Kubernetes

Kubernetes Introduction

What is GitOps?

GitOps is a way to manage and deploy software using Git as the single source of truth. Think of it like this: instead of manually configuring servers or applications, you store all the configuration files in a Git repository.

When you want to make changes (like deploying a new version of your app), you simply update the files in Git. Then, automated tools watch for those changes and apply them to your infrastructure automatically. This makes it easier to track changes, roll back if something goes wrong, and keep everything organized. It's like using Git for your code, but for your entire system setup! ArgoCD is a GitOps Tool.

Kubernetes, also known as K8s, is an open-source platform for managing containerized workloads and services. It provides a way to deploy, scale, and manage containerized applications across a cluster of nodes.

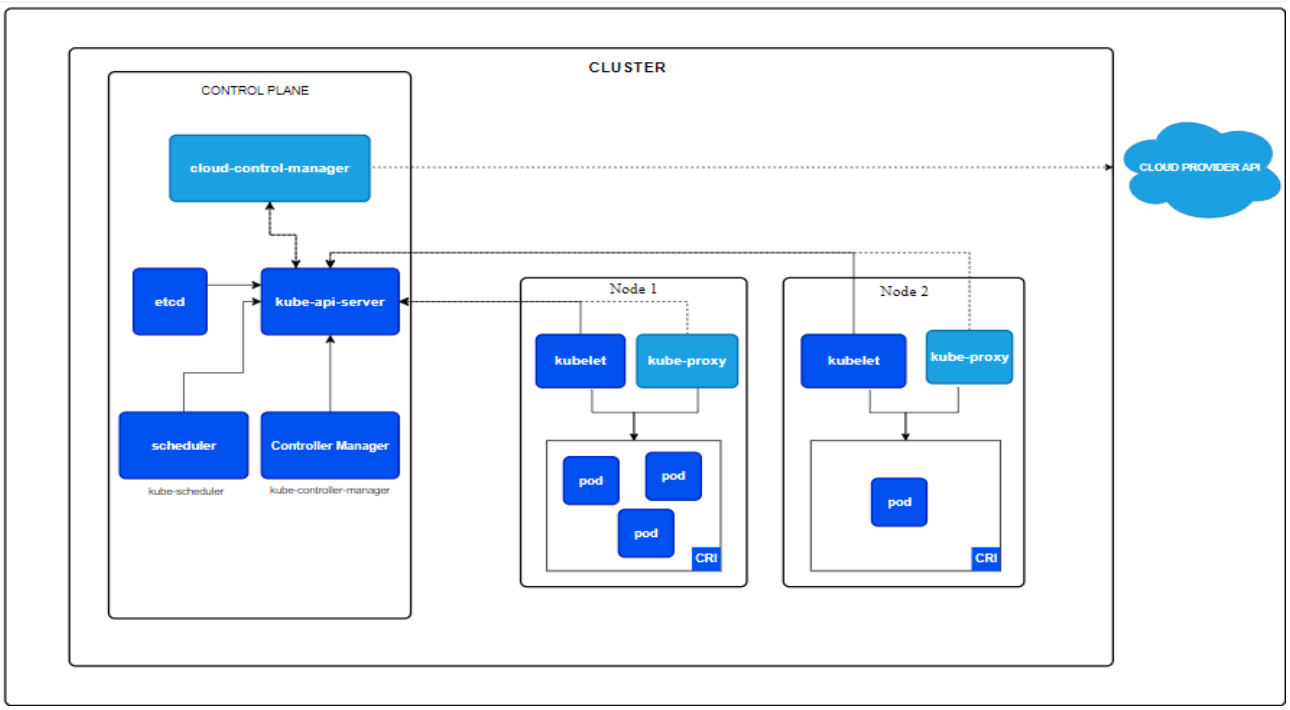
K8s consists of master and worker nodes, where master node is responsible for managing overall state of cluster, while worker node run the actual workloads.

Kubernetes Cluster:

Two ways we can Create Kubernetes Cluster:

1. Using kubeadm (On-Premises)
2. Using Minikube (Local Development)
3. Using Managed Kubernetes Services
    a. Amazon EKS (Elastic Kubernetes Service)
    b. Google GKE (Google Kubernetes Engine)
    c. Azure AKS (Azure Kubernetes Service)

Kubernetes Architecture:



COMPONENTS OF MASTER NODE: (Etcd, Kub-scheduler, Kube-controller, Kube api server)

ETCD: Is just like a database stores data regarding the cluster such as nodes, pods, configs, secrets, other. Kubectl get control command is from etcd server.

KUBE-SCHEDULER: Scheduler responsible for assigning the newly created pods to nodes based on container requirements.

KUBE-CONTROLLER: watches the state of nodes in the cluster and takes actions to ensure that nodes are stable and healthy. For example, if a node fails, the node controller will take actions to ensure that the workloads running on the failed node are rescheduled onto other nodes in the cluster.

KUBE-APISERVER: is used to handle request and response b/w master and worker nodes.

COMPONENTS OF WORKER NODE:

KUBELET: The kubelet communicates with the kube- apiserver to receive instructions on which pods to run on the node and reports back to the master node with updates on the status of the containers and their health.

KUBEPROXY: is responsible for managing the networking and routing configurations for services within the cluster. When a service is established, Kubernetes generates a set of iptables rules on each node within the cluster

Communication Flow:

User Interaction: Users interact with the cluster using the Kubernetes API (usually via kubectl).

Kube-API Server: Receives requests and communicates with etcd to store or retrieve state information.

Scheduler: Kube-API communicate with kube-schedular and decides where to run new pods based on available resources.

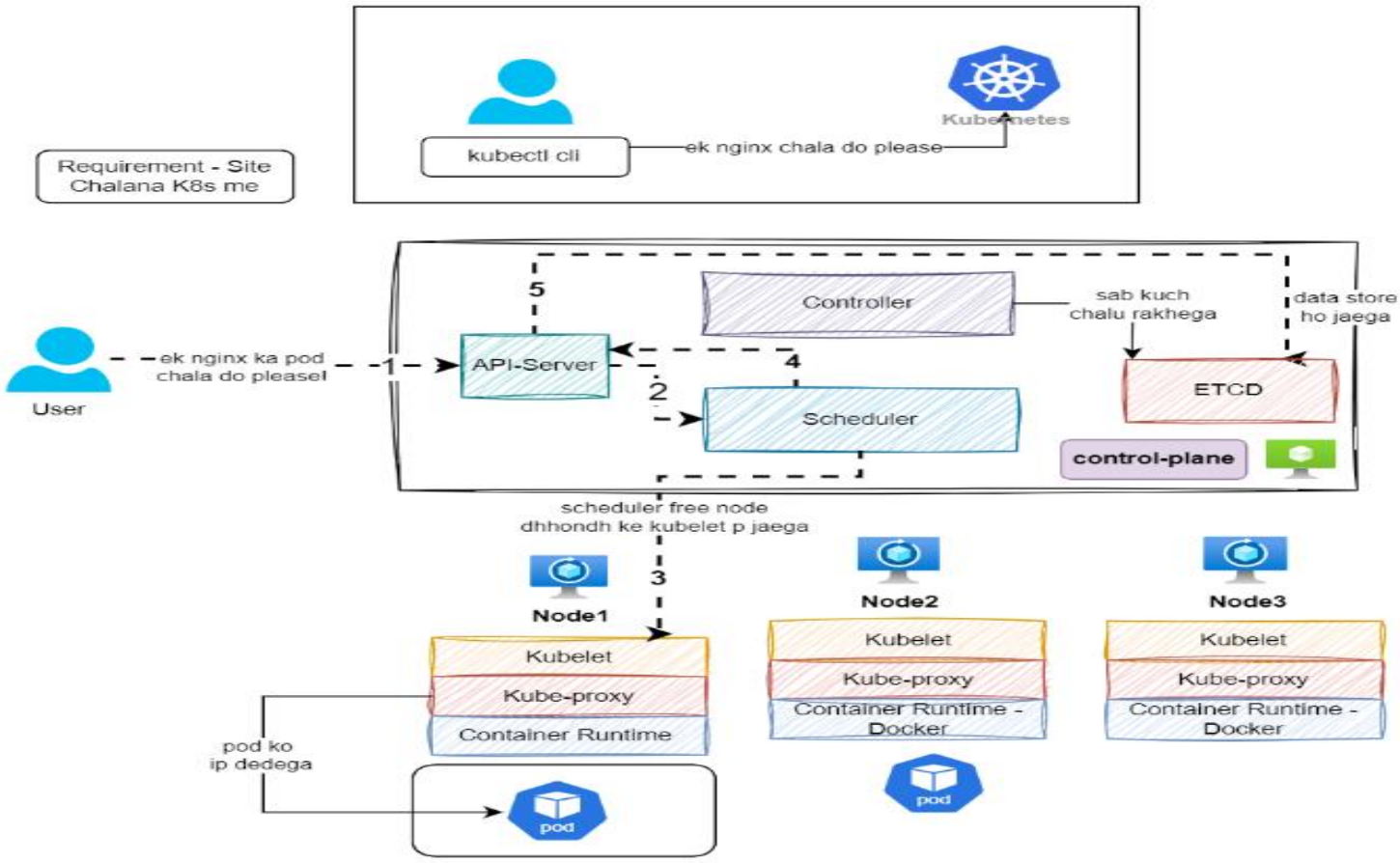
Controllers: Monitor the state of the cluster and make adjustments as needed (like creating or deleting pods).

Kubelet: On worker nodes, it pulls the desired state and ensures that the actual state matches the desired state by managing the containers.

Kube-Proxy: Manages network communication, allowing pods to connect and communicate with each other.Kube-proxy provides IP to pod.

Pods: Smallest deployable unit in k8s. It is a group of one or more container running instances of an application. Ephemeral in nature (means if pods fail k8s automatically create replica of that pod)

Namespace: In Kubernetes, a namespace provides isolation environments within a single cluster. This isolation allows you to separate resources, such as pods and services, so that they do not interfere with each other. It enables multiple users or applications to coexist in the same cluster without conflict, making it easier to manage resources and apply access controls.



Kubernetes Networking:

Types of services:

Cluster-ip : Makes the service accessible only inside the cluster

Nodeport: Exposes the service on the static port on each node in the cluster, which can be accessed from outside the cluster using the node ip address.

Loadbalancer: Automatically creates a load balancer (e.g., in the cloud) to expose the service to the internet.

Headless: A headless service in Kubernetes is a service that does not have a cluster IP address. This means it doesn't route traffic through a single IP but instead allows direct access to the individual pods backing the service.

Key Features:

- **No Load Balancer:** Unlike regular services, a headless service doesn't load balance requests. Instead, it allows clients to connect directly to the pods.
- **DNS Records:** When you create a headless service, Kubernetes creates DNS A records for each pod. This means that when you query the service's DNS name, you get the IP addresses of all the associated pods instead of a single IP.

Use Cases:

- Stateful Applications: Useful for databases and other stateful applications where you need to connect to specific instances.
- Service Discovery: Allows other applications to discover and connect to pods directly, useful in microservices architectures.

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

When you query the DNS for my-headless-service, you'll receive the IP addresses of all pods with the label app: my-app. This direct access can be helpful for certain application architectures that require pod-level communication.

ExternalName: Maps a service to an external DNS name, redirecting traffic outside the cluster.

Ingress: An Ingress is a Kubernetes resource that defines a set of rules for routing external HTTP(S) traffic to a service. Ingress resources require an Ingress controller to be deployed in the cluster, which is responsible for implementing the routing rules. Ingress controllers are available for many popular web servers, such as Nginx

Network Policies: **Network Policies** in Kubernetes are a way to control the traffic flow to and from pods. They help enhance security by defining rules that specify which pods can communicate with each other and how they can do so. Here's a simple overview:

Key Features of Network Policies:

1. **Traffic Control**:
   - Network policies allow you to specify which pods can send and receive traffic. You can restrict access based on pod labels, namespaces, and IP addresses.
2. **Isolation**:
   - By default, all pods can communicate with each other. With network policies, you can isolate pods, allowing communication only where it's explicitly allowed.
3. **Layer 3 and Layer 4**:
   - Network policies work at the network layer (Layer 3) and transport layer (Layer 4), allowing you to define rules based on IP addresses and ports.
4. **Declarative Configuration**:
   - You define network policies in YAML files, making it easy to manage and version control.

Example of a Network Policy:

Here's a simple example of a network policy that allows only certain pods to communicate:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-access
  namespace: my-namespace
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
```

In this example:

- **podSelector**: Specifies the pods (in this case, those labeled with role: db) that this policy applies to.
- **ingress**: Specifies the allowed traffic. Here, only pods labeled role: frontend can send traffic to the db pods.

Summary:

- Network policies in Kubernetes are essential for securing and controlling pod communication. By defining rules, you can ensure that only authorized pods can interact with each other, enhancing your application's security posture.

What is default service in Kubernetes?

ClusterIP is the default type of service, which is used to expose a service on an IP address internal to the cluster.

Deployment in Kubernetes:

1. REPLICA SET: A ReplicaSet in Kubernetes is a type of controller that ensures a specified number of identical pods are running at all times. Here's an easy description of its key features and purpose:

- **Purpose**: The main job of a ReplicaSet is to maintain a stable set of replica pods running at any given time. If a pod crashes or is deleted, the ReplicaSet automatically creates a new one to replace it.
- **Desired State:** You define how many replicas (identical copies of a pod) you want. The ReplicaSet makes sure that this number is always maintained.
- **Pod Template:** It uses a pod template to know how to create new pods. This template specifies things like the container image, ports, and labels.
- **Label Selector:** The ReplicaSet uses labels to identify the pods it should manage. This helps in tracking and managing the correct set of pods.

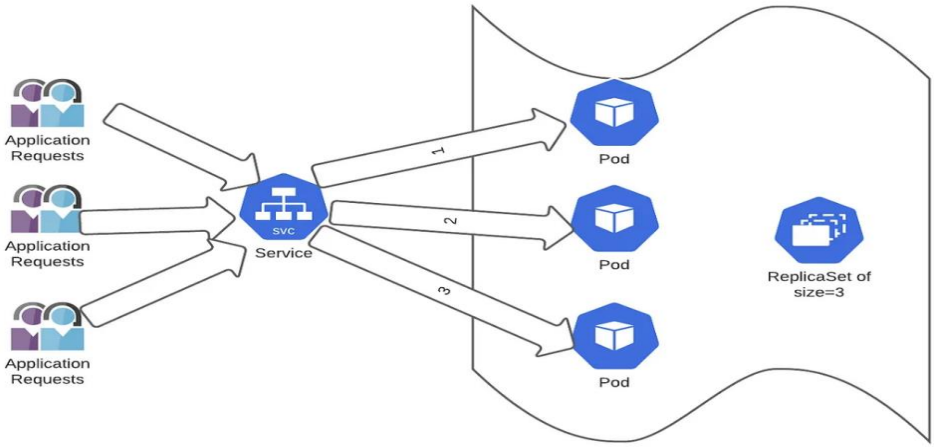
Drawbacks of ReplicaSets

- **Manual Management:** You have to create, update, and delete them manually, which can be complex.
- **No Rolling Updates:** They don’t support automatic rolling updates; you need to handle updates yourself.
- **Limited Health Checks**: They only restart crashed pods but don’t manage unhealthy containers well.
- **Complex Label Selectors:** Managing labels can get tricky, leading to possible misconfigurations.
- **Scaling Challenges:** Scaling up or down requires manual adjustments, which can be cumbersome.
- **No Deployment Strategies:** They don’t support advanced deployment strategies like canary or blue-green deployments.

Summary: ReplicaSets are good for ensuring pod availability, but they require more manual work and lack some advanced features

Example Scenario

Imagine you have a web application that you want to run with three identical copies (replicas). If one copy goes down, the ReplicaSet will automatically create a new one to ensure you always have three running.




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

2. **DEPLOYMENT:** A **Deployment controller** in Kubernetes manages the deployment of applications. It ensures that the desired number of pod replicas are running and allows for easy updates, scaling, and rollback of applications.

   Key Features of a Deployment

   - **Automates Updates**: Easily update your application without downtime using rolling updates.
   - **Scaling**: Adjust the number of pod replicas quickly based on demand.
   - **Self-Healing**: Automatically replaces failed or deleted pods to maintain the desired state.
   - **Versioning**: Keeps track of different versions of your application, allowing you to roll back to a previous version if needed.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
spec:
  replicas: 3  # Number of pod replicas
  selector:
    matchLabels:
      app: my-nginx  # Label selector to identify pods
  template:
    metadata:
      labels:
        app: my-nginx  # Labels applied to the pods
    spec:
      containers:
        - name: nginx-container  # Name of the container
          image: nginx:latest  # Container image
          ports:
            - containerPort: 80  # Port the container listens on
```

3. **StatefulSet:** A **StatefulSet** in Kubernetes is a specialized controller used to manage stateful applications. It provides unique features to handle the deployment and scaling of a set of pods, ensuring that each pod has a stable identity and persistent storage.

   What is a StatefulSet?

   - **Purpose**: StatefulSets are designed for applications that require stable, unique network identifiers and stable storage. This is especially important for databases and other stateful applications.
   - **Stable Identity**: Each pod in a StatefulSet has a unique identity, which includes a stable hostname derived from the StatefulSet name and a unique ordinal index (e.g., web-0, web-1, etc.).
   - **Ordered Deployment and Scaling**: Pods are created, deleted, and scaled in a defined order. For instance, pods are started one at a time and in sequence, ensuring that the application's state is preserved.
   - **Persistent Storage**: StatefulSets typically use PersistentVolumeClaims (PVCs) to ensure that data is preserved even if the pod is restarted or rescheduled.

Key Features of StatefulSets

- **Unique Network Identity**: Each pod has a consistent network identity that remains the same across restarts.
- **Stable Storage**: Each pod can be associated with its own persistent storage, allowing it to maintain state.
- **Ordered Deployment**: Pods are deployed in order, which is critical for applications that require careful initialization.
- **Graceful Scaling**: Scaling up or down is done in a controlled manner, preserving the application's state.
-

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: "redis"
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: redis:latest
          ports:
            - containerPort: 6379
          volumeMounts:
            - name: redis-storage
              mountPath: /data
  volumeClaimTemplates:
    - metadata:
        name: redis-storage
      spec:
        accessModes: ["PersistentVolumeClaim"]
        resources:
          requests:
            storage: 1Gi
```

4. **Blue-Green Deployment :** Blue-Green Deployment is a strategy used to minimize downtime and reduce risk during application updates. In a Kubernetes context, it involves maintaining two separate environments: the "blue" (current version) and the "green" (new version). Here's a breakdown of how it works and how to implement it in Kubernetes:

Overview of Blue-Green Deployment

- **Two Environments**:
  - **Blue**: The live version of your application.
  - **Green**: The new version that you want to deploy.
- **Switching Traffic**: Once the green environment is fully tested and ready, you switch the traffic from blue to green.
- **Rollback**: If something goes wrong, you can quickly switch back to the blue environment.

Steps to Implement Blue-Green Deployment in Kubernetes

1. **Set Up Two Deployments**:
   Create two separate deployments in Kubernetes, one for blue and one for green. For example:

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

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

2. **Service Configuration:** Create a Kubernetes service that will route traffic to one of the deployments. Initially, it will point to the blue deployment:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  selector:
    app: my-app
    version: blue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

3 Deploy the Green Version:

Deploy the green version of your application alongside the blue version. Make sure it passes all necessary tests.

4. Switch Traffic:

Update the service selector to point to the green deployment:

-
```yaml
spec:
  selector:
    app: my-app
    version: green
```

Apply the changes with kubectl apply -f service.yaml.

5. **Monitor and Validate**:

- Monitor the green deployment to ensure it is functioning correctly. Validate that users are accessing the new version as expected.

6 Rollback (if needed):

- If you encounter issues, revert the service selector back to the blue deployment to quickly rollback to the previous version.

7.Clean Up:

- Once the green deployment is confirmed stable and successful, you can scale down or delete the blue deployment.

  Considerations

- **Database Migrations**: Be cautious with database changes; ensure they are backward-compatible.
- **Traffic Routing**: You can use more sophisticated routing mechanisms, such as an Ingress controller, to manage traffic.
- **Automation**: Consider automating the deployment and rollback processes with CI/CD tools for efficiency and consistency.
- **Monitoring and Logging**: Implement proper monitoring and logging to catch any issues early.

  This strategy provides a robust way to manage application updates, minimizing risks and ensuring a smooth user experience.

Blue-Green Deployment is ideal for:

1. **Zero Downtime**: Ensures users experience no interruptions during updates.
2. **High-Stakes Applications**: Reduces risk for critical services needing reliability.
3. **Frequent Releases**: Supports safe, regular updates without user impact.
4. **Testing in Production**: Allows real-time testing of new features alongside the current version.
5. **Gradual Rollouts**: Facilitates monitoring before fully committing to new versions.
6. **Complex Deployments**: Isolates testing of updates in multi-component systems.
7. **Rollback Needs**: Enables quick reversions if issues arise.
8. **Infrastructure Changes**: Validates new environments before full migration.
9. **User Acceptance Testing (UAT)**: Allows selective testing without affecting all users.

   In summary, it's great for minimizing risk and ensuring smooth transitions during application updates.

5. **Canary Deployment:** It is a gradual rollout strategy where the new version of the application is released to a small subset of users first (the "canary"). If successful, the deployment is scaled to all users, reducing risk by testing the update on a smaller group first.

What is a Canary Deployment?

Canary Deployment is a strategy where a new version of an application is released to a small subset of users before a full rollout. This allows you to monitor for issues without affecting all users.

Steps to Implement Canary Deployment in Kubernetes

1. **Set Up Your Initial Deployment**:
   - Create your main deployment (e.g., app-v1) that serves all users.

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

2. **Create the Canary Deployment**:
    - Deploy the new version (e.g., app-v2) alongside the current version.
    -
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-v2
spec:
  replicas: 1  # Start with one pod for canary
  selector:
    matchLabels:
      app: my-app
      version: v2
  template:
    metadata:
      labels:
        app: my-app
        version: v2
    spec:
      containers:
      - name: my-app
        image: my-app:v2
```

3. **Set Up a Service**:
    - Create a service that points to both versions. Adjust the weights to control traffic.

-
```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

4. **Route Traffic to the Canary**:
    - If using a service mesh or an ingress controller, configure it to route a small percentage (e.g., 10%) of traffic to the canary deployment.
5. **Monitor the Canary**:
    - Keep an eye on logs, performance metrics, and user feedback to ensure the new version is working well.
6. **Gradual Rollout**:
    - If the canary version performs well, gradually increase the number of replicas and traffic to it while decreasing the original version.
    -
```
# Scale up the canary
kubectl scale deployment my-app-v2 --replicas=3
```
7. **Full Rollout**:
    - Once confident in the new version, switch all traffic to the canary and scale down or remove the old version.
    -
```
kubectl delete deployment my-app-v1
```
8. **Clean Up**:
    - After the full rollout, you can remove the canary deployment or keep it for future updates.

Conclusion

Canary Deployments allow you to test new features safely with a small group of users before rolling out to everyone, reducing the risk of introducing issues in your application.

Canary Deployment is useful in these situations:

1. **Gradual Feature Rollouts**: Introduce new features slowly to see how users react.
2. **High-Risk Changes**: Test major updates carefully to catch any problems early.
3. **Performance Monitoring**: Check the impact of performance changes on a small group of users.
4. **User Feedback**: Gather real user opinions on new features before a full launch.
5. **A/B Testing**: Compare different versions to see which one users prefer.
6. **System Stability**: Keep the system stable while testing new versions.
7. **Complex Apps**: Safely test updates in multi-part applications.
8. **Frequent Releases**: Ensure quality with regular updates without slowing down development.

6. **DEOMON SET:** It is a type of controller ensures a copy of specific pod is running on all nodes in cluster. Used for deploying system level or cluster wide services such as log collectors, monitoring agents and network plugins ,which need to run on every node in the cluster.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: simple-logger
  namespace: kube-system
spec:
  selector:
    matchLabels:
      app: simple-logger
  template:
    metadata:
      labels:
        app: simple-logger
    spec:
      containers:
      - name: logger
        image: busybox
        command: ["sh", "-c", "while true; do echo 'Logging...'; sleep 10; done"]
```

7. **What is HPA? How does a Horizontal PodAutoscaler work?**

Horizontal Pod Autoscaler (HPA) is a Kubernetes feature that automatically adjusts the number of pod replicas in a Deployment or ReplicaSet based on observed metrics. It helps ensure that your applications can handle varying workloads effectively and efficiently.

Key Features of HPA:

1. Dynamic Scaling:
    - HPA automatically scales the number of pods up or down in response to changes in demand. For example, during peak usage times, HPA can increase the number of pods to manage the load.
2. Metric Monitoring:
    - HPA uses metrics to make scaling decisions, such as CPU utilization, memory usage, or custom metrics defined by the user. You can specify thresholds for these metrics to trigger scaling.
3. Configuration:
    - You can set minimum and maximum replica counts. For example, you might want at least 2 pods running but no more than 10.
4. Integration:
    - HPA works in conjunction with Kubernetes Deployments or ReplicaSets, allowing it to manage pods without requiring manual intervention.

How HPA Works:

1. Monitoring: HPA continuously monitors the defined metrics from the pods in the target Deployment.
2. Scaling Decision: When the average metric exceeds a specified threshold, HPA decides to scale up the number of replicas. Conversely, if the metric drops below a certain point, it may scale down.

3. Update Deployment: HPA updates the desired number of replicas in the Deployment, which triggers Kubernetes to create or terminate pods as needed.

Example Use Case:

- If you have a web application that experiences traffic spikes during certain hours, you can configure HPA to scale up the number of pods during those peak times and scale down during off-peak hours, ensuring optimal resource usage and performance.

Summary:

HPA is a powerful tool for managing application scalability in Kubernetes, helping maintain performance while optimizing resource usage based on real-time demand.

-
```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: AverageUtilization
        averageUtilization: 80
```

How HPA Autoscaling work:

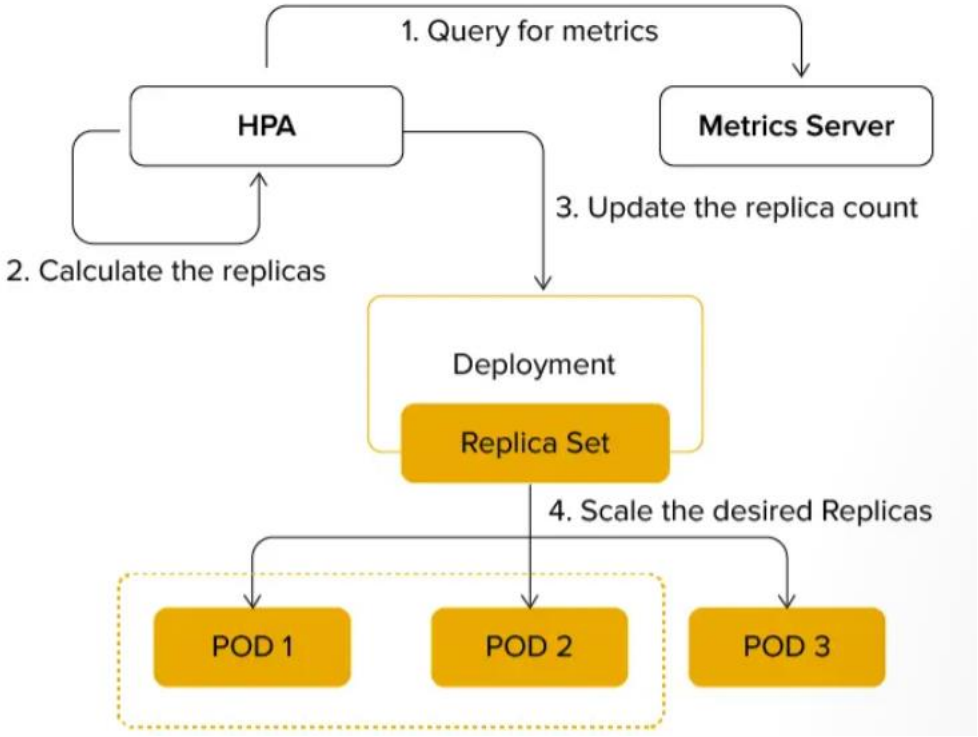
1. Horizontal Pod Autoscaling keeps monitoring the metrics server for resource usage.

2. HPA will calculate the required no of replicas on the basis of collected resource usage.

3. Then HPA decides to scale up the application to the no of replicas required.

4. After that, HPA will change the desired no of replicas.

5. Sinch HPA is monitoring on a continuous basis, the process repeat from step 1



Suppose I have a Deployment YAML with 2 replicas, and I create an HPA with a minimum of 1 replica and a maximum of 10. If the load increases and the HPA calculates that 6 replicas are needed, will it create 3 Deployment YAML files, or how does it work?

In Kubernetes, a Deployment manages the desired state of a set of pods, including the number of replicas. When you create a Horizontal Pod Autoscaler (HPA) for a Deployment, it does not create multiple replicas of the Deployment itself but instead adjusts the number of pods (replicas) that the existing Deployment manages.

Scenario Explanation

1. Initial Setup:
    - You have a Deployment defined with a specified number of replicas (e.g., 2 replicas).
    - You have created an HPA for this Deployment, with a minimum of 1 replica and a maximum of 10 replicas.
2. Scaling Behavior:
    - If the load increases and the metrics (e.g., CPU or memory usage) indicate that more replicas are needed, the HPA calculates how many additional replicas are required.
    - For example, if the HPA determines that 6 replicas are needed based on the metrics, it will update the existing Deployment's desired state.
3. HPA Action:
    - The HPA will change the desired replica count of the existing Deployment from 2 to 6.
    - It does this by modifying the spec.replicas field of the Deployment, not by creating multiple copies of the Deployment.
4. Resulting State:
    - The Deployment now specifies that 6 pods should be running. Kubernetes will automatically create 4 additional pods to meet this desired state (going from 2 to 6 total replicas).

- If the load decreases, the HPA can also scale down the number of replicas, adjusting the Deployment as needed.

Summary

- The HPA does not create additional Deployments or ReplicaSets; it modifies the existing Deployment to adjust the number of pods (replicas).
- The scaling is done at the pod level within the Deployment, and Kubernetes ensures that the desired number of pods is maintained according to the HPA's decisions.

So in your example, if the HPA determines that 6 replicas are needed, it will simply update the existing Deployment to scale up to that number, and you will still have only one Deployment managing all those replicas.

If I have an HPA implemented for a Deployment with 2 replicas, and I run a scale-out command, will the scaling happen at the Deployment level or the HPA level?

When you have a Horizontal Pod Autoscaler (HPA) implemented on a Deployment with 2 replicas, and you run a scale-out command (for example, using kubectl scale deployment my-app --replicas=4), the following will happen:

1. Scale at Deployment Level

- The scale-out command directly changes the desired state of the Deployment to have 4 replicas.
- Kubernetes will then create additional pods to meet this desired state, regardless of the HPA settings.

2. HPA Behavior

- The HPA will continue to monitor the metrics (like CPU or memory utilization) of the pods.
- If the metrics indicate that the average utilization exceeds the defined threshold, the HPA may scale the pods further based on its own logic.
- Conversely, if the metrics drop below the threshold, the HPA may scale down the number of replicas.

Summary

- The scale-out command you issued operates at the Deployment level and sets the desired replicas to 4.
- The HPA will also operate based on the metrics and may adjust the number of replicas further (either increasing or decreasing) as needed.

Interaction

- The HPA and the Deployment can both influence the number of replicas, but they operate independently.
- If you keep scaling manually and the HPA is also active, you might see a bit of back and forth as both systems try to adjust the number of replicas based on their respective logic.

In essence, the Deployment handles the manual scale changes, while the HPA automates scaling based on resource metrics.

Pod Monitoring:

What are probe? Explain different type of probe in Kubernetes?

Ans: In Kubernetes, **probes** are a way to check the health and status of your application containers. They help ensure that your applications are running smoothly and can automatically restart or remove unhealthy containers. There are three main types of probes in Kubernetes:

1. Liveness Probes:

- Purpose: Liveness probes determine if a container is still running. If a liveness probe fails, Kubernetes will restart the container.
- Use Case: Use liveness probes to check if your application has become unresponsive or stuck in a state where it can't recover.
- Example: You might configure a liveness probe to check an HTTP endpoint of your application:

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
```

2. Readiness Probes

- Purpose: Readiness probes determine if a container is ready to accept traffic. If a readiness probe fails, Kubernetes will stop sending traffic to that container.
- Use Case: Use readiness probes during startup, to check if the application has completed initialization, or to ensure it can handle requests.
- Example: You might configure a readiness probe to check an HTTP endpoint:

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

3. Startup Probes

- Purpose: Startup probes are used to determine if an application has started successfully. They can be helpful for applications that take a long time to initialize.
- Use Case: If a startup probe fails, Kubernetes will restart the container. Once the startup probe succeeds, the liveness and readiness probes take over.
- Example: You might configure a startup probe like this:

```
startupProbe:
  httpGet:
    path: /start
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

Summary

- Liveness Probes: Check if the application is alive; if it fails, Kubernetes restarts the container.
- Readiness Probes: Check if the application is ready to handle traffic; if it fails, Kubernetes stops sending requests.
- Startup Probes: Check if the application has started successfully; if it fails, Kubernetes restarts the container.

Pod Type:

1. Static pod: A static pod is a pod that is managed directly by the kubelet on a specific node, rather than through the Kubernetes API server.

Key Features

- **Created by Kubelet:** Unlike regular pods, which are usually created via the Kubernetes API, static pods are defined in configuration files that the kubelet reads.
- **Configuration File Path:** We define static pods in YAML files and place them in a specific directory on the node. Path: /etc/Kubernetes/manifests/. The kubelet watches this directory for changes.
- **No Controller:** Static pods are not managed by a higher-level controller like Deployments or ReplicaSets. They exist independently and are restarted by the kubelet if they fail.
- **Useful for System Pods:** They are often used for essential system components or for debugging purposes because they ensure that critical pods are always running on a particular node.

When to Use Static Pods

- **Critical Components**: If you need to run components like etcd or the Kubernetes API server itself, which should always be present and directly tied to a specific node.
- **Development and Testing:** Static pods are useful for developers because they allow for quick and easy setup without the overhead of the full Kubernetes deployment process. This makes them great for testing and debugging scenarios.

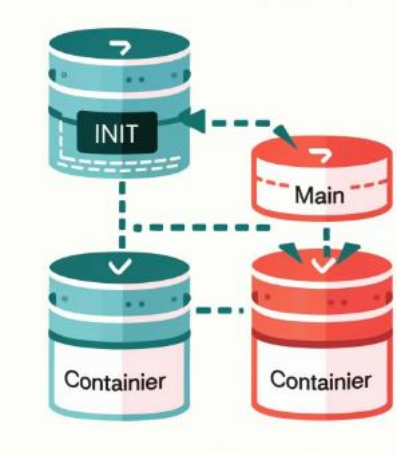
2. Init container: An init container is a special type of container that runs before the main application containers in a pod.

Key Features

- **Runs First:** Init containers always run first. They must complete successfully before the main application containers start.
- **One or More:** You can have multiple init containers in a pod. They will run in the order you define them.
- **Purpose: They** are typically used for tasks that need to be completed before the main application starts. This could include:
    - Setting up configurations.
    - Downloading files or dependencies.
    - Waiting for a service to be ready.
- **Temporary:** Once an init container finishes its task, it is not reused. It exits, and only the main application containers continue to run.

Example Use Case : Imagine you have a web application that needs to connect to a database:

- **Init Container:** You could use an init container to wait until the database is ready to accept connections before starting your web app.
- **Main Container:** After the init container successfully completes, the web application container starts.




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  initContainers:
    - name: init-db
      image: busybox
      command: ['sh', '-c', 'echo "Initializing database..." && sleep 5']

  containers:
    - name: main-app
      image: my-web-app-image
      ports:
```

3. Sidecar Containers: A sidecar container is like a helper that runs next to your main application in a container. They work together in the same space (called a pod in Kubernetes).

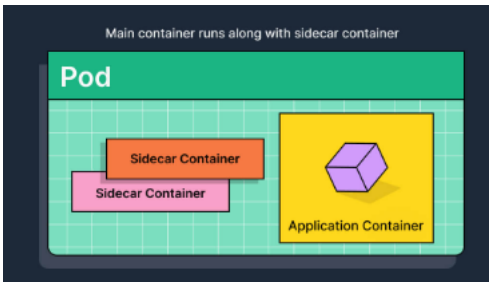
Key Points

- **Works Together:** The sidecar container is always with the main container. They share resources and communicate easily.
- **Extra Features:** The sidecar provides additional functions, such as:
  - **Logging**: It can collect and send logs from the main application to another place for storage or analysis.
  - **Monitoring**: It can keep track of the main application's performance and health.
  - **Proxying:** It can manage network requests to the main application, like load balancing or security.
- **Starts First:** Often, the sidecar starts before the main application. This helps set up everything needed for the main app to run smoothly.

Simple Example

Imagine you have a car (the main application) and a sidecar (the helper). The sidecar doesn't drive but helps with things like:

- Carrying tools for repairs (logging).
- Keeping track of how much fuel you have left (monitoring).
- Helping you navigate (proxying).




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: main-app
      image: my-main-app-image  # This is your main application
    - name: logging-sidecar
      image: fluent/fluentd  # This is the sidecar that collects logs
```

Volume in Kubernetes:

Type of volume:

1. emptyDir

- Description: A temporary directory that shares storage among containers in a pod. The data is deleted when the pod is removed.

- Use Case: For scratch space or for sharing files between containers.
-
```
volumes:
- name: my-empty-dir
  emptyDir: {}
```

2. hostPath

- Description: Mounts a file or directory from the host node's filesystem into your pod.
- Use Case: Useful for development or debugging, but not recommended for production due to tight coupling with the node.
-
```
volumes:
- name: my-host-path
  hostPath:
    path: /path/on/host
```

3. NFS (Network File System)

- Description: Allows you to mount a remote NFS share into your pod.
- Use Case: For sharing data between multiple pods across nodes.
-
```
volumes:
- name: my-nfs
  nfs:
    server: nfs-server.example.com
    path: /path/to/nfs
```

4. PersistentVolume (PV)

- Description: A storage resource in the cluster that is provisioned by an administrator or dynamically via a Storage Class.
- Use Case: For long-term storage that persists beyond pod lifecycles.
-
```
volumes:
- name: my-pv
  persistentVolumeClaim:
    claimName: my-pvc
```

5. ConfigMap

- Description: A way to inject configuration data into pods. ConfigMaps can be mounted as files or environment variables.
- Use Case: For managing application configuration.
-
```
volumes:
- name: my-config
  configMap:
    name: my-configmap
```

6. Secret

- Description: Similar to ConfigMaps, but designed to hold sensitive data such as passwords or tokens.
- Use Case: For securely managing sensitive information.

-
```yaml
volumes:
- name: my-secret
  secret:
    secretName: my-secret
```

7. Cinder (OpenStack)

- Description: Used to mount OpenStack Cinder volumes in pods.
- Use Case: For Kubernetes clusters running on OpenStack.
-
```yaml
volumes:
- name: my-cinder
  cinder:
    volumeID: my-volume-id
    fsType: ext4
```

8. Azure Disk

- Description: Mounts an Azure Disk to a pod.
- Use Case: For Kubernetes clusters running in Azure.
-
```yaml
volumes:
- name: my-azure-disk
  azureDisk:
    diskName: my-disk
    diskURI: my-disk-uri
    readOnly: false
```

9. AWS EBS (Elastic Block Store)

- Description: Mounts an AWS EBS volume to a pod.
- Use Case: For Kubernetes clusters running on AWS.
-
```yaml
volumes:
- name: my-ebs
  awsElasticBlockStore:
    volumeID: aws://region/volume-id
    fsType: ext4
```

10. RBD (Ceph RADOS Block Device)

- Description: Allows you to mount a Ceph RBD volume.
- Use Case: For environments using Ceph as the storage backend.
-
```yaml
volumes:
- name: my-rbd
  rbd:
    monitors:
      - ceph-monitor1:6789
    pool: rbd
    image: my-image
    fsType: ext4
```

1. Persistent Volume (PV)

- What it is: A PV is like a piece of storage in your Kubernetes cluster. It can be thought of as a hard drive that is available for use.

- Key Points:
    - Admins set up PVs to provide storage that can be used by applications.
    - PVs exist independently of the pods that use them.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 5Gi  # Size of the volume
  accessModes:
    - ReadWriteOnce  # Access mode for the volume
  hostPath:
    path: /mnt/data  # Path on the host node (for testing/development)
```

2. Persistent Volume Claim (PVC)

- What it is: A PVC is a request for storage by a user. It's like asking for a specific amount of storage space from the available PVs.
- Key Points:
    - Users create PVCs to claim the storage they need.
    - When a PVC is created, Kubernetes finds a matching PV to fulfill that request.
    -

```yaml
# pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: ebs-storage  # Reference to the Storage Class
```

3. Storage Class

- What it is: A Storage Class defines different types of storage available in the cluster. It helps manage how storage is created and its characteristics (like speed and type).
- Key Points:
    - Storage Classes specify the type of storage (e.g., fast SSDs, standard disks).
    - When a PVC is created, you can specify which Storage Class to use, allowing for dynamic storage provisioning.
    -

```yaml
# storage-class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2  # General Purpose SSD
  fsType: ext4
```

Step by Step Implementation of PV, PVC and POD

Step 1: Create a Persistent Volume (PV)

First, create a Persistent Volume definition. This example uses a local path for demonstration purposes:

```yaml
# pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data  # Change this to a suitable path on your node
```

kubectl apply -f pv.yaml

Step 2: Create a Persistent Volume Claim (PVC)

Next, create a PVC that requests storage from the PV:

```yaml
# pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

kubectl apply -f pvc.yaml

Step 3: Verify PV and PVC

Check if the PV and PVC are created and bound:

kubectl get pv

kubectl get pvc

You should see that your PVC is bound to the PV.

Step 4: Create a Pod that Uses the PVC

Now, create a Pod that uses the PVC:

```yaml
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx  # Example image
    volumeMounts:
    - mountPath: /usr/share/nginx/html  # Path inside the container
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc  # Reference to the PVC
```

Apply the Pod:

kubectl apply -f pod.yaml

Step 5: Verify the Pod

Check if the Pod is running:

kubectl get pods

we should see your Pod in the Running state.

Step 6: Access the Pod

To access the Pod and verify the volume mount, you can execute a shell inside the Pod:

kubectl exec -it my-pod -- /bin/sh

Inside the Pod, you can navigate to /usr/share/nginx/html to see the mounted volume.

Summary

1. Create a Persistent Volume (PV): Defines the storage resource.
2. Create a Persistent Volume Claim (PVC): Requests a specific amount of storage.
3. Create a Pod: Uses the PVC to mount the storage.
4. Verify: Check that the PV, PVC, and Pod are properly created and bound.

This process allows you to manage persistent storage in a Kubernetes cluster effectively.

Step by Step Implementation of StorageClass, PVC and POD

Step 1: Create a Storage Class

First, create a Storage Class that specifies the type of storage to be used. Here's an example for AWS EBS (Elastic Block Store):

```yaml
# storage-class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2  # General Purpose SSD
  fsType: ext4
```

Apply the Storage Class:

kubectl apply -f storage-class.yaml

Step 2: Create a Persistent Volume Claim (PVC)

Now, create a PVC that uses the Storage Class to request storage:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: ebs-storage  # Reference to the Storage Class
```

Apply the PVC:

kubectl apply -f pvc.yaml

Step 3: Verify the PVC and Dynamic Provisioning

Check if the PVC is created and bound:

kubectl get pvc

You should see that your PVC is bound to a dynamically provisioned Persistent Volume.

Step 4: Create a Pod that Uses the PVC

Now, create a Pod that uses the PVC:

```yaml
# pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx  # Example image
    volumeMounts:
    - mountPath: /usr/share/nginx/html  # Path inside the container
      name: my-volume
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc  # Reference to the PVC
```

Apply the Pod:

kubectl apply -f pod.yaml

Step 5: Verify the Pod

Check if the Pod is running:

kubectl get pods

You should see your Pod in the Running state.

Step 6: Access the Pod

To access the Pod and verify the volume mount, you can execute a shell inside the Pod:

kubectl exec -it my-pod -- /bin/sh

Inside the Pod, navigate to /usr/share/nginx/html to see the mounted volume.

Summary

- Create a Storage Class: Defines the type and parameters of the storage to be provisioned.
- Create a Persistent Volume Claim (PVC): Requests storage using the defined Storage Class.
- Verify: Check that the PVC is bound to a dynamically provisioned Persistent Volume.
- Create a Pod: Uses the PVC to mount the storage.
- Verify Pod: Ensure the Pod is running and can access the mounted volume.

This process allows for dynamic provisioning of storage in a Kubernetes cluster using Storage Classes and PVCs.

Kubernetes Ingress

Ingress is a Kubernetes resource that manages external access to services within a cluster. It provides HTTP and HTTPS routing to services based on defined rules, allowing users to expose multiple services through a single IP address.

Ingress Controller

- Definition: An Ingress Controller is a component that listens for Ingress resource updates and configures a reverse proxy or load balancer accordingly. It implements the rules defined in Ingress resources.
- Examples:
  - NGINX Ingress Controller: The most commonly used controller that leverages NGINX as a reverse proxy.
  - Traefik: A modern reverse proxy that can automatically discover services.
  - HAProxy Ingress: Uses HAProxy as the backend for handling traffic.
- Installation: Ingress Controllers are deployed as pods in the cluster, and they require a service of type LoadBalancer or NodePort to expose external traffic.

Ingress Rules

- Definition: Ingress Rules define how HTTP/S traffic should be routed to the services within the cluster.
- Components:
  - Host: The domain name for routing (e.g., example.com).
  - Path: The specific path to match (e.g., /api).
  - Service: The target service and port to forward the traffic to.

Example Ingress Resource

Here's a sample YAML file for an Ingress resource:

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

Summary

- Ingress: Manages external access to services in a Kubernetes cluster.
- Ingress Controller: A component that implements the rules defined in Ingress resources.
- Ingress Rules: Specify how traffic should be routed based on host and path to different services.

This setup allows for efficient management of external access to multiple services with minimal exposure to the outside world.

Kubernetes Security

Kube Benchmarking: Kube Benchmarking involves assessing the performance and security configurations of a Kubernetes cluster against best practices and benchmarks (like the CIS Kubernetes Benchmark). Tools like kube-bench can be used to run checks on cluster configurations to ensure compliance with recommended security practices.

- Usage:
  - Identify security vulnerabilities.
  - Assess performance metrics.
  - Ensure best practices are followed.

Secrets

Kubernetes Secrets are used to store sensitive information, such as passwords, OAuth tokens, and SSH keys. Secrets allow you to manage this sensitive data securely and make it accessible to your applications.

- Features:
  - Base64 encoded data for storage.
  - Can be consumed by pods as environment variables, files, or directly through the API.
  - Can be created from literal values, files, or directories.

Example of Creating a Secret

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: dXNlcm5hbWU=  # base64 encoded
  password: cGFzc3dvcmQ=  # base64 encoded
```

ConfigMap

ConfigMaps are used to manage non-sensitive configuration data in Kubernetes. They allow you to separate your configuration from your application code, making it easier to change configurations without redeploying the application.

- Features:
  - Store configuration in key-value pairs.
  - Can be consumed by pods as environment variables, command-line arguments, or files.

Example of Creating a ConfigMap

-
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  app.properties: |
    key1=value1
    key2=value2
```

Key Vault Integration: Key Vault Integration allows Kubernetes to securely access secrets stored in external vaults, such as Azure Key Vault, HashiCorp Vault, or AWS Secrets Manager. This helps manage sensitive data outside the Kubernetes cluster while providing secure access to applications.

- Usage:
  - Fetch secrets dynamically from a key vault.
  - Integrate with services like Azure Key Vault to pull secrets at runtime.

Example (Azure Key Vault)

1. Create an Azure Key Vault and store secrets.
2. Use an external secret operator (like Azure Key Vault Secrets Store CSI Driver) to synchronize secrets into Kubernetes.

-
```yaml
apiVersion: secrets-store.csi.k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-keyvault-secrets
spec:
  provider: azure
  parameters:
    usePodIdentity: "false"
    keyvaultName: "myKeyVault"
    objects: |
      array:
        - |
          objectName: mySecret
          objectType: secret  # possible types: secret, key, certificate
```
Summary

- Kube Benchmarking: Evaluates cluster performance and security.
- Secrets: Securely manage sensitive data.
- ConfigMaps: Manage non-sensitive configuration data.
- Key Vault Integration: Access external secrets securely from vaults.

These components enhance the security and manageability of applications running in Kubernetes environments.

Kubernetes CRD

1. Helm :

What is helm? What is chart.yaml and value.yaml in helm?

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications on Kubernetes clusters. It allows you to define, install, and upgrade even the most complex Kubernetes applications using a simple command-line interface.

1. Templates

- Definition: Templates are files that contain Kubernetes manifest definitions with placeholders for dynamic values. They are written in YAML format and utilize the Go templating language.
- Purpose: Templates allow you to create flexible and reusable configurations. You can customize resources based on the values provided at deployment time.
- Example: A simple deployment template might look like this:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-app
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}-app
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}-app
    spec:
      containers:
        - name: app
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: {{ .Values.containerPort }}
```

2. values.yaml

- Definition: The values.yaml file is a key part of a Helm chart that defines the default configuration values for the chart.
- Purpose: It allows users to customize the behavior of the chart without modifying the templates directly. Users can override these values during installation or upgrades.
- Example: A typical values.yaml file might look like this:
-

```yaml
replicaCount: 2
image:
  repository: my-app
  tag: latest
containerPort: 80
```

3. chart.yaml

- Definition: The Chart.yaml file contains metadata about the Helm chart itself, including its name, version, and description.
- Purpose: This file helps users understand what the chart does and its dependencies. It’s essential for packaging and sharing the chart.
- Example: A simple Chart.yaml might look like this:
-

```yaml
apiVersion: v2
name: my-app
description: A Helm chart for My Application
version: 1.0.0
appVersion: 1.0
```

Summary

- Templates: Define Kubernetes resources with dynamic placeholders.
- values.yaml: Contains default configuration values for the chart.
- Chart.yaml: Provides metadata about the chart.

Together, these components make Helm a powerful tool for managing Kubernetes applications, allowing for reusable, customizable, and versioned deployments.

Service Mesh

What is a Service Mesh?
A Service Mesh is a dedicated infrastructure layer that manages service-to-service communication within a microservices architecture. It provides a way to control how different services interact with each other over a network, adding capabilities like traffic management, security, and observability without requiring changes to the application code.
Key Features of a Service Mesh

1. Traffic Management:
    - Control over the flow of traffic between services, including routing, load balancing, and retries.
    - Support for canary deployments, blue/green deployments, and traffic splitting.
2. Security:
    - Enforce security policies such as mutual TLS (mTLS) for encrypting service communication.
    - Manage authentication and authorization between services.
3. Observability:
    - Provide visibility into service interactions through metrics, logging, and tracing.
    - Enable monitoring of performance, error rates, and service latency.
4. Resilience:
    - Implement patterns such as circuit breaking and timeout handling to enhance application resilience.
    - Automatically retry failed requests and manage fault tolerance.

Popular Service Mesh Implementations

1. Istio:
    - One of the most widely used service meshes.
    - Provides extensive traffic management, security, and observability features.
2. Linkerd:
    - A lightweight service mesh that focuses on simplicity and performance.
    - Easy to install and configure, making it suitable for smaller applications.
3. Consul:
    - Offers service discovery and configuration management along with service mesh capabilities.
    - Works well with both Kubernetes and non-Kubernetes environments.
4. AWS App Mesh:
    - A managed service mesh for applications running on AWS.
    - Integrates with AWS services and provides features like traffic routing and service discovery.

How a Service Mesh Works

- Data Plane: A layer of lightweight proxies (sidecars) that intercepts all network traffic between services. These proxies handle communication, routing, and security.
- Control Plane: A centralized component that manages the configuration of the service mesh. It defines policies and routes that the data plane proxies enforce.

Example Scenario
In a microservices application, suppose you have multiple services like User Service, Order Service, and Payment Service. A service mesh can:

- Route traffic between these services intelligently, ensuring that user requests are directed to the correct service version (e.g., during canary deployments).
- Secure the communication between services using mTLS, ensuring data encryption.
- Monitor and trace requests across services, providing insights into performance bottlenecks and errors.

Summary

A service mesh enhances the capabilities of microservices by managing communication, security, and observability, allowing developers to focus on building features without worrying about the complexities of service interactions. This infrastructure layer is crucial for maintaining scalable and resilient applications in a cloud-native environment.

Istio

Istio is an open-source service mesh that provides a way to control how microservices share data with one another. It helps manage the complexities of microservice architectures by offering features such as traffic management, security, observability, and policy enforcement, all without requiring changes to the application code.

Key Features of Istio

1. Traffic Management:
    - Enables sophisticated routing rules and traffic policies, allowing developers to control the flow of traffic between services.
    - Supports canary releases, blue/green deployments, and traffic splitting for smooth transitions.
2. Security:
    - Provides strong identity and access controls using mutual TLS (mTLS) for service-to-service communication.
    - Facilitates fine-grained access control policies and encryption of data in transit.
3. Observability:
    - Integrates with monitoring tools to provide insights into service performance, error rates, and latency.
    - Supports distributed tracing and logging, helping to visualize service interactions and diagnose issues.
4. Policy Enforcement:
    - Allows the implementation of policies for rate limiting, access control, and more, helping maintain compliance and governance.
5. Platform Agnostic:
    - Istio can be deployed on any cloud or on-premises infrastructure, making it a flexible choice for managing microservices across different environments.

Architecture Components

1. Envoy Proxy:
    - A lightweight proxy that intercepts all inbound and outbound traffic to services. It is deployed as a sidecar alongside each service instance.
2. Control Plane:
    - Manages and configures the proxies, handling tasks like service discovery, traffic management, and policy enforcement. The primary component of the control plane is istiod.
3. Data Plane:
    - Composed of the Envoy proxies, which handle the actual data traffic between services.

Use Cases

- Microservices Management: Simplifies the management of microservices by handling communication complexities.
- Security: Secures service communication and enforces access policies.
- Traffic Control: Fine-tunes how traffic flows between services, enabling safer deployments.
- Monitoring: Provides insights into service performance, helping teams optimize applications.

Summary

Istio acts as a powerful intermediary for microservices, offering capabilities that improve the reliability, security, and observability of applications. It is especially useful in cloud-native environments, where microservices often communicate over complex networks.

Step-by-Step Implementation of Istio Service Mesh

Step 1: Prerequisites

1. **Kubernetes Cluster**: Ensure you have a running Kubernetes cluster (minikube, AKS, GKE, EKS, etc.).
2. **kubectl**: Install and configure kubectl to interact with your cluster.
3. **Istio CLI**: Download and install the Istio command-line interface.

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-*
export PATH=$PWD/bin:$PATH
```

Step 2: Install Istio

1. **Install Istio using the provided install profile**:

istioctl install --set profile=demo -y

This command installs Istio with a demo profile, which includes the core components needed for a service mesh.

2. **Verify the installation**:

kubectl get pods -n istio-system

Ensure all Istio components (like istiod and ingress gateways) are running.

Step 3: Configure Namespace for Istio Injection

1. **Label the namespace** where your application will run for automatic sidecar injection:

kubectl label namespace <your-namespace> istio-injection=enabled

Replace <your-namespace> with your desired namespace (e.g., default).

Step 4: Deploy Your Application

1. **Create a sample application** (for example, a simple microservice). Here's a YAML example for a basic deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloworld
  namespace: <your-namespace>
spec:
  replicas: 3
  selector:
    matchLabels:
      app: helloworld
  template:
    metadata:
      labels:
        app: helloworld
    spec:
      containers:
      - name: helloworld
        image: <your-helloworld-image>
        ports:
        - containerPort: 8080
```

2. **Apply the deployment**:

kubectl apply -f <your-deployment-file>.yaml

Step 5: Configure Ingress (Optional)

If you want to expose your service externally:

1. **Create an Istio Gateway**:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: helloworld-gateway
  namespace: <your-namespace>
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

2. **Create a Virtual Service**:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: helloworld
  namespace: <your-namespace>
spec:
  hosts:
  - "*"
  gateways:
  - helloworld-gateway
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        host: helloworld
        port:
          number: 8080
```

3. **Apply the gateway and virtual service**:

kubectl apply -f <your-gateway-file>.yaml
kubectl apply -f <your-virtual-service-file>.yaml

Step 6: Access Your Application

1. **Get the external IP** of the Istio ingress gateway:

kubectl get svc istio-ingressgateway -n istio-system

2. **Access your application** using the external IP in your browser or via curl:

curl http://<external-ip>/

Step 7: Monitor and Manage

1. **Use Istio's built-in observability features** (like Kiali, Grafana, and Jaeger) to monitor traffic and visualize service interactions.
2. **Set up policies and security** (like mutual TLS) as required for your application.

Summary

1. **Install Istio** on your Kubernetes cluster.
2. **Label the namespace** for automatic sidecar injection.
3. **Deploy your application** in the labeled namespace.
4. **Configure Ingress** to expose your services if needed.
5. **Access your application** through the Istio ingress gateway.
6. **Monitor and manage** using Istio's observability tools.

By following these steps, you can successfully implement a service mesh in your Kubernetes environment, enabling advanced traffic management, security, and observability for your microservices.

- **Kubernetes Monitoring – Prometheus, Grafana, Grafana Loki, Pixie**

Kubernetes Best Practises

- **Trivy Tool/ Clair/ Anchore Engine/ Snyk/ Docker Bench for Security/ Sysdig Secure/ Grype**

TAINTS: A taint is applied to a node and allows the node to repel pods. When a node has tainted no pod is scheduled on that node unless the pod has matching toleration.

TOLERATION: A toleration is applied to a pod and allow the pod to tolerate node with matching taints. Toleration enables pod to be scheduled onto nodes with specific taints.

AFFINITY: Allows you to specify the pod should be scheduled on a node if certain condition are meets, typically based on labels.

Types of affinity

- Node affinity: Used to specify rules for which nodes a pod can be scheduled based on labels of the nodes.
- Pod-affinity: Used to specify rules for which pods should be co-located on the same node based on the labels of the other pod running on the same node.

ANTI- AFFINITY: Prevents pods from being scheduled on same node or group of nodes.

NODE SELECTORS: Simple pod scheduling feature that allows scheduling a pod onto a node whose label match the node selector labels specified by user.

Label/Selector:

1. Label

Kya hai?
Labels ek key-value pair hote hain jo Kubernetes objects (jaise Pods, Services) ko identify karte hain. Inka use karke aap objects ko categorize aur organize kar sakte hain.

Example:

```
metadata:
  labels:
    app: my-app
    env: production
```

Visualization:
Socho aapke paas kai boxes hain, aur har box par tags hain:

- Box 1: "app: my-app, env: production"
- Box 2: "app: my-app, env: staging"

Yeh labels aapko batate hain ki kaunse boxes (pods) kis application ya environment se belong karte hain.

2. Selector

Kya hai?
Selector ek tarika hai jisse aap specific labels ke basis par objects ko filter kar sakte hain. Yeh aapko kuch criteria ke hisaab se pods ya services dhoondhne mein madad karta hai.

Example:

```
selector:
  matchLabels:
    app: my-app
```

Visualization:
Aap soch sakte hain ki aapne ek filter laga diya hai. Is filter se sirf un boxes dikhte hain jinke labels "app: my-app" hain.

Taint/Toleration

3. Taint

Kya hai?
Taints nodes par lagaye jaate hain, jo yeh specify karte hain ki kaunse pods un nodes par nahi chal sakte. Iska matlab hai ki agar ek node par taint hai, toh wahan sirf un pods ko chalne diya jaayega jo is taint ko tolerate karte hain.

Example:

```
kubectl taint nodes node1 key=value:NoSchedule
```

Visualization:
Socho, ek node (box) par ek sticker laga hai jo kehta hai, "Is node par sirf un pods ko allow kiya jayega jo key=value tolerate karte hain."

4. Toleration

Kya hai?
Tolerations pods par lagaye jaate hain jo unhe tainted nodes par chalne ki permission dete hain. Agar pod ka toleration hai, toh wo tainted node par chal sakta hai.

Example:

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"
  effect: "NoSchedule"
```

Visualization:
Aapke pod par ek sticker hai jo kehta hai, "Main is node par chal sakta hoon, chahe wahan taint ho ya na ho."

Affinity

Kya hai?
Affinity rules specify karte hain ki kaunse pods ko ek saath chalna chahiye. Yeh scheduling decision ko influence karte hain.

Types:

- **Node Affinity**: Iska matlab hai ki pods ko specific nodes par chalana.
- **Pod Affinity**: Iska matlab hai ki specific pods ke saath chalne ki preference dena.

Example (Node Affinity):

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: disktype
          operator: In
          values:
          - ssd
```

Visualization:
Aapka pod keh raha hai, "Mujhe sirf un nodes par chalna hai jahan disktype 'ssd' hai."

Anti-Affinity

Kya hai?
Anti-affinity rules specify karte hain ki kaunse pods ko ek saath nahi chalna chahiye. Yeh redundancy aur failover ke liye helpful hain.

Types:

- **Required Anti-Affinity**: Pods ko ek saath nahi chalne dena.
- **Preferred Anti-Affinity**: Pods ko saath chalne se discourage karna, lekin agar zarurat ho toh allow karna.

Example (Required Anti-Affinity):

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - my-app
      topologyKey: "kubernetes.io/hostname"
```

Visualization:
Aapka pod keh raha hai, "Mujhe us node par nahi chalna chahiye jahan mera hi dusra pod chalu hai."

- **Pod Distribution**
- **Budget**
- **Certmanager**
- **CICD with Kubernetes**

GitOPS-ArgoCD : ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. It enables you to automate the deployment of applications by syncing the desired state stored in a Git repository with the actual state of your Kubernetes cluster. Here's a breakdown of its key features:

Key Features of ArgoCD:

1. **GitOps Model**:
   - ArgoCD follows the GitOps principles, where the entire application state (including configurations and resources) is stored in Git. This allows for version control, easy rollbacks, and auditing.
2. **Declarative Configuration**:
   - You define your application's desired state using YAML manifests, which ArgoCD uses to manage the deployment and updates of your applications in Kubernetes.
3. **Automated Syncing**:
   - ArgoCD continuously monitors your Git repository for changes. When updates are detected, it can automatically synchronize the Kubernetes cluster to reflect the latest configuration.
4. **User Interface**:
   - ArgoCD provides a web-based user interface and a command-line interface (CLI) for managing applications, viewing their status, and performing operations like sync and rollback.
5. **Health and Status Monitoring**:

- It offers built-in health checks and status assessments, allowing you to see if your applications are running as expected or if there are issues.

6. **Multi-Cluster Support**:
    - ArgoCD can manage applications across multiple Kubernetes clusters, providing a unified way to deploy and monitor applications in different environments.

Summary:

In summary, ArgoCD is a powerful tool for implementing GitOps workflows in Kubernetes, making it easier to manage application deployments, maintain consistency, and ensure that your cluster's state matches the desired configuration stored in Git.

Role-Based Access Control (RBAC): RBAC is a security mechanism in Kubernetes that allows you to control access to resources based on the user's role and permissions. in RBAC, you define roles and cluster roles that specify a set of permissions, such as read, write, or delete, for a particular set of resources. You then create role bindings and cluster role bindings that associate roles and cluster roles with users, groups, or service accounts.