traffic spikes. Use custom metrics (e.g., HTTP request rates) with tools like Datadog to trigger scaling.
2. **Database Scaling**:
   1. Matrix servers rely heavily on databases like PostgreSQL. Ensure that your database is also scalable (e.g., using managed database services with autoscaling).
3. **Stateful Components**:
   1. If scaling stateful components (e.g., media storage), consider using StatefulSets and ensure proper data persistence with PersistentVolumeClaims (PVCs).
4. **Event-Based Scaling**:
   1. Use KEDA if you need to scale based on external events, such as message queue length or API request counts.

## **Example Architecture**

|  |  |
| --- | --- |
| **Component** | **Scaling Strategy** |
| Synapse Pods | Horizontal Pod Autoscaler |
| Database (Postgres) | Managed DB with autoscaling |
| Node Pools | Cluster Autoscaler |
| Media Store | StatefulSet with PVCs |

By combining these strategies, you can ensure that your Matrix server deployment is resilient, cost-efficient, and capable of handling dynamic workloads effectively.

## **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.

Virtual Server management by using KubeVirt ----

Minio Kubernetes -- Object Storage in your Kubernetes clsuter with MiniIO. Ceph kubernetes is an alternative.

Velero kubernetes -- For Kubernetes backup.

Trino -- Trino is an open-source distributed SQL query engine for adhoc and batch ETL queries against multiple types of data sources.

Linkerd provides a layer of abstraction to help control these communications, giving developers better visibility, improving network efficiency and enabling more robust security.

Experience with serverless architectures (Lambda, Cloud Functions, etc.).

Knowledge of service mesh technologies (Istio, Linkerd, Consul).

Experience with policy as code (OPA, Kyverno, HashiCorp Sentinel).

Familiarity with database scaling strategies for PostgreSQL, MySQL, or NoSQL solutions

Infrastructure as Code (IaC): Design and maintain cloud infrastructure using Terraform, CloudFormation, or Pulumi.

CI/CD Pipelines: Build and manage deployment pipelines using GitHub Actions, GitLab CI, Jenkins, or ArgoCD.

Cloud Management: Deploy and maintain cloud services on AWS / GCP / Azure with a focus on security, scalability, and cost optimization.

Containerization & Orchestration: Develop and manage OpenShift stacks, Kubernetes clusters, Docker containers, and Helm charts.

Monitoring & Observability: Implement monitoring and alerting solutions using Prometheus, Grafana, Datadog, New Relic, or similar.

Automation & Scripting: Automate infrastructure provisioning, deployments, and operational tasks using Bash, Python, or Go.

Security & Compliance: Implement security best practices, including IAM policies, network security, and compliance automation.

Incident Management & Troubleshooting: Respond to and mitigate production incidents, conducting post-mortems and improving system reliability.