KUBERNETES NOTES: SIMPLIFIED FOR EVERYONE

By Rakesh Kumar Jangid

Table of Contents

- Set User for the Current Context

5. Deployment Strategy and Its Types

6. Create & Manage Kubernetes Resources

- Command Line Method (CLI Method)
- Direct Edit Method (etcd Database)
- YAML Manifest Files (File Method)

7. Namespace in Kubernetes

- What is a Namespace in Kubernetes?
- Namespace-Dependent Resources vs. Independent Resources

8. Core Kubernetes Workloads

- Pod
- ReplicationController (RC)
- ReplicaSet (RS)
- Deployment
- DaemonSet
- Static Pod
- Init Container

9. Kubernetes Pod Scheduling

- What is Pod Scheduling in Kubernetes?
- Why is Pod Scheduling Different from Default kube-scheduler Scheduling?

9.1 Types of Pod Scheduling

- nodeName Based Pod Scheduling (Direct)
- nodeSelector Based Pod Scheduling (Labels Based)
- Taint & Tolerations
- Cordon & Uncordon
- Affinity & Anti-Affinity

10. Kubernetes Configuration Management

- ConfigMap and Secrets in Kubernetes

11. Kubernetes Storage Management

- Persistent Volumes (PV)
- Persistent Volume Claims (PVC)
- Storage Classes

12. Kubernetes Resource Quota

13. Kubernetes Sidecar Containers & BusyBox

14. Kubernetes Networking & Services

- Service, kube-proxy & CNI

15. Kubernetes Security & Authentication

- Creating an Authentication Configuration File in Kubernetes Using API and Custom Certification Keys
- RBAC (Role-Based Access Control)

16. Kubernetes Ingress

- Ingress Controller & Ingress Resource

17. Setting Up Kubernetes v1.30.2 Cluster Using Kubeadm on Ubuntu 22.04 LTS

About the Author

Hello, I am Rakesh Kumar Jangid from Jaipur, Rajasthan, India. I am currently working at GIP Technologies Pvt Ltd, Jaipur, as a DevOps Corporate Trainer. With over 3+ years of experience, I have been deeply involved in training and mentoring professionals in DevOps and Kubernetes.

I hold multiple certifications, including:

- RHCSA EX200 (Red Hat Certified System Administrator)
- EX294 (Red Hat Certified Engineer in Ansible Automation)
- EX188 (Red Hat Certified OpenShift Administrator)
- Google CKA (Certified Kubernetes Administrator)

Why This Volume 1 of Notes?

This is not a book but rather my handwritten notes gathered over time, covering almost all possible topics in Kubernetes. I call it "Volume 1" because learning never stops, and there is always more to explore.

I created these notes to help learners, professionals, and Kubernetes enthusiasts understand concepts in a structured, practical, and easy-to-follow way. My only intention behind sharing this is to spread knowledge and good values in the universe.

Stay Connected

- LinkedIn: https://www.linkedin.com/in/rakeshkumarjangid
- My Blog: https://rakamodify.store/

Lastly, I believe that I could only put this together because of the grace of Lord Krishna and my Guru. In truth, they are the ones making everything happen.

Thank You

Rakesh Kumar Jangid

From Containers to Kubernetes: Understanding the Need

When we build an application, just writing the code is not enough. We also need to deploy it properly so that it runs smoothly, scales when needed, and works without interruption.

In the past, companies used a method called monolithic architecture, where the entire application was built as a single unit. This approach had many problems. If even a small part of the application had an issue, the whole system could fail. Scaling was difficult, and adding or updating features took a lot of time and effort.

Later, container tools like Docker and Podman made application deployment easier by packaging everything into lightweight containers. But there were still challenges. Managing one or two containers was simple, but handling hundreds or thousands of them manually was not practical.

What's missing with Docker and Podman?

While tools like Docker and Podman are great for creating and running containers (**Application Hosting Platforms**), they fall short when it comes to managing large, complex applications. Here's why:

1. Lack of Orchestration:
   Docker and Podman are great for running individual containers, but they can't manage many containers working together. When deploying complex apps (like those built with microservices), you need features like load balancing (distributing traffic evenly), scaling (adding or reducing resources), and self-healing (restarting failed containers). These features aren't available in standalone tools.

2. Manual Scaling and Deployment:
   If your app suddenly needs more power (like during a sale on an e-commerce site), scaling up with Docker or Podman means doing it manually—adding more containers yourself. Similarly, deploying updates or new versions requires hands-on work, making it slow and error-prone.

3. Limited High Availability:
   High availability means your app stays online, even if part of it fails. Without orchestration, it's hard to set up backups or failover systems to keep everything running smoothly.
4. Version Control and Rollbacks:
   If a new version of your app has a bug, you might need to go back to the previous

version. Docker and Podman don’t have built-in tools to manage versions or roll back easily if something goes wrong.

5. Network and Service Discovery:
   Containers often need to communicate with each other. For example, a web app might talk to a database container. Docker and Podman have basic networking, but they don’t automatically handle advanced tasks like finding and connecting to other containers in a large setup.

6. Zero Downtime Deployment:
   Updating an app without taking it offline is crucial for user experience. With standalone tools, ensuring zero downtime during updates is complicated and requires extra work.

What is need of Orchestration tool like Kubernetes and why this is needed?

Why Do We Need Kubernetes?

1. Scalability
   Kubernetes automatically adjusts the number of containers based on your app's traffic or workload. For example, if your app gets a traffic spike during a sale, Kubernetes can instantly add more containers to handle the load and scale down when traffic decreases, saving resources. Bytheway this type of scaling is known as Horizontal Scaling.

   ### What is Horizontal Vs Vertical Scaling?

   horizontal and vertical scaling in Kubernetes can indeed be understood and implemented on two levels:

   a) Host Server and Resource Level (Infrastructure Scaling)

   - Horizontal Scaling: Adds more physical or virtual machines (nodes) to the cluster. Kubernetes automatically distributes workloads across these additional nodes. For example, if the cluster is running out of resources, you can add another server to expand the available capacity.
   - Vertical Scaling: Increases the resources (CPU, memory, disk) of existing nodes in the cluster. This is done at the infrastructure level by upgrading the hardware or resizing virtual machines.

   b) Pod or Deployable Resource Unit Level (Application Scaling)

   - Horizontal Scaling: Adds more pods (replicas 2 to 5) of your application to handle increased workload. This is managed through Kubernetes'

deployment scaling, where new pod instances are created to share the load. There is No downtime and this is more fault tolerance.

- Vertical Scaling: Increases the resource allocation (CPU, memory) of individual pods by adjusting their resource limits and requests. Kubernetes ensures that the pod has more resources to handle its tasks efficiently. Best and simpler to manage for single-pod applications.

2. Resource Allocation
   Kubernetes ensures your hardware (servers) is used efficiently. It decides where to run containers so that no server is overloaded while others are underused. This optimized resource allocation saves costs and improves performance.
3. Self-Healing
   If a container crashes or has issues, Kubernetes detects it and replaces or restarts it automatically. It constantly monitors the health of your app to ensure everything runs smoothly without requiring manual intervention.
4. Security
   Kubernetes offers built-in security features such as:
   - Isolation between containers to prevent one from affecting others.
   - Role-based access control (RBAC) to limit who can perform certain actions.
   - Secret management for handling sensitive information like API keys and passwords securely.
5. Portability
   Kubernetes runs on any environment **(Dev, Test, Prod),** whether it's a cloud provider like AWS, Azure, Google Cloud, or an on-premises server. This portability means you can move your app between environments without making significant changes.
6. Uptime
   Kubernetes helps maintain near 100% uptime for your app. Through features like load balancing, rolling updates, and self-healing, it ensures your app stays online, even during updates or unexpected failures.
7. Support for Microservices
   Modern applications often use microservices—small, independent services that work together. Kubernetes makes managing these microservices easier by handling networking, scaling, and service discovery (letting containers find and talk to each other).
8. DevOps Support
   Kubernetes is a perfect fit for DevOps workflows. It integrates well with CI/CD (Continuous Integration/Continuous Deployment) pipelines, enabling teams to

automate deployments, testing, and rollbacks. It also supports collaboration between development and operations teams, making it a key part of the DevOps culture.

Why is Kubernetes Essential for Modern Apps?

In today's fast-moving world, applications need to be flexible, scalable, and reliable. Traditional methods of deploying applications as a single unit (monolithic architecture) don't work well anymore. With Kubernetes:

- Scalability and Uptime are ensured, so apps can handle traffic surges without downtime.
- Resource Efficiency reduces costs and maximizes hardware usage.
- Portability means apps can run anywhere, making them future-proof.
- Microservices Management simplifies working with modern, modular application designs.
- DevOps Integration helps teams deliver faster and more reliably.

What is a monolithic application, and how does it differ from a microservices based application?

Monolithic Architecture

A **monolithic architecture** is built as one big, unified system where all application dependencies, code, and layers work together as a single unit. If one part of the code or logic fails, the entire application can go down. Any changes require redeploying the entire app, which often leads to significant downtime and an increased chance of errors in the future.

Real-world examples of monolithic applications:

- **eBay** (early versions): Everything was built as one large application, making it challenging to scale and update.
- **WordPress** (before multi-site support): Early versions were monolithic, where all features were integrated tightly together.
- **Shopify** (initial stages): Initially, it was a monolithic app before transitioning to microservices as it grew.

Microservices Architecture

In contrast, a **microservices architecture** splits the application into smaller, independent parts, often running on separate systems like containers. Each part works independently, so if a change is needed in one part, the rest of the application continues running smoothly without downtime or added risk.

Real-world examples of microservices-based applications:

- **Netflix**: Moved from a monolithic architecture to microservices to handle the massive scale of users and content.
- **Amazon**: Uses microservices for different functions like order management, payment processing, and recommendations.
- **Uber**: Uses microservices to manage different parts of its app like ride requests, driver location tracking, and payment systems.

¶ *Modern applications are commonly built with a microservices approach, and Kubernetes is the ideal platform to manage and scale such architectures efficiently.*

- **Monolithic**: One big box, tightly connected, prone to failure and downtime when changes are made.
- **Microservices**: Many small boxes, independently running, allowing updates without impacting the rest of the app.

¶ *Netflix, for example, moved from monolithic to microservices to handle global demand, and Kubernetes is now a popular choice for managing such setups.*

What Is Cluster? Describe K8s Architecture With Diagram

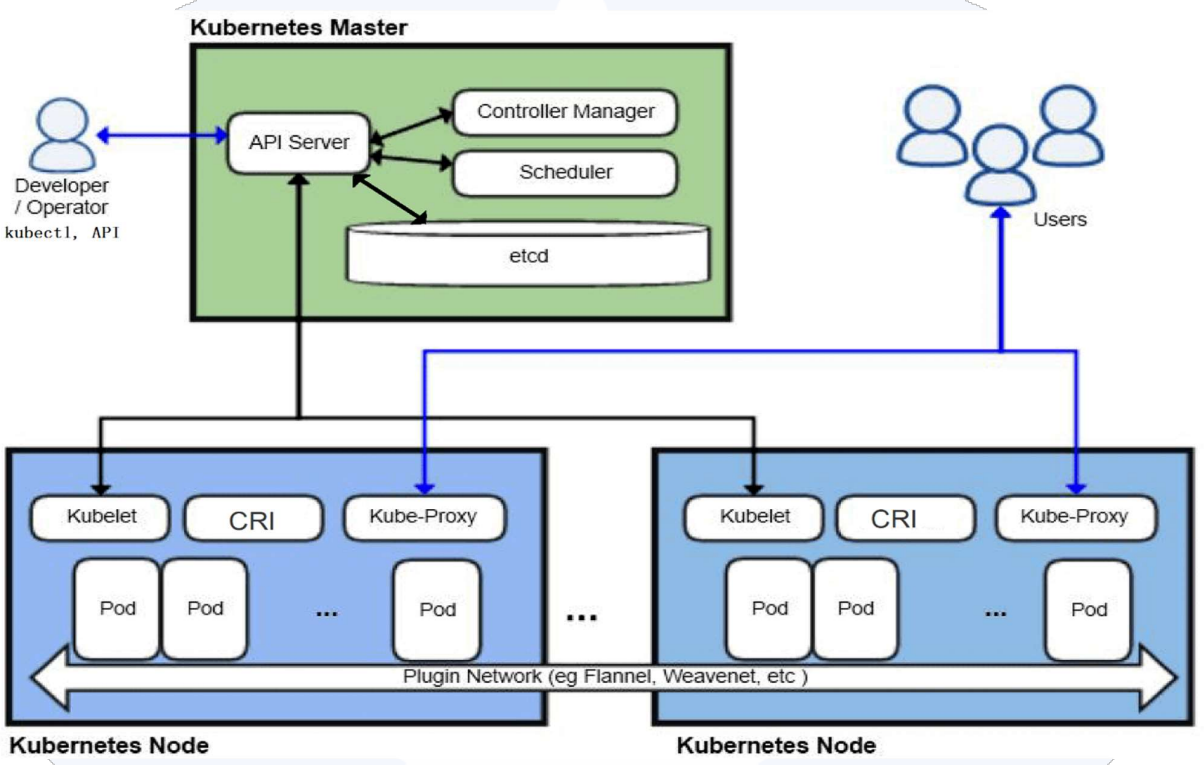
A **Kubernetes cluster** is the foundation of Kubernetes' architecture. It consists of multiple machines (nodes) that work together to manage, deploy, and run containerized applications. The cluster ensures workloads are properly distributed, running efficiently, and achieving the desired state across all components. You can think of it as a team of computers (nodes) that collaborate to deploy and manage applications, ensuring they run smoothly and can handle changes in demand. The cluster is made up of two types of nodes: **master nodes** and **worker nodes**.

Note: In-Short:

A **Kubernetes cluster** has two main components and later sub-components:

1. Master Node(s) (Control Plane)
    - API Server
    - etcd
    - Kube-Scheduler
    - Kube-Controller Manager

2. Worker Node(s) (Slave Nodes- where your Pods mostly run)
    - Kubelet
    - Kube-Proxy
    - Container Runtime



Note: In Detailed Description

Master Node Components

- API Server

  The **API Server** is the central point of interaction for the entire Kubernetes system. It processes all requests related to creating, updating, or deleting resources, ensuring they are routed to the correct parts of the system. The API Server also checks for user authorization and enforces rules, ensuring that only authorized users can make changes to the cluster. All other components communicate through the API Server to interact with the cluster, making it the 'middleman' or 'hub' for communication between all other cluster arch-components.

- etcd

  etcd saved current state of the cluster**. etcd** is a key-value store used by Kubernetes to hold its most critical data, including the current state of all resources within the cluster (such as Pods, nodes, services, etc.). This distributed database ensures data consistency across all nodes in the cluster and provides backup for the cluster's state. If anything goes wrong, **etcd** helps restore the cluster to its previous state.

- Kube-Scheduler

  The **Kube-Scheduler** is responsible for determining which worker node should run a new Pod. It evaluates the resources available on the nodes (e.g., CPU and memory) and considers any placement rules (such as affinity or taint tolerations). The scheduler then places the Pod on the most suitable node based on these factors. In short, the Kube-Scheduler determines the ideal node for running a new Pod.

- Kube-Controller Manager

  The **Kube-Controller Manager** ensures that the Kubernetes cluster always remains in its desired state. It continuously monitors **etcd** for any changes, which are triggered by requests from the API Server. Whenever the state of the cluster changes, the controller works to restore the system to the desired as per requested state by taking corrective actions, such as scaling applications, replacing failed Pods, or ensuring node health. The Kube-Controller Manager automatically performs these tasks to keep the cluster in the desired state

Worker Node Components

- Kubelet

  The **Kubelet** is an agent that runs on each worker node in the Kubernetes cluster. It receives instructions from the API Server, which runs on the master node. The Kubelet ensures that the Pods, the smallest deployable units in Kubernetes, are running as expected. It also continuously monitors the health of the node and its Pods, reporting the status back to the master node API Server to ensure the cluster remains in the desired state.

- Kube-Proxy

  Kube-Proxy is the default networking component in Kubernetes responsible for managing network traffic for Pods and Services. When you create a **Service** in Kubernetes, **Kube-Proxy** automatically manages the network routing and ensures that traffic is correctly forwarded to the appropriate Pods as per dedicated services,

regardless of the nodes they are running on. It performs load balancing by distributing traffic across multiple Pods to prevent overloading any single Pod. Additionally, Kube-Proxy manages network rules using **iptables** or **IPVS** on each node, ensuring that network traffic is routed efficiently and accurately.

While Kube-Proxy handles traffic routing and load balancing, it works alongside the **Container Network Interface (CNI)**, which handles the underlying networking setup and IP assignment for Pods. Essentially, Kube-Proxy ensures seamless communication between services and Pods, enabling proper network traffic management within the cluster.

- Container Runtime

  The **Container Runtime** is the main software behind, responsible for running and creating containers inside Pods. It performs tasks such as pulling container images, starting and stopping containers, and managing the overall lifecycle of containers. Examples of container runtimes include **containerd** and **CRI-O**. For instance, Docker uses containerd as its container runtime, while Podman uses CRI-O. The Container Runtime runs on all worker nodes and receives instructions from the Kubelet.

➢ Step-by-Step work flow

How Kubernetes Processes a Client Request:

When a client (user) interacts with Kubernetes, they send commands to the API Server using tools like kubectl or through applications using the Kubernetes API. For example, they might issue commands to create a Pod, Service, Deployment, PVC, or Quota. Here's how the process unfolds in the background:

Step-by-Step Work-Flow:

1. Client to API Server:
   The client or application sends a request to the API Server. This request is validated and authenticated by the API Server. The API Server is the central management entity that interacts with other Kubernetes components.
2. API Server and etcd:
   The API Server stores the desired state of the cluster in etcd, the distributed key-value store that acts as the source of truth for cluster data.
3. Controller Manager (kube-controller-manager):
   - The kube-controller-manager continuously monitors the state stored in etcd and compares it to the desired state defined in the request.

- If it detects any discrepancies (e.g., a Pod specified in a Deployment doesn't exist), it takes action to resolve them, such as creating a new Pod.

4. Scheduler (kube-scheduler):
    - The kube-scheduler is responsible for determining which node in the cluster is most suitable to run the Pod.
    - The scheduler makes decisions based on factors like resource availability (CPU, memory) and scheduling policies.
5. Scheduler to API Server:
    - Once the scheduler selects a node, it informs the API Server of the node assignment for the requested resource.
6. API Server to Kubelet:
    - The API Server then communicates with the Kubelet on the selected node.
    - The Kubelet is the node-level agent responsible for managing Pods and their containers.
7. Kubelet to Container Runtime:
    - The Kubelet interacts with the Container Runtime (e.g., containerd, CRI-O) to create and manage containers as specified in the request.
    - The container runtime pulls the required container images, starts the containers, and manages their lifecycle.

Additional Important Tools You Should Know About

Although **kubectl, kubeadm,** and **CNI plugins** aren't part of the core Kubernetes cluster architecture. They are tools and components that help manage, configure, and facilitate Kubernetes clusters but are not considered part of the architecture itself.

- kubectl

  kubectl is the command-line tool used to interact with a Kubernetes cluster. It allows you to run commands against the Kubernetes API server, which manages the cluster. With **kubectl**, you can perform various operations such as:

  - **Creating, updating, and deleting resources**: like Pods, Deployments, Services, etc.
  - **Monitoring and debugging**: viewing the status of resources, logs from Pods, and other information.
  - **Managing configurations**: applying and modifying configurations in the cluster.

Essentially, **kubectl** acts as the primary way for administrators and developers to communicate with and control the Kubernetes cluster.

- Kubeadm

  kubeadm is a tool for easily setting up a Kubernetes cluster. It provides commands to help you initialize the master node, join worker nodes to the cluster, and manage certain cluster configurations. It automates the process of setting up the control plane components (API server, controller manager, scheduler) and helps in creating a secure, production-grade cluster. But you still need to configure and manage the network and storage components separately, that is not the part of kubeadm cluster setup.

- CNI Networking

  CNI (Container Network Interface) is a specification and a set of plugins used for configuring networking in Kubernetes. It is responsible for providing networking features that allow containers and Pods to communicate with each other and with the outside world, even when running across multiple worker nodes and within a single deployment resource.

  CNI plugins manage things like:

  - Assigning IP addresses to containers and Pods.
  - Enabling Pod-to-Pod communication across nodes.
  - Configuring network policies to control traffic between Pods.

  In Kubernetes, CNI ensures that Pods can communicate with each other within the cluster and with external services, and it can integrate with different network plugins (like Calico, Flannel, Weave, etc.) to provide various networking features.

Kubernetes commands

Basic Kubernetes Commands:

1. List Available Kubernetes Resources

   This command displays all the Kubernetes resource types (like Pods, Services, ConfigMaps) along with their API version, short name, kind, and whether they are namespaced. It helps in understanding what resources are available and how they are grouped. This is an important command for writing and learning YAML for Kubernetes resources.

Command: # kubectl api-resources

```
root@control-k8s:~# kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
bindings                                       v1                                     true         Binding
componentstatuses                 cs           v1                                     false        ComponentStatus
configmaps                        cm           v1                                     true         ConfigMap
endpoints                         ep           v1                                     true         Endpoints
events                            ev           v1                                     true         Event
limitranges                       limits       v1                                     true         LimitRange
namespaces                        ns           v1                                     false        Namespace
nodes                             no           v1                                     false        Node
persistentvolumeclaims            pvc          v1                                     true         PersistentVolumeClaim
persistentvolumes                 pv           v1                                     false        PersistentVolume
pods                              po           v1                                     true         Pod
podtemplates                                   v1                                     true         PodTemplate
replicationcontrollers            rc           v1                                     true         ReplicationController
```

2. Generate a Token to Join Nodes

   As we know, kubeadm is a cluster setup command-line tool and this command generates a token for adding new worker nodes to a cluster and provides the join command to execute on the nodes. It's useful for scaling your cluster by adding more nodes.

Command: # kubeadm token create--print-join-command

```
root@control-k8s:~# kubeadm token create --print-join-command
kubeadm join 192.168.199.173:6443 --token hg3qwt.hc22ov14ah9l5ulh --discovery-to
ken-ca-cert-hash sha256:19749ace7b5ecceb5d6353c56a5692f3c2dabeb22ae9c9828d765c72
32d26b42
root@control-k8s:~#
```

3. Explain a Kubernetes Resource

   Provides a detailed explanation of the structure and fields of a Kubernetes resource. The--recursive flag displays all possible nested fields, making it easier to understand the complete configuration of the resource. For example, <resource-name> could be resources like Pod, Deployment, Service, PersistentVolume (PV), PersistentVolumeClaim (PVC), ConfigMap, Secret, etc.

¶ **Note:** As a professional Kubernetes administrator, you typically don't need to memorize every option in a YAML file. Instead, you would use the kubectl explain --recursive command to understand the structure and options. Additionally, the official Kubernetes documentation at kubernetes.io is an excellent resource to refer to. For practice purposes, however, we are writing YAML manifest files by including all options, such as explain, kubernetes.io, and --dry-run=client -o yaml > example.yml to generate and understand the configuration.

Command: # kubectl explain <resource-name>--recursive

```
root@control-k8s:~# kubectl explain pod --recursive
KIND:       Pod
VERSION:    v1

DESCRIPTION:
    Pod is a collection of containers that can run on a host. This resource is
    created by clients and scheduled onto hosts.

FIELDS:
  apiVersion    <string>
  kind  <string>
  metadata      <ObjectMeta>
    annotations <map[string]string>
    creationTimestamp   <string>
```

Configuration-Related Commands:

1. View Kubeconfig Content

   Directly opening and modifying the main cluster configuration file **(~/.kube/config)** is not the standard practice in organizations. The following command is a safer way to display the details of the current kubeconfig file, including clusters, contexts, users, and preferences. It helps in verifying and troubleshooting configuration settings.

   Command: # kubectl config view

```
root@control-k8s:~# kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://192.168.199.173:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
root@control-k8s:~#
```

2. Shows the currently active context

   This command indicates which cluster and user kubectl is interacting with by displaying the current context from the kubeconfig file.

Command: # kubectl config current-context

```
root@control-k8s:~# kubectl config current-context
kubernetes-admin@kubernetes
root@control-k8s:~#
```

3. Lists all users in the kubeconfig file

   This command lists all users defined in your kubeconfig file, showing the available credentials for kubectl.

   Command: # kubectl config get-users

```
root@control-k8s:~# kubectl config get-users
NAME
kubernetes-admin
root@control-k8s:~#
```

4. Renames an existing context in the kubeconfig

   This command allows you to rename a context in your kubeconfig file, making it easier to reference.

   Command: # kubectl config rename-context <old-context> <new-context>

5. List All Contexts

   Shows all the available contexts in your kubeconfig file. Each context links a cluster, a user, and a namespace, allowing you to switch between different environments. If there is a blank value under NAMESPACE, it uses the **default** namespace by default. You can check the available namespaces using the command kubectl get ns.

   Command: # kubectl config get-contexts

```
root@control-k8s:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
root@control-k8s:~#
```

6. Switch to a Different Context

   If you want to update the current active context in the kubeconfig file to a new one, this command ensures that all kubectl commands target a specific cluster and environment.

   Command: # kubectl config use-context <your-context>

7. Set Namespace for the Current Context

   Modifies the current context to use a specific namespace by default. This eliminates the need to specify the namespace for every command.

   Command: # kubectl config set-context--current--namespace=<new_namespace>

```
root@control-k8s:~# kubectl config set-context --current --namespace=grras
Context "kubernetes-admin@kubernetes" modified.
```

   And now check again the context.

```
root@control-k8s:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin   grras
```

8. Set User for the Current Context

   Changes the user associated with the current context, allowing you to switch credentials or access levels.

   Command: # kubectl config set-context--current--user=<new-user>

Deployment Strategy and its types

A **deployment strategy** in Kubernetes is how you update or run your application. It controls how new changes (like updates or scaling) are applied to ensure minimal downtime and keep your app running smoothly.

Here's a list of deployment strategies and resources in Kubernetes:

1) Pod
2) ReplicationController (RC)
3) ReplicaSet (RS)
4) Deployment
5) DaemonSet
6) Static Pod
7) Init Container

For your information, it's important to know that most deployment strategies in Kubernetes share a common YAML syntax, including fields like apiVersion, kind, metadata, and spec. You can use the kubectl explain command followed by the resource name to get help in writing the complete or required YAML manifest files as per your need.

```yaml
apiVersion: v1        # Specifies the version of the Kubernetes API
kind: Pod             # Specifies the type of Kubernetes resource
metadata:             # Metadata about resource
spec:                 # Pod Specification
```

With Detailed Description:

```yaml
apiVersion: apps/v1             # API version for the resource type
kind: Pod                       # Type of resource (Pod in this case)
metadata:
  name: example-pod             # Name of the Pod
  labels:
    app: example-app            # Labels for categorizing the Pod
spec:
  containers:
    - name: example-container   # Container name
      image: nginx              # Container image used in the Pod
      ports:
        - containerPort: 80     # Port exposed by the container
```

Create & Manage Kubernetes resources

In Kubernetes we can create and manage Kubernetes resources in several ways, depending on your needs and workflow:

- ✓ Command Line Method (CLI Method):
- ✓ Direct Edit Method (etcd Database):
- ✓ YAML Manifest Files (File Method):

- Command Line Method (CLI Method):
  You can use Kubernetes command line tool **"kubectl"** to create, update, or delete resources directly from the command line. For example:

  Command: # kubectl run <pod-name> --image=<image-name>

  ¶ Although Command Line Method (CLI Method) is not the standard way to work, because we don't have any working history with that but this method is quick for smaller tasks or when working with predefined configurations.

- Direct Edit Method (etcd Database):
  Kubernetes allows you to directly edit resources in the cluster using the kubectl edit command, which opens the resource's YAML configuration in an editor. This allows you to make changes on the fly without needing to modify a manifest file.

  Command: # kubectl edit deployment <deployment-name>

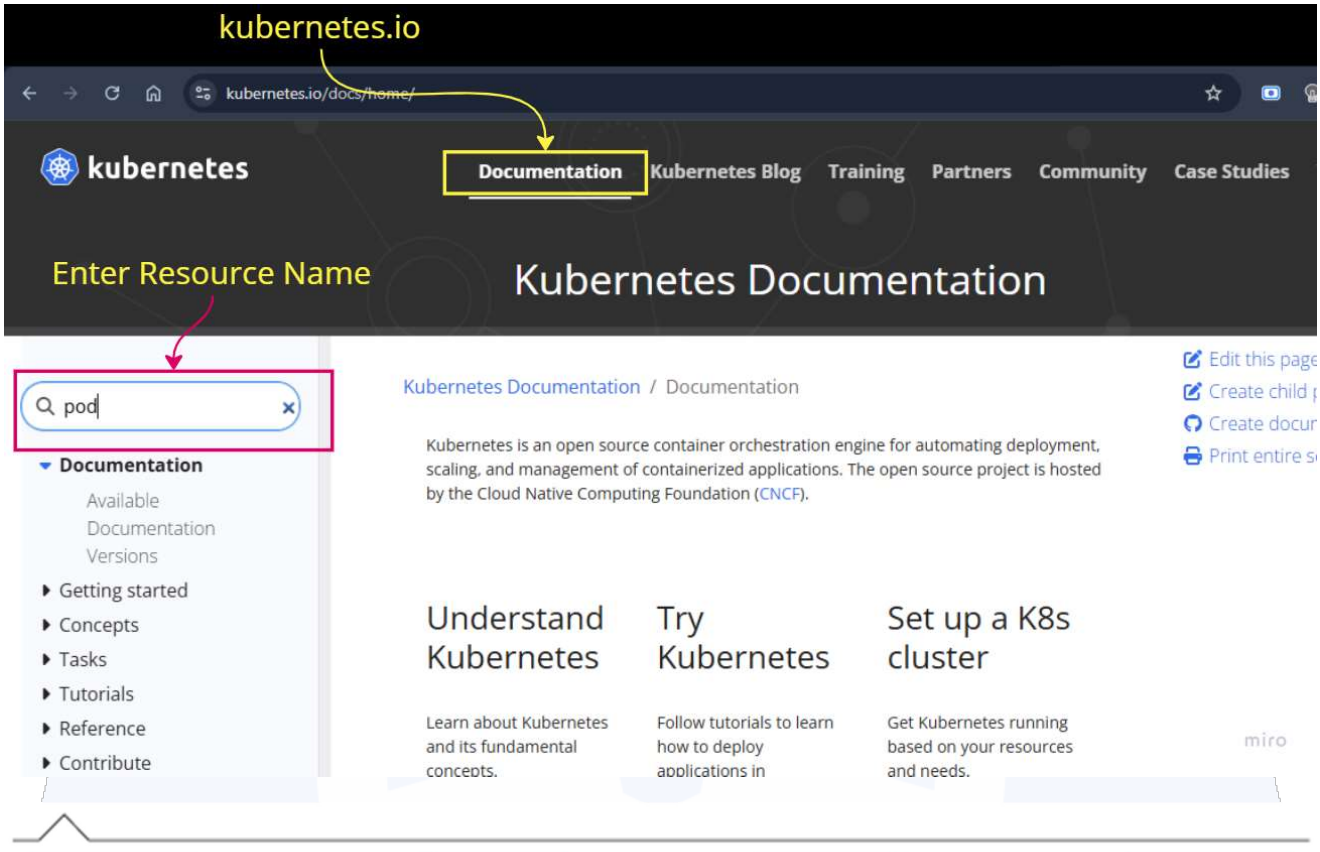
  While this method can be useful for quick, live updates, editing resources directly through the Kubernetes API server (etcd) is not the best practice for the following reasons:
  - ✓ **Risk of misconfiguration**: Incorrect changes can cause serious issues in the cluster.
  - ✓ **Lack of version control**: Direct edits are not easily traceable, making it harder to maintain consistency and track changes.

- YAML Manifest Files (File Method):
  Writing and applying YAML manifest files is the most flexible and declarative way to define Kubernetes resources. The YAML file contains the full definition of the resource, including its configuration, replicas, and other specifications. You can create a Kubernetes manifests file with any name but file extention should be either **.yaml** or **.yml** at the end of file. You can apply the file using:

  Command: # kubectl apply-f <file>.yml

Command: # kubectl apply-f <file>.yaml

This method is ideal for version control, repeatable deployments, and managing complex resources instead of running as CLI command. Kubernetes provides its own official documentation, so you don't need to create YAML files from scratch. You can directly copy or refer to the resource YAML examples available there.



Namespace (What is a Namespace in Kubernetes?)

Namespaces in Kubernetes are logical partitions within a cluster that allow you to group and isolate resources like Pods, Services, and ConfigMaps. Think of them as separate workspaces inside the same cluster. Namespaces are especially useful when managing large-scale systems involving multiple teams, projects, or environments.

¶ Example for Better Understanding:

Just like in Docker or Podman, you can run two separate applications without conflicts. How does that happen? Because each container runs in its own namespace, which logically isolates the environment. Similarly, in Kubernetes, you can isolate application-related resources like Pods, Deployments, Services, Secrets, Volumes, and even access controls by using namespaces. You can create your own namespace, and use accordingly.

- How to create a namespace?

Command: #kubectl create namespace <namespace>

```
root@control-k8s:~# kubectl create namespace grras
namespace/grras created
```

- How to delete a namespace?

  Command: #kubectl delete namespace <namespace>

```
root@control-k8s:~# kubectl delete ns grras
namespace "grras" deleted
```

- How to see all a namespaces?

  Command: #kubectl get ns

```
root@control-k8s:~# kubectl get namespaces
NAME              STATUS   AGE
default           Active   9d
grras             Active   9s
kube-node-lease   Active   9d
kube-public       Active   9d
kube-system       Active   9d
```

- How to check what is my default or current namespace?

  Command: #kubectl config get-contexts

```
root@control-k8s:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
root@control-k8s:~#
```

Note: But here namespace is showing blank, what does it mean?

If you see that the namespace column is blank, it simply means the resource is being created in the **default namespace**. In Kubernetes, if you don't mention a specific namespace while creating or managing resources, they automatically go into the default namespace.

Command: #kubectl config get-contexts

```
root@control-k8s:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
root@control-k8s:~#

root@control-k8s:~# kubectl get namespaces
NAME              STATUS   AGE
default           Active   9d
grras             Active   9s
kube-node-lease   Active   9d
kube-public       Active   9d
kube-system       Active   9d
root@control-k8s:~#
```

If your namespace column is showing blank, which means he using "default" namespace

- How to set namespace in config file without open config file directly?

Modifies the current context to use a specific namespace by default. This eliminates the need to specify the namespace for every command.

Command: # kubectl config set-context--current--namespace=<new_namespace>

```
root@control-k8s:~# kubectl config set-context --current --namespace=grras
Context "kubernetes-admin@kubernetes" modified.
```

And now check again the context.

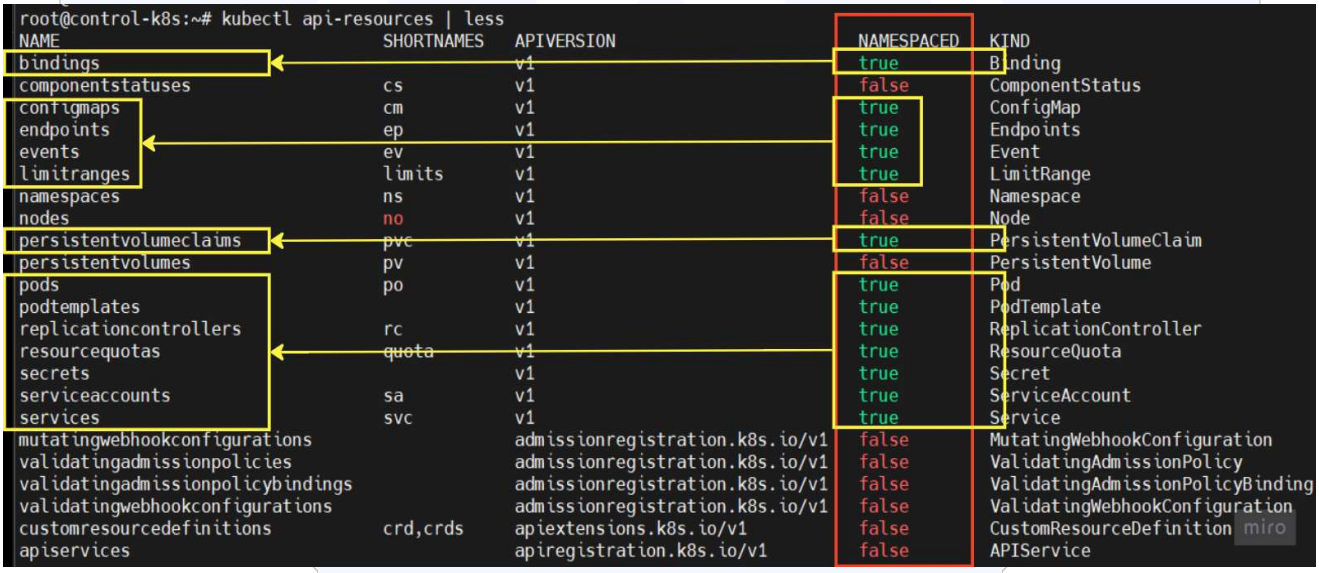
Command: #kubectl config get-contexts

```
root@control-k8s:~# kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin   grras
```

Namespace-Dependent Resources vs. Independent Resources:

Now here we have to understand that, Not all Kubernetes resources are tied to a namespace. To figure out which ones are namespace-dependent:

Command: # kubectl api-resources



¶ Before creating any resource, first check if it is namespace-dependent. If it is, you'll need to either specify the namespace in the command line using -n or--namespace or let it use the default namespace set in your configuration.

For example we have to create a pod in specific namespace. Run this

Command: kubectl run <pod-name>--image=image:tag-n <namespace>

Pod (Smallest Deployable Unit in Kubernetes)

A Pod in Kubernetes is the smallest and most basic unit of deployment. It represents a single instance of a running process in your cluster and can contain one or more containers. All containers within a Pod share the same network IP, storage volume, and namespaces, which means they can communicate with each other easily via localhost networking. Pod is a namespace dependent Kubernetes resources. Either you have to define in command line or it will take bydefault by your config file.

Practice with Pods in Kubernetes:

For practicing Pod, first we will create a Pod and then will manage it. So, we can create a Pod resource from following methods:

Command: Pod creation

- Pod created using CLI method: Syntax

  Command: # kubectl run example-pod--image=nginx--port=80

```
root@control-k8s:~# kubectl run example-pod --image=nginx --port=80
pod/example-pod created
```

  ¶ Note: you can create a manifests file from cli using--dry run=client. The dry-run method validates and generates a YAML manifest from the provided CLI options without creating the actual resource.

  ¶ The--dry-run=client option generates the manifest at the client level without interacting with the API server, while the--dry-run=server option also available that validates the configuration against the API server to ensure it would work if applied.

  ¶ Later, the-o option specifies the output format, such as YAML (-o yaml), JSON (-o json), or other formats. You can redirect this output to a file for future use.

  Command: # kubectl run example-pod--image=nginx--port=80--dry run=client-o yaml > example-pod.yml

- Pod created using File method: Syntax

  Command: # kubectl create-f example-pod.yml

```
root@control-k8s:~# kubectl create -f example-pod.yml
pod/example-pod created
```

- Pod created using manifests file method: Syntax

```yaml
apiVersion: v1                  # Specifies the version of the Kubernetes API to use for the resource (e.g., v1,
kind: Pod                       # Specifies the type of Kubernetes resource (e.g., Pod, Deployment, Service).
metadata:
  name: example-pod             # Name of the resource, must be unique within the namespace.
  labels:
    app: example-app            # Labels for categorizing the resource, can be used for selection and filtering.
spec:                           # This is Pod Specification
  containers:
    - name: example-container   # The name of the container inside the pod.
      image: nginx              # The container image to be used (e.g., nginx, mysql).
      ports:
        - containerPort: 80     # Ports exposed by the container within the pod.
```

Pod Management Commands

- Check the status of running Pod in short:

  Command: # kubectl get po

```
root@control-k8s:~# kubectl get po
NAME          READY   STATUS    RESTARTS   AGE
example-pod   1/1     Running   0          2m28s
```

- Check the status of Pod in detailed (including node name and labels)

  Command: # kubectl get po –show-labels

```
root@control-k8s:~# kubectl get po --show-labels
NAME          READY   STATUS    RESTARTS   AGE     LABELS
example-pod   1/1     Running   0          2m11s   app=example-app
```

  Command: # kubectl get po-o wide

```
root@control-k8s:~# kubectl get po -o wide
NAME          READY   STATUS    RESTARTS   AGE     IP          NODE        NOMINATED NODE   READINESS GATES
example-pod   1/1     Running   0          2m42s   10.40.0.1   node1-k8s   <none>           <none>
root@control-k8s:~#
```

- Delete a Pod using file & command line methods

  Command: # kubectl delete-f example-pod.yml

```
root@control-k8s:~# kubectl delete -f example-pod.yml
pod "example-pod" deleted
root@control-k8s:~#
```

  Command: # kubectl delete pod example-pod

```
root@control-k8s:~# kubectl delete pod example-pod
pod "example-pod" deleted
```

¶ How to practice about Pod resource:

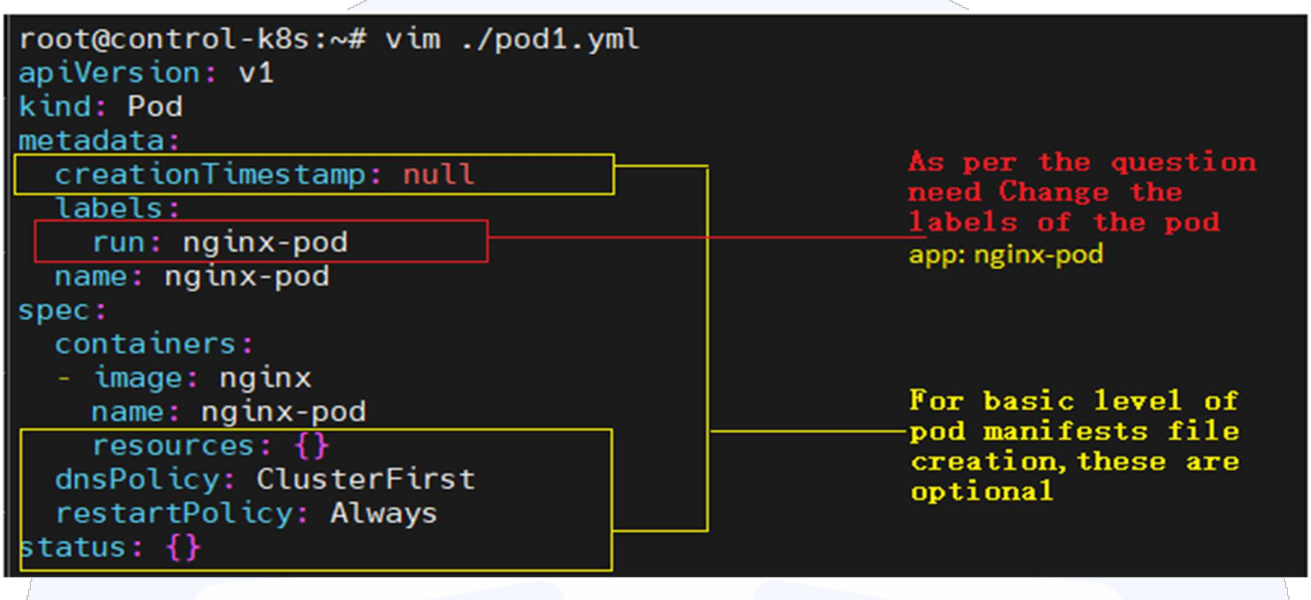
¶ Using the CLI and file method, create a Pod running a NGINX container In custom namespace “grras” with the following specifications:

¶ Name the Pod: nginx-pod, Assign the label: app: nginx-pod, Deploy the Pod using the manifest In custom namespace “grras”

Command: # kubectl run nginx-pod--image=nginx--dry-run=client-o yaml > pod1.yml

```
root@control-k8s:~# kubectl run nginx-pod --image=nginx --dry-run=client -o yaml > pod1.yml
root@control-k8s:~# vim ./pod1.yml
```

Command: # vim ./pod1.yml



- Run this and Verify its creation. In custom namespace “grras”

Command: # kubectl create-f ./pod1.yml

Command: # kubectl get po-o wide--show-labels

```
root@control-k8s:~# kubectl create -f ./pod1.yml
pod/nginx-pod created
root@control-k8s:~# kubectl get po -o wide --show-labels
NAME        READY   STATUS    RESTARTS   AGE   IP          NODE        NOMINATED NODE   READINESS GATES   LABELS
nginx-pod   1/1     Running   0          14s   10.40.0.1   node1-k8s   <none>           <none>            app=nginx-pod
root@control-k8s:~#
```

- Monitor the Pod's status and logs. In custom namespace “grras”

Command: # kubectl logs <pod-name>-c <container-name>

```
root@control-k8s:~# kubectl logs nginx-pod -c nginx-pod
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
```

- Inspecting a Pod

Command: # kubectl describe po <pod-name>

```
root@control-k8s:~# kubectl describe pod nginx-pod
Name:             nginx-pod
Namespace:        grras
Priority:         0
Service Account:  default
Node:             node1-k8s/192.168.199.174
Start Time:       Wed, 08 Jan 2025 08:02:16 +0530
Labels:           app=nginx-pod
Annotations:      <none>
```

- Accessing a Pod's Shell

Command: # kubectl exec-it <pod-name>-- /bin/sh

```
root@control-k8s:~# kubectl exec -it nginx-pod -- /bin/bash
root@nginx-pod:/# ls
bin   dev                  docker-entrypoint.sh  home  lib64  mnt  proc  run   srv  tmp  var
boot  docker-entrypoint.d  etc                   lib   media  opt  root  sbin  sys  usr
```

- After testing, clean up the resources you created. In custom namespace "grras"(Note: No HA feature available)

Command: # kubectl delete pod <pod-name>
Command: # Kubectl delete-f pod-file.yml *##------ Using manifests file method*
Command: # kubectl get po

```
root@control-k8s:~# kubectl delete pod nginx-pod
pod "nginx-pod" deleted
```

```
root@control-k8s:~# kubectl get po
No resources found in grras namespace.
```

What is the limitations of Pod?

Pods are the foundation of Kubernetes the smallest deployable unit—lightweight and simple. However, they are not designed to handle the demands of real-world application deployments independently. Here's why:

No High Availability (HA):

If a Pod crashes or fails for any reason, Kubernetes can restart it, but this does not ensure redundancy. To keep your application always available, you need multiple Pods running and managed together. So if one fails, other can take place of it without any downtime.

No Scaling:

Pods cannot automatically scale up or down to handle future high demand for application services. Scaling requires a system that dynamically adjusts the number of Pods based on the load on the application. However, with Pods alone, this process cannot happen automatically—you have to monitor the load manually and scale Pods yourself. This can be a tedious and error-prone task, often leading to downtime and disruptions in service.

No Persistent Storage:

Data stored in a Pod is ephemeral by default. If the Pod restarts, all data is lost unless persistent storage is configured.

No Rollback or Rollout:

In today's real-world scenarios, having the ability to roll out new versions or roll back to a previous stable state is critical. Pods by themselves lack these features, making them unsuitable for modern, reliable deployments.

After Pod what is next ??? (Replication Controller)

The Replication Controller is like a manager for your Pods. It ensures all Pods with the same labels are grouped under its control and maintains a specified number of replicas at all times. For example, if you want high availability or need to prepare for future demand, you can create multiple Pods with the same configuration. The Replication Controller will take charge of all these Pods that match its labels, ensuring they meet your desired count.

What is Replication Controller?

A Replication Controller in Kubernetes ensures that a specific number of replicas (copies) of a Pod are always running. It constantly monitors Pods and makes adjustments to maintain the desired number of replicas using matching selectors and labels.

Main Functions of Replication Controller (RC)

- Automatic Pod Creation: If a Pod crashes or gets deleted, the Replication Controller automatically creates a replacement. This is possible because the Pod's labels match the selector labels defined in the Replication Controller. All Pods with matching labels are managed under the control of the Replication Controller. It ensures the desired number of Pods, as specified in the replicas section of the resource YAML file, is always maintained. It's like orphan Pods are now being cared for by a parent, as the Replication Controller takes them under its management.

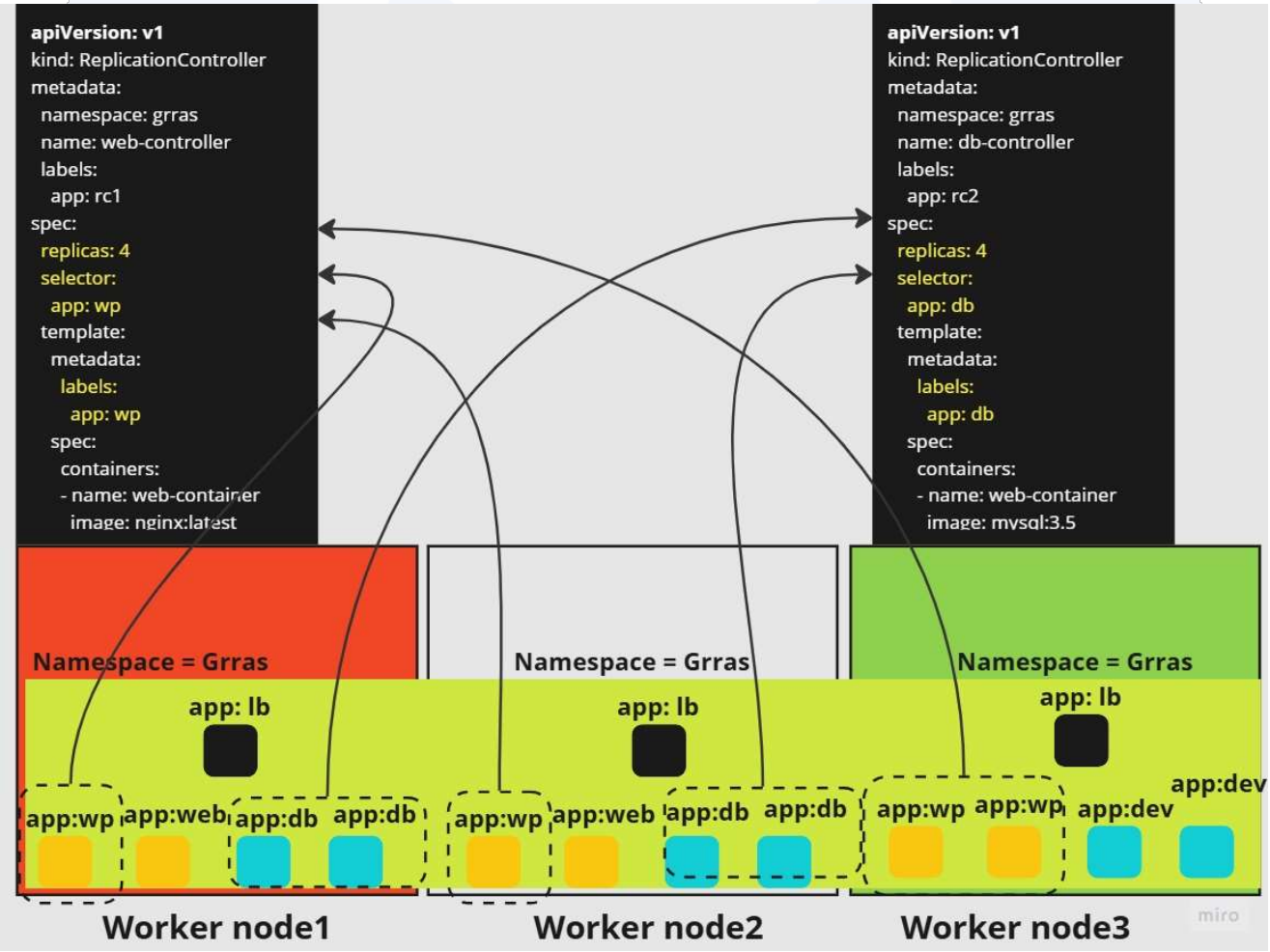
- Pod Removal: If there are more Pods than the desired count, the Replication Controller identifies and removes the unnecessary ones. However, before removing excess Pods, it first takes control of any orphan Pods to bring them under its management.

Why is it Better than Using Pods Alone?

- High Availability: It keeps multiple Pods running to ensure redundancy and fault tolerance.

- Auto-Recovery: If a Pod fails or is accidentally deleted, it recreates it automatically.
- Manual Scaling: You can easily increase or decrease the number of replicas based on demand.

```yaml
apiVersion: v1                  # API version for the resource.
kind: ReplicationController     # Defines the resource type.
metadata:                      # metadata section of ReplicationController.
  name: web-controller          # Name of the ReplicationController.
  labels:                       # Metadata labels for ReplicationController
    env: test          # Label for categorizing the controller.
spec:                  # This is ReplicationController spec.
  replicas: 4          # Requested number of Pod replicas.
  selector:            # Selectors take-over all pods with below labels
    app: wp            # Label to select and manage matching Pods.
  template:            # Pod template used for creating replicas.
    metadata:          # Pod Metadata
      labels:          # Pod labels
        app: wp        # Labels applied to created Pods.
    spec:
      containers:
      - name: web-container  # Container name in the Pod.
        image: nginx:latest  # Container image to use (NGINX).
```



¶ How to practice ReplicationController resource??:

¶ In a Kubernetes cluster, you already have 4 Pods in the grras
¶ namespace with the following labels:
¶ 2 Pods with the label app: web.
¶ 2 Pods with the label app: db.

¶ Now, create a ReplicationController named web-controller with a requested replicas value of 4.

¶ Since 2 Pods with the label app: web already exist, the Kubernetes controller will create only 2 additional Pods to reach the desired total of 4 replicas, as specified in the replicas field of the ReplicationController.

¶ For monitoring purpose. If you will describe the pod this time, you will see they are managed by "web-controller" ReplicationController. Because this time they are not orphan, they have parents.

Answer:

Create the Pods using an online raw YAML file. (The YAML file has been uploaded to my GitHub account. You can download that file from wget)

Command: # kubectl create-f \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rc4pod.yml

```
root@control-k8s:~# kubectl create -f https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rc4pod.yml
pod/pod1 created
pod/pod2 created
pod/pod3 created
pod/pod4 created
root@control-k8s:~#
```

Check now: Pods are being created on random nodes as scheduled by the kube scheduler. You can verify this by describing the Pods to confirm that they are for now not managed by the ReplicationController.

Command: #kubectl get po-o wide--show-labels

```
root@control-k8s:~# kubectl get po -o wide --show-labels
NAME   READY   STATUS    RESTARTS   AGE   IP          NODE        NOMINATED NODE   READINESS GATES   LABELS
pod1   1/1     Running   0          31s   10.38.0.1   node2-k8s   <none>           <none>            app=wp
pod2   1/1     Running   0          31s   10.40.0.1   node1-k8s   <none>           <none>            app=wp
pod3   1/1     Running   0          31s   10.40.0.2   node1-k8s   <none>           <none>            app=db
pod4   1/1     Running   0          30s   10.38.0.2   node2-k8s   <none>           <none>            app=db
root@control-k8s:~#
```

Describe the pod

Command: # kubectl describe po <pod-name>

```
root@control-k8s:~# kubectl describe po pod1  | less
root@control-k8s:~#
root@control-k8s:~# kubectl describe po pod1  | less
Node:             node2-k8s/192.168.199.149
Start Time:       Wed, 08 Jan 2025 17:03:50 +0530
Containers:
  con1:
    Container ID:   containerd://57d16e7b0e1bdaa126ee1a8c8ad0c48d055367995449545385efff939227d05d
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:42e917aaa1b5bb40dd0f6f7f4f857490ac7747d7ef73b391c774a41a8b994f15
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 08 Jan 2025 17:03:59 +0530
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-rcqs2 (ro)
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  10m   default-scheduler  Successfully assigned grras/pod1 to node2-k8s
  Normal  Pulling    10m   kubelet            Pulling image "nginx"
  Normal  Pulled     10m   kubelet            Successfully pulled image "nginx" in 3.274s (3.274s including waiting
  Normal  Created    10m   kubelet            Created container con1
  Normal  Started    10m   kubelet            Started container con1
```

> ¶ Info: In the Pod's logs, at the end, you can see that the kube-scheduler is responsible for selecting the best node for the Pod (We are not choosing any nodes, its kube-scheduler), while the kube-controller ensures that the desired state matches the requested state in any condition, that you gives as replicas=value.

Now create a ReplicationController named web-controller with a requested replicas value of 4 and the label dev=wp. This will allow it to take control of the existing orphan Pods already running in the cluster that match the specified label.

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: web-controller
spec:
  replicas: 4
  selector:
    app: wp # Selector to manage Pods with this label
  template:
    metadata:
      labels:
        app: wp  # Label applied to newly created Pods
    spec:
      containers:
      - name: web-container
        image: nginx:latest
```

All the Pods with this label are now managed by the ReplicationController named web-controller.
And we have two pods with same label

pod1
pod2

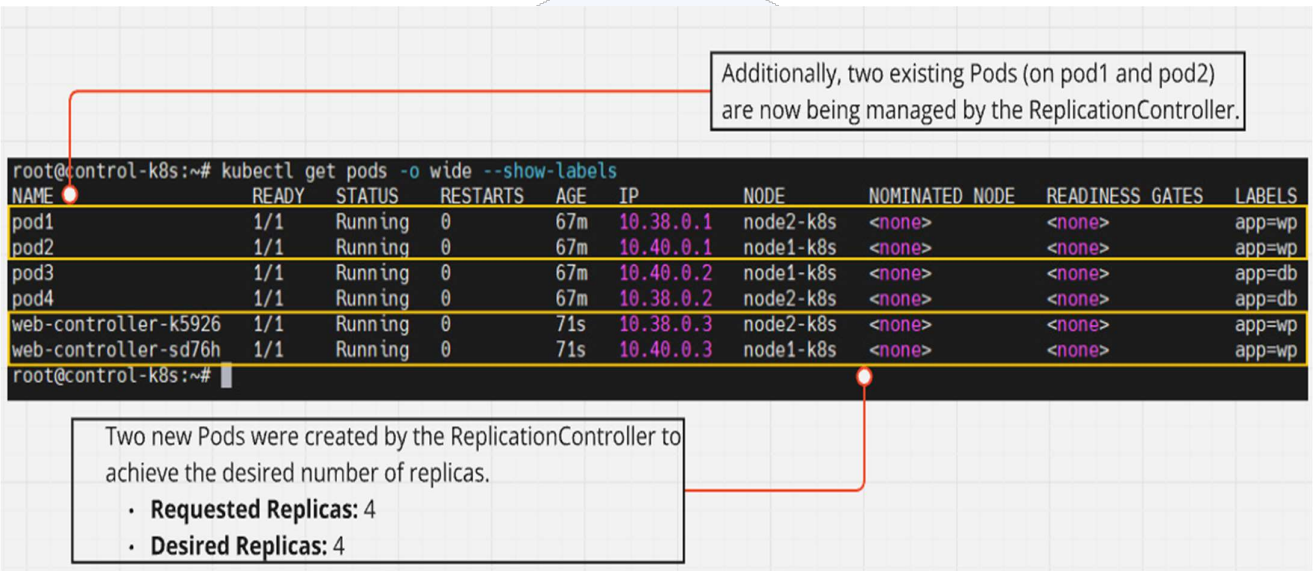
Create the replication-controller and check the logs of pod

Command: # kubectl create-f \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rc1.yml

```
root@control-k8s:~# kubectl create -f https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rc1.yml
replicationcontroller/web-controller created
root@control-k8s:~#
```

This time Describe the pods and check the logs of pod

```
root@control-k8s:~# kubectl get rc -n grras -o wide
NAME             DESIRED   CURRENT   READY   AGE   CONTAINERS      IMAGES          SELECTOR
web-controller   4         4         4       35s   web-container   nginx:latest    app=wp
root@control-k8s:~#
```



Created “rc1.yml” are managed by “ReplicationController”

```
root@control-k8s:~# kubectl describe pod  pod1 pod2 | grep -i "Controlled By:"
Controlled By:  ReplicationController/web-controller
Controlled By:  ReplicationController/web-controller
```

¶ Note: If you want to scale up or scale down the number of Pod replicas, the kube-controller-manager will manage the desired state, while the kube-scheduler will schedule Pods accordingly.

¶ If you increase the replicas, new Pods with the label app:wp will be created. If you decrease the replicas, extra Pods will be removed, and the system will ensure the desired state matches the requested state. This is the process behind scaling in Kubernetes.

Now Scale up the replicas to 10

Command: # kubectl scale rc web-controller--replicas=10

```
root@control-k8s:~# kubectl scale rc web-controller --replicas=10
replicationcontroller/web-controller scaled
root@control-k8s:~# kubectl get rc -o wide
NAME             DESIRED   CURRENT   READY   AGE   CONTAINERS      IMAGES          SELECTOR
web-controller   10        10        10      23m   web-container   nginx:latest    app=wp
root@control-k8s:~#
```

```
root@control-k8s:~# kubectl get po -o wide --show-labels
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED NODE   READINESS GATES   LABELS
pod1                   1/1     Running   0          91m   10.38.0.1    node2-k8s   <none>           <none>            app=wp
pod2                   1/1     Running   0          91m   10.40.0.1    node1-k8s   <none>           <none>            app=wp
pod3                   1/1     Running   0          91m   10.40.0.2    node1-k8s   <none>           <none>            app=db
pod4                   1/1     Running   0          90m   10.38.0.2    node2-k8s   <none>           <none>            app=db
web-controller-2qcpv   1/1     Running   0          83s   10.38.0.5    node2-k8s   <none>           <none>            app=wp
web-controller-dkrks   1/1     Running   0          83s   10.40.0.6    node1-k8s   <none>           <none>            app=wp
web-controller-gmnx6   1/1     Running   0          83s   10.40.0.5    node1-k8s   <none>           <none>            app=wp
web-controller-gwg92   1/1     Running   0          83s   10.38.0.6    node2-k8s   <none>           <none>            app=wp
web-controller-k5926   1/1     Running   0          24m   10.38.0.3    node2-k8s   <none>           <none>            app=wp
web-controller-mpd99   1/1     Running   0          83s   10.38.0.4    node2-k8s   <none>           <none>            app=wp
web-controller-rh46n   1/1     Running   0          83s   10.40.0.4    node1-k8s   <none>           <none>            app=wp
web-controller-sd76h   1/1     Running   0          24m   10.40.0.3    node1-k8s   <none>           <none>            app=wp
root@control-k8s:~#
```

Now Scale down the replicas to 3

Command: # kubectl scale rc web-controller--replicas=3

```
root@control-k8s:~# kubectl scale rc web-controller --replicas=3
replicationcontroller/web-controller scaled
root@control-k8s:~# kubectl get rc -o wide
NAME             DESIRED   CURRENT   READY   AGE   CONTAINERS      IMAGES          SELECTOR
web-controller   3         3         3       25m   web-container   nginx:latest    app=wp
root@control-k8s:~#
```

```
root@control-k8s:~# kubectl get po -o wide --show-labels
NAME                   READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED NODE   READINESS GATES   LABELS
pod1                   1/1     Running   0          91m   10.38.0.1    node2-k8s   <none>           <none>            app=wp
pod2                   1/1     Running   0          91m   10.40.0.1    node1-k8s   <none>           <none>            app=wp
pod3                   1/1     Running   0          91m   10.40.0.2    node1-k8s   <none>           <none>            app=db
pod4                   1/1     Running   0          91m   10.38.0.2    node2-k8s   <none>           <none>            app=db
web-controller-k5926   1/1     Running   0          25m   10.38.0.3    node2-k8s   <none>           <none>            app=wp
root@control-k8s:~#
```

¶ When you delete some Pods with matching labels, and the number falls below the requested replicas, the ReplicationController will automatically create the exact number of Pods needed to restore the requested = desired number of replicas. This is managed by the kube-scheduler and the controller manager.

Delete some pods managed by ReplicationController (RC)

Command: # kubectl scale rc web-controller--replicas=3

```
root@control-k8s:~# kubectl delete pod pod1 pod2 pod3 pod4
pod "pod1" deleted
pod "pod2" deleted
pod "pod3" deleted
pod "pod4" deleted
```

New Pods are immediately spun up by the ReplicationController (RC) because it is managing Pods with the label app=wp. This demonstrates a High Availability feature, which standalone Pods cannot provide.

Command: # kubectl get po--show-labels

```
root@control-k8s:~# kubectl get  po --show-labels
NAME                   READY   STATUS    RESTARTS   AGE    LABELS
web-controller-5pt4g   1/1     Running   0          16m    app=wp
web-controller-k5926   1/1     Running   0          117m   app=wp
web-controller-w76bh   1/1     Running   0          16m    app=wp
```

```
root@control-k8s:~# kubectl delete -f https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rc1.yml
replicationcontroller "web-controller" deleted
```

What are the limitations of ReplicationController (RC)?

ReplicationController (RC) has some limitations:

- It uses simple **matchLabels** selectors to manage Pods, which can only match **a single key=value pair**, making it less flexible.
- It doesn't support **complex label selection or expressions** for managing Pods.
- It lacks advanced features like **rolling updates and rollbacks,** requiring manual intervention for changes.

Then what is the option?---------→ ReplicaSet is the solution!

Why is ReplicaSet (RS) better?

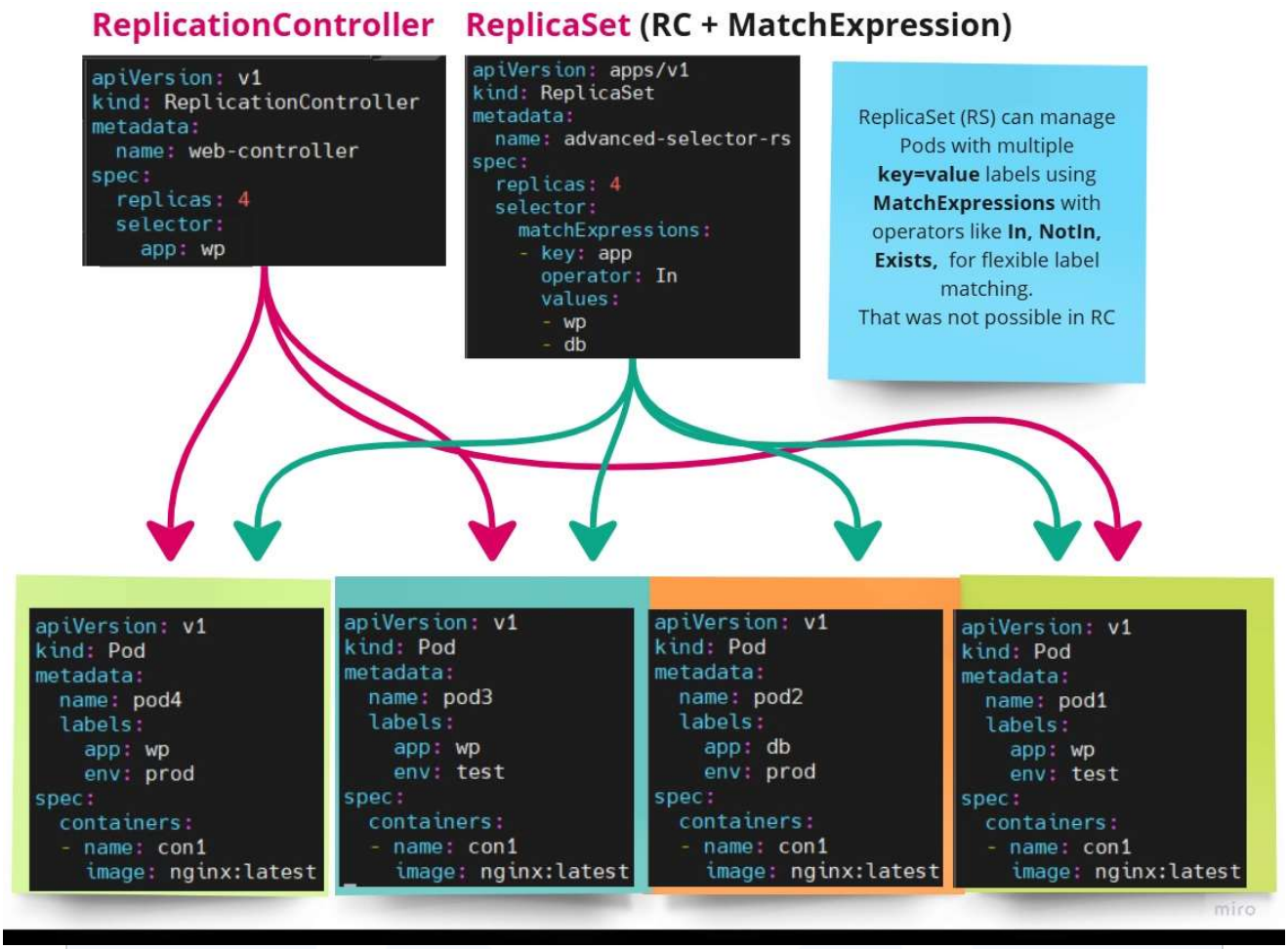
ReplicaSet is an improved version of RC with more flexibility:

- It supports both **matchLabels** and **matchExpressions,** allowing it to manage Pods using multiple conditions, such as matching multiple key-value pairs or using operators like **In, NotIn, or Exists**.
- This means RS can manage more complex scenarios, like selecting Pods with labels **env=prod** and **app=web** or even Pods that have a label tier with any value.
- It integrates seamlessly with Deployments, enabling rolling updates, rollbacks, and easier scaling.

¶ Lets understand the problem!

¶ Create four Pods with different labels like app=wp, env=test, and app=db, env=prod. A ReplicationController (RC) can only manage Pods with one specific label using matchLabels, and it doesn't support features like rolling updates or rollbacks.

¶ On the other hand, a ReplicaSet (RS) can manage Pods with multiple labels using matchExpressions, making it more flexible and better for advanced scenarios.



Create four pods and check the labels

Command: # kubectl create-f \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rs4pod.yml

```
root@control-k8s:~# kubectl create -f  \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rs4pod.yml
pod/pod1 created
pod/pod2 created
pod/pod3 created
pod/pod4 created
```

```
root@control-k8s:~# kubectl get po --show-labels -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP           NODE        NOMINATED NODE   READINESS GATES   LABELS
pod1   1/1     Running   0          49s   10.38.0.1    node2-k8s   <none>           <none>            app=wp,env=test
pod2   1/1     Running   0          49s   10.40.0.2    node1-k8s   <none>           <none>            app=db,env=prod
pod3   1/1     Running   0          48s   10.38.0.2    node2-k8s   <none>           <none>            app=wp,env=test
pod4   1/1     Running   0          48s   10.40.0.1    node1-k8s   <none>           <none>            app=wp,env=prod
root@control-k8s:~#
```

Now Run ReplicaSet from cloud manifest RS yaml file.

Command: # kubectl create-f \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rs1.yml

```
root@control-k8s:~# kubectl create -f  \
https://raw.githubusercontent.com/rakesh08061994/k8s-practice/refs/heads/master/rs1.yml
replicaset.apps/advanced-selector-rs created
root@control-k8s:~#
```

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: advanced-selector-rs
spec:
  replicas: 4 #Requested pods number
  selector:
    matchExpressions:
    - key: app  #This is the key name
      operator: In  #In Operator means that, following matching labels is now being managed by RS
      values:
      - wp   #This is value1
      - db   #This is value2
  template:
    metadata:
      labels:
        app: wp #Later scalling pods will create with this labels only
    spec:
      containers:
      - name: con1
        image: nginx:latest
```

Now check which pods is being managed by this ReplicaSet(RS)

Command: # kubectl describe rs advanced-selector-rs

```
root@control-k8s:~# kubectl describe rs advanced-selector-rs
Name:         advanced-selector-rs
Namespace:    grras
Selector:     app in (db,wp)
Labels:       <none>
Annotations:  <none>
Replicas:     4 current / 4 desired
Pods Status:  4 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=wp
  Containers:
```

Command: # kubectl describe <pod1> … <pod..n>

```
root@control-k8s:~# kubectl describe pod pod1 pod2 pod3 pod4 | grep -i "Controlled"
Controlled By:  ReplicaSet/advanced-selector-rs
Controlled By:  ReplicaSet/advanced-selector-rs
Controlled By:  ReplicaSet/advanced-selector-rs
Controlled By:  ReplicaSet/advanced-selector-rs
```

What is “In, NotIn, Exists” label Operators in ReplicaSet ?

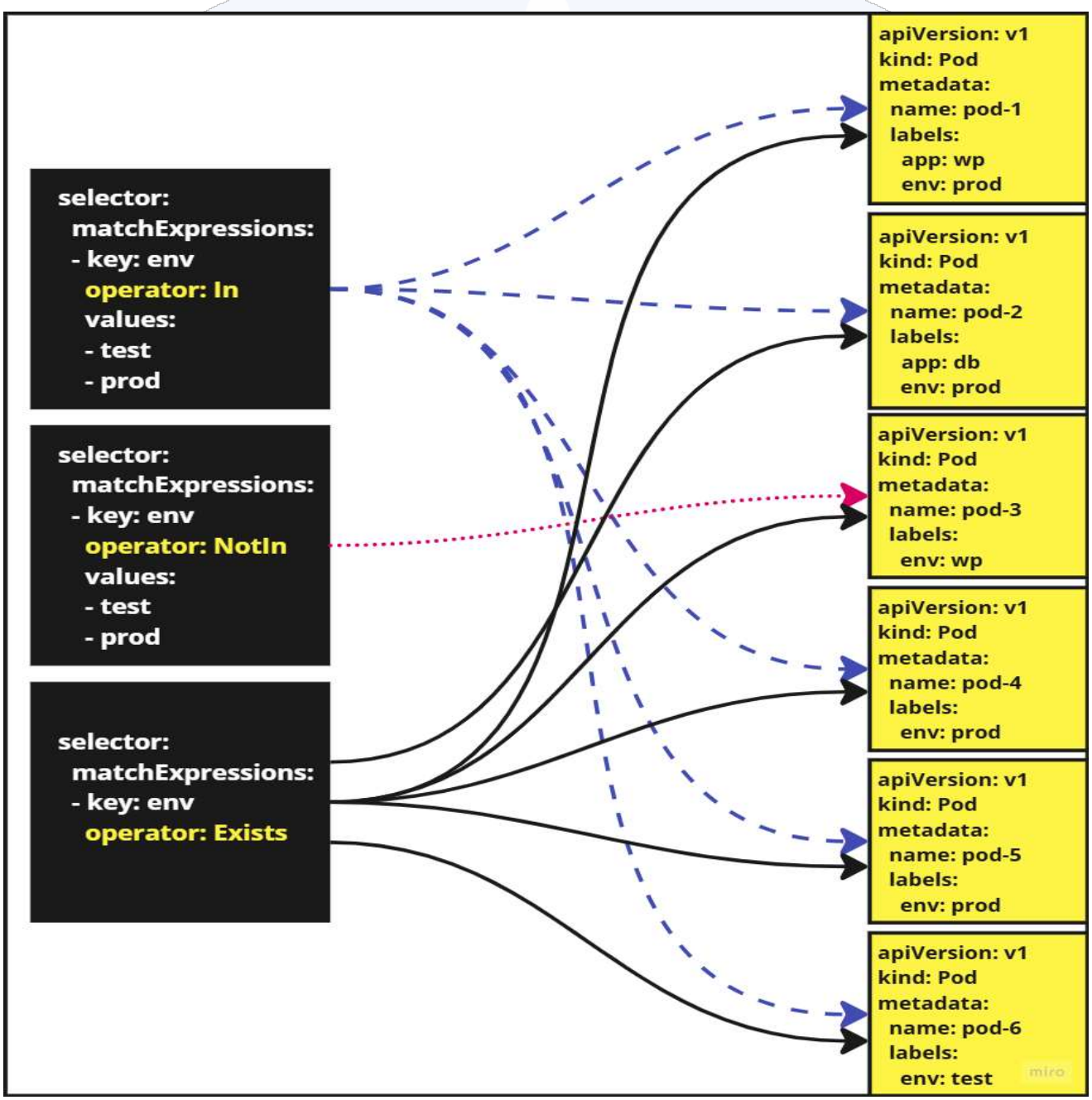
We know that instead of the single key:value matching capability in ReplicationController, ReplicaSet offers much more flexibility with operators and MatchExpressions, where each operator provides unique functionality.

So, if I were to explain the In, NotIn, and Exists operators in ReplicaSet, I would say:

- In Operator: Used to match Pods where the label key has a value that exists in a specified list of values. For example, key: In [value1, value2] selects Pods where the label key is either value1 or value2.
- NotIn Operator: Matches Pods where the label key does not have a value in the specified list. For example, key: NotIn [value1, value2] excludes Pods with those values.
- Exists Operator: Matches Pods that have a specific key, regardless of the value. For example, key: Exists selects Pods that have the key defined in their labels.

These operators make ReplicaSet more versatile and capable of managing Pods with complex label-based conditions

Graphical Representations



- Using In Operator: Matches Pods where env is either test or prod.

- **Using NotIn Operator:** Matches Except Pods where env is test & prod.
- **Using Exists Operator**: Matches all Pods where the app key exists, regardless of its value. No Value section is needed

---

Important: Avoid Operator Selectors Overlaps in ReplicaSets: Don't Let Them Fight Over Pods!

If you run multiple ReplicaSets with overlapping operator selectors (like **In, NotIn,** and **Exists**) that match the same Pods, they will fight over those Pods because Kubernetes doesn't know which ReplicaSet should "own" them. Here's what happens:

1. **Conflict:**
   Each ReplicaSet will try to manage the Pods to match its desired replica count. For example:
   - One ReplicaSet might create a Pod because it thinks a Pod is missing.
   - Another ReplicaSet might delete the same Pod because it thinks there are too many Pods.
2. **Unpredictable Results:**
   You can't be sure which ReplicaSet will actually manage the Pods. Some Pods might end up managed by both ReplicaSets, which is confusing.
3. **Troubleshooting is Hard:**
   If you're trying to understand why Pods are being created or deleted, it will be difficult because multiple ReplicaSets are involved, and their actions overlap.

How to Avoid This Problem

- Use unique selectors: Make sure each ReplicaSet targets specific Pods with non-overlapping labels or matchExpressions.
- Plan labels carefully: For example:
  - Use labels like app=frontend for one ReplicaSet.
  - Use labels like app=backend for another ReplicaSet.
  - This way, they won't target the same Pods.
- Check before deploying: Always double-check your selectors and labels to avoid overlaps.

Limitations of ReplicaSet (RS) in Simple Terms:

1. **No Built-in Updates**: If you need to update your Pods (like changing the image version), ReplicaSet doesn't do it for you. You'd have to delete and recreate it or manage updates manually.
2. **No Rollbacks**: If something goes wrong with your changes, there's no easy way to go back to a previous working version.
3. **Basic Purpose Only**: ReplicaSet's job is simple—just keep the right number of Pods running. It doesn't handle things like updates or more advanced app management

Resource Name	High Availability	Replica Control	Single Key-Value Pair Labels	Multi Key-Value Pair Labels	Rolling Update	Rollback
Pod	✗ No	✗ No	✓ Yes	✗ No	✗ No	✗ No
ReplicationController	✓ Yes	✓ Yes	✓ Yes	✗ No	✗ No	✗ No
ReplicaSet	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✗ No	✗ No
Deployment	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✓ Yes

What is rolling update and rollback feature?

What is a Rolling Update?

A rolling update is a way to update your application without stopping it entirely. Instead of shutting everything down and applying the new version, Kubernetes updates your application in small steps, one piece at a time.

Imagine you're running an app with 5 servers (Pods):

1. During a rolling update, Kubernetes takes one server (Pod) offline, updates it to the new version, and then puts it back online with proper networking and high availability management without downtime because your pods may run or distribute on multiple nodes.
2. Once the updated server is working perfectly, it moves to the next one.
3. This process continues until all servers are updated.

Why Rolling Update useful:

Your app stays up and running the whole time. Users won't notice downtime because most of the servers are still working while the update happens.

What is a Rollback?

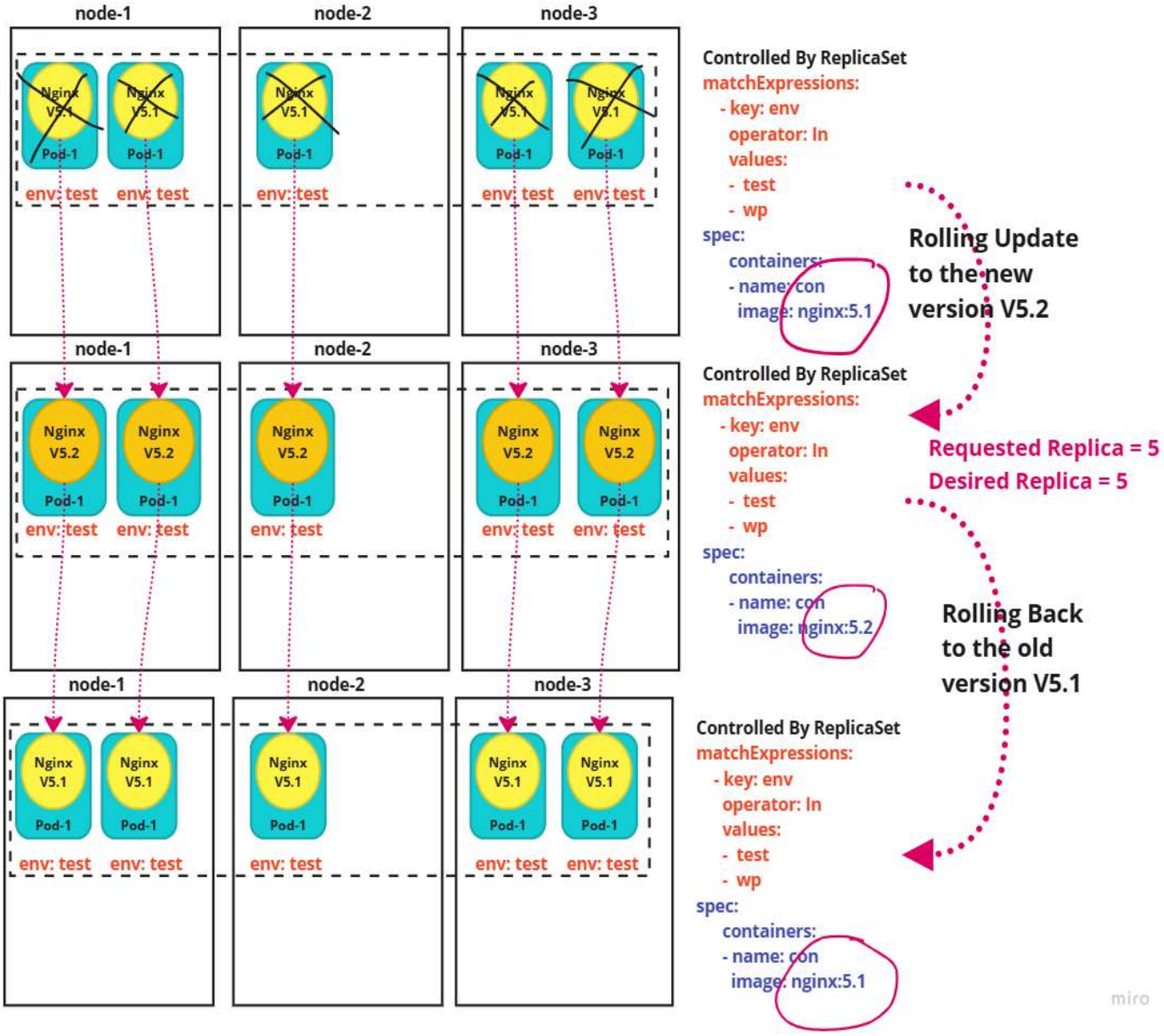
A rollback is a safety net for when something goes wrong during or after an update. It lets you quickly go back to the previous version of your app that was working fine.

Here's an example:

1. You update your app to a new version using a rolling update with proper networking and high availability management without downtime because your pods may run or distribute on multiple nodes.
2. After the update, users start reporting problems (maybe a bug in the new version).
3. Kubernetes allows you to "roll back" to the old version automatically, so your app works again without needing manual fixes.

Why Rollback useful:

Mistakes happen, and rollbacks let you recover quickly without major disruptions.



So, what's the solution?

The solution is the Deployment resource in Kubernetes. As you can see in the diagram, Deployment is the most mature, reliable, and ideal choice for deploying applications in an organization.

It comes with some great features like:

- **High Availability:** Ensures your app is always available.
- **Replica Control:** Automatically manages the number of Pods running.
- **Single and Multi Key-Value Pair Labels:** Lets you easily manage Pods based on labels.
- **Rolling Update:** Updates your application one step at a time, without downtime.
- **Rollback:** If something goes wrong, you can quickly revert to a previous version.

This makes Deployment the go-to resource for managing applications, especially in production environments. It's all about making updates smooth and reliable with minimal effort!

Resource Name	High Availability	Replica Control	Single Key-Value Pair Labels	Multi Key-Value Pair Labels	Rolling Update	Rollback
Pod	✗ No	✗ No	✓ Yes	✗ No	✗ No	✗ No
ReplicationController	✓ Yes	✓ Yes	✓ Yes	✗ No	✗ No	✗ No
ReplicaSet	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✗ No	✗ No
Deployment	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✓ Yes

Deployments = (ReplicaSet + Rolling Update + Roll Back)

A Deployment resource in Kubernetes is a powerful tool for managing and updating your application consistently over time. It ensures that your app runs efficiently by handling rolling updates, scaling to meet demand, and maintaining high availability, so your application stays up and running without interruptions.

This resource type is more advanced than previous container and pod management resources like ReplicationController and ReplicaSet, offering additional features like automated rollbacks, better control over updates, and seamless scaling.

Key Features of a Deployment:

- Container Orchestration: Kubernetes handles the management of containers within Pods, making sure they run efficiently, stay healthy, and remain consistent. It automatically restarts containers if they crash, ensuring minimal disruption to your app.
- Enhanced Microservice Architecture: Kubernetes is ideal for deploying and managing microservices. It allows you to break down an application into smaller, independent components (microservices), making it easier to scale and update parts of the app without affecting the whole system.

- Auto Scaling / High Availability: Kubernetes automatically adjusts the number of Pods (containers) running based on the workload. This means your app will scale up when there's more demand and scale down when traffic is lower, ensuring there is no downtime and the app is always available. By default, you can manually scale the number of Pods in a Deployment by running This allows you to change the number of Pods based on your needs, but it doesn't happen automatically.

  But in this we have to know that To achieve **automatic scaling** based on workload, you need to configure the **Horizontal Pod Autoscaler (HPA)**. HPA automatically adjusts the number of Pods in your Deployment based on metrics such as CPU utilization, memory usage, or custom metrics. This means that when your application experiences higher traffic or resource demand, HPA will automatically scale up the number of Pods, and when the load decreases, it will scale down the Pods.

- Load Balancing: When multiple Pods are running, Kubernetes evenly distributes incoming traffic among them. This helps to prevent any single Pod from being overloaded, improving the overall reliability and performance of your application.

- Replica Control: Kubernetes ensures that the right number of Pods (based on the desired state you set) are running at all times. If a Pod crashes, Kubernetes automatically creates a new one to replace it, ensuring that your app always has the correct number of replicas.

- Automated Rolling Updates: With a Deployment, updates are done gradually, one Pod at a time. This ensures that your application remains available while the update is being applied, minimizing the risk of downtime during updates. The update behavior is controlled using two key settings: **MaxSurge** and **MaxUnavailable**.

    - MaxSurge determines how many Pods can be created above the desired number of Pods during an update (i.e., the extra Pods that can be temporarily added). You can specify this as either a fixed number (e.g., 2) or a percentage (e.g., 25%).

    - MaxUnavailable specifies how many Pods can be unavailable during the update process. Similar to **MaxSurge**, you can define this in either a fixed number (e.g., 1) or as a percentage (e.g., 50%).

- These settings give you fine-grained control over the update process, allowing Kubernetes to manage the update in a way that keeps your application running smoothly without downtime.

- Automated Rollback: If something goes wrong during an update (like bugs or unexpected issues), Kubernetes allows you to automatically roll back to the previous, stable version of your app. This ensures that any potential issues don't disrupt your users. Just like with rolling updates, the rollback also follows the MaxSurge and MaxUnavailable settings to minimize downtime.

- Label Management: Labels are used to organize and select Pods based on specific criteria (such as app version or environment). This helps Kubernetes manage Pods more efficiently, as you can target specific Pods for scaling, updating, or other operations based on their labels.

¶ In reality, you can think of Deployment as an upgraded version of ReplicaSet with RollingUpdate and Rollback features.:

¶ When you run a Deployment, behind the scenes, Kubernetes actually creates a ReplicaSet. The ReplicaSet is responsible for managing the Pods, but there's an important difference:

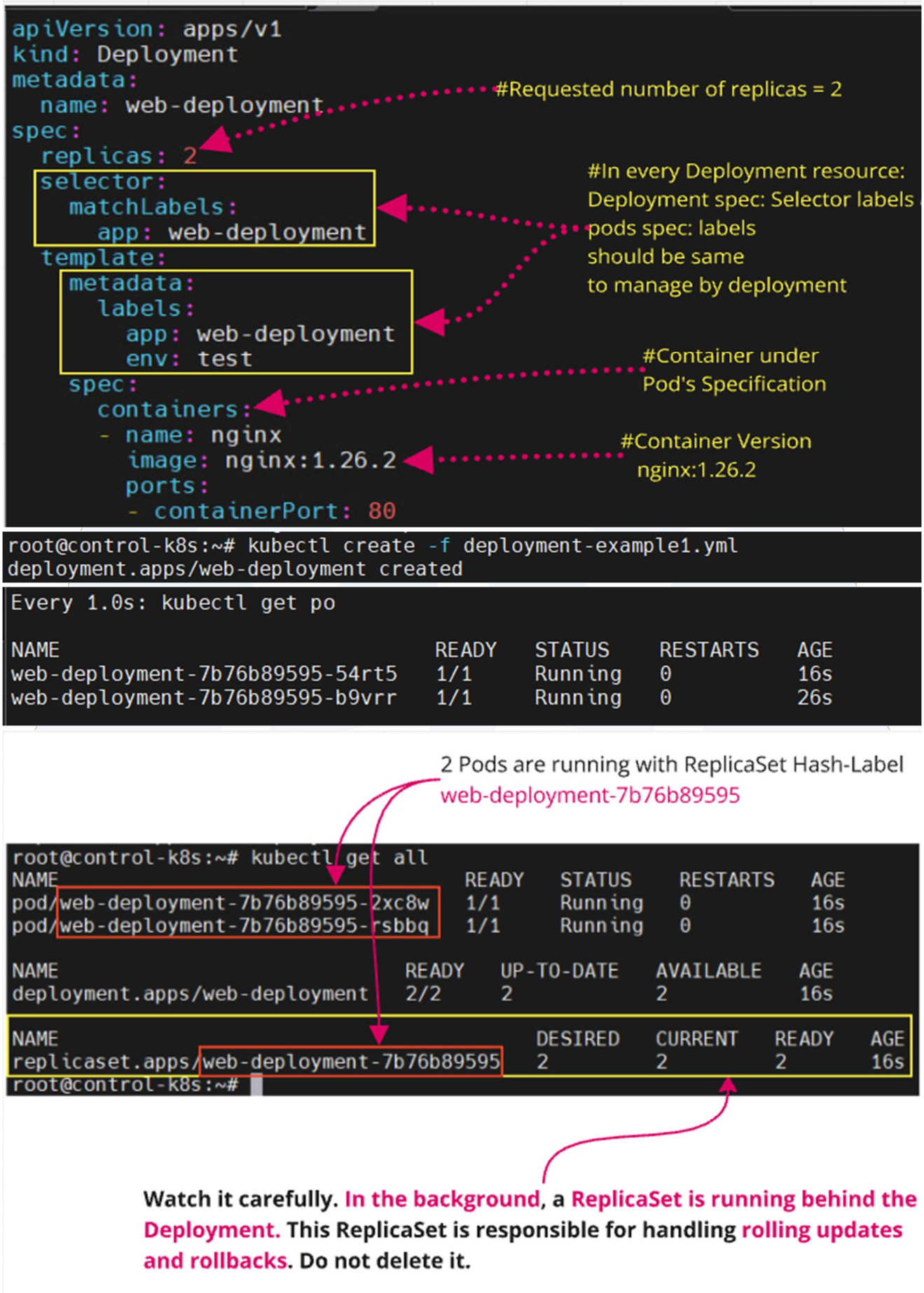
¶ The Deployment adds a hash value to each ReplicaSet. This hash helps manage the Pods and their updates by making sure it doesn't get mixed up with older Pods (orphan Pods). Even if the labels (key-value pairs) match, the hash ensures that old Pods are not included, and only the ones managed by the current ReplicaSet are kept.

¶ This feature in Deployment is more reliable and intelligent it prevents the problem of orphan Pods and manages the update process smoothly.

How to Practice this??

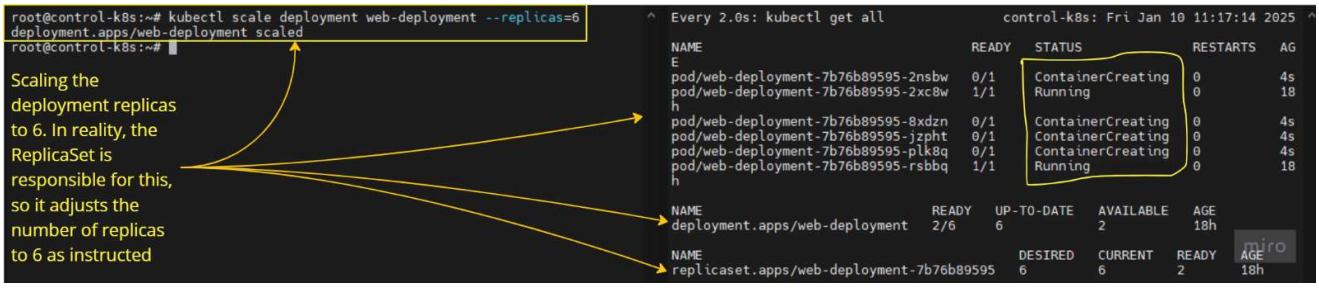
Question: You are managing a Kubernetes deployment and need to perform a series of actions for a web application. The steps are as follows:

1. Create a Deployment:
    - Name the deployment "web-deployment".
    - Use the image version nginx:1.26.2 with 2 replicas.
    - Label the Pods with env: test.



2. Scale Up the Deployment:
    - Using CLI method Increase the number of replicas to 6.

Command: #kubectl scale deployment--replicas=6



¶ In above image when you scale a Deployment in Kubernetes (e.g., increasing replicas to 6), the Deployment doesn't directly manage the Pods. Instead, it delegates this task to the underlying ReplicaSet associated with the Deployment. The ReplicaSet ensures that the desired number of replicas (in this case, 6) is maintained by creating or terminating Pods as needed.

- Using the Manifest File: Update the replicas count directly in the YAML file and reapply it by "kubectl apply -f ". Because you have to apply these change to the actual cluster. Only change the YAML is not sufficient.

```
root@control-k8s:~# vim deployment-example1.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 6
  selector:
    matchLabels:
      app: web-deployment
```

```
root@control-k8s:~# kubectl apply -f deployment-example1.yml
deployment.apps/web-deployment configured
root@control-k8s:~#
```

- Using the Edit ETCD Database method: Modify the Deployment information in the etcd database (not recommended in production).

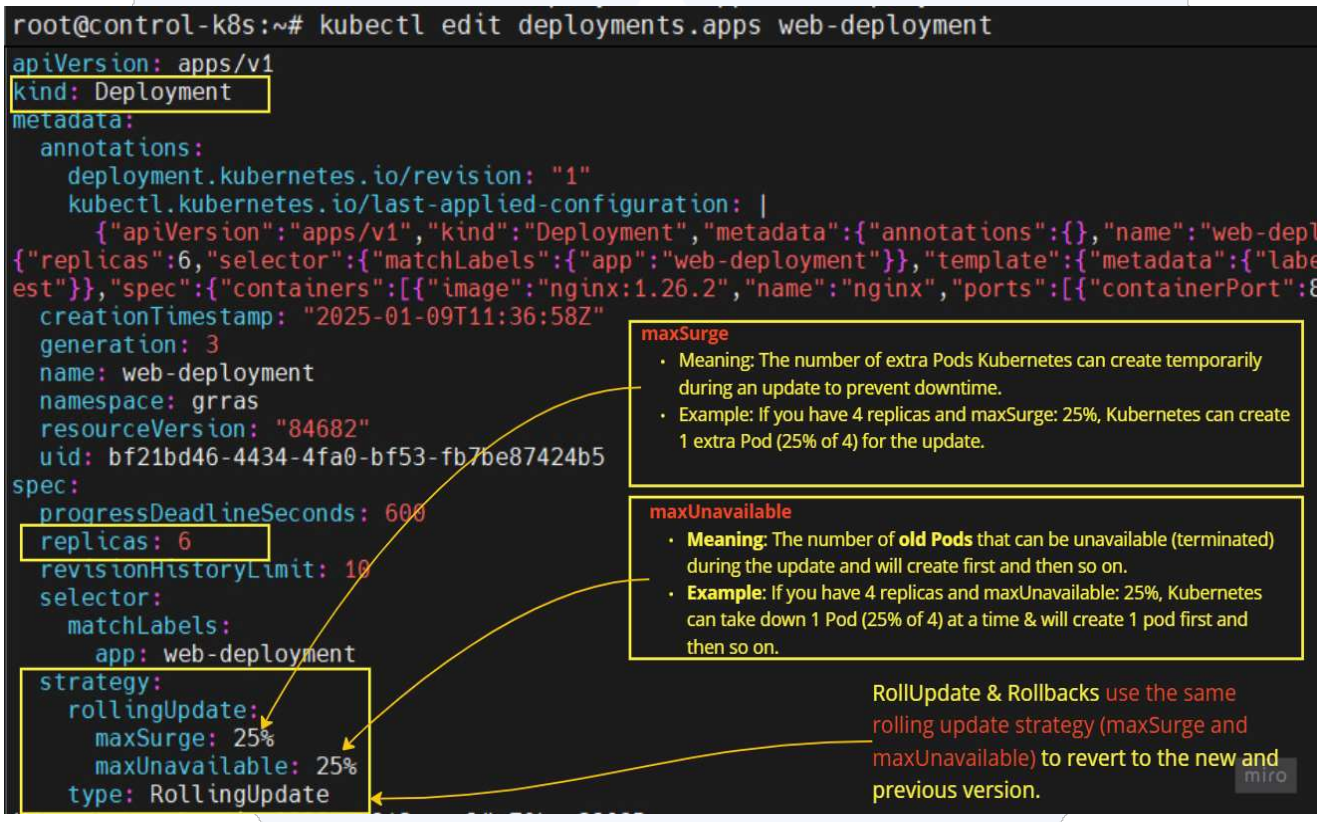
¶ Info: When you're updating or rolling back a Deployment in Kubernetes, you have two important settings: maxSurge and maxUnavailable.

¶ maxSurge: This tells Kubernetes how many extra Pods it can create temporarily during an update or rollback. For example, if you have 4 replicas and set maxSurge to 25% in deployment, then it will allow 1 extra Pod to be created to prevent downtime. Because 25% of 4 is 1. This helps prevent downtime because newly created Pods need time to be ready, join the network, and be managed by services before the old Pods are terminated. So this setting is like all pods make available in every condition.

- maxUnavailable: This tells Kubernetes how many Pods can be taken down (or become unavailable) during the update or rollback. If you set maxUnavailable to 25% for 4 replicas, Kubernetes can make 1 Pod unavailable at a time.

- Both of these settings are part of the RollingUpdate strategy, which Kubernetes uses to manage both updates to a new version and rollbacks to a previous version.

- So, whether you're rolling-updating or rolling-back, Kubernetes uses the RollingUpdate strategy along with maxSurge and maxUnavailable to gradually change the Pods, keeping your app available without downtime.



3. Rolling Update the image version to nginx:1.27.3. (One more RepliaSet with new version)

So you can do this using any of the three methods: command line, editing the manifests file, or directly editing the deployment to the etcd.

In reality When you perform a rolling update on a Deployment with a newer version, actually the Deployment **creates a new ReplicaSet in the background** with the updated version nginx:1.27.3. As a result, the Deployment now manages two ReplicaSets: one with the **previous version 1.27.2** and another with the new **version 1.27.3**. This ensures a smooth transition between versions while maintaining application availability. Later, if

needed, you can roll back to the previous version using the rollback feature. At that time, the existing ReplicaSets created during the update process will be utilized to revert to the stable version.

Command: # kubectl set image deployment/web-deployment nginx=nginx:1.27.3

4. Rollback the Deployment: (Revert Back to the older Replicaset version)

   A rollback in Kubernetes allows you to revert your Deployment to a previous state if something goes wrong with an update. When you perform a rollback, Kubernetes will utilize the ReplicaSet associated with the previous version of your application to restore its state.

   This ensures that the transition back to the older version is seamless, without the need to recreate Pods manually. Rollbacks are especially useful when a new update introduces bugs or unexpected behavior, allowing you to quickly return to a stable version while investigating the issue

   Now, you can choose what you want to do:

   A. Go to the most recent previous version:
      command: # kubectl rollout undo deployment/web-deployment

   B. Go to a specific revision: (First, check the rollout history to identify the revision you need)
      Command: # kubectl rollout history deployment/web-deployment

```
root@control-k8s:~# kubectl rollout history deployment/web-deployment
deployment.apps/web-deployment
REVISION  CHANGE-CAUSE
4         <none>
5         <none>

root@control-k8s:~#
```

NAME of ReplicaSets

replicaset.apps/web-deployment-66f498c855

replicaset.apps/web-deployment-7b76b89595

C. Then, rollback to a specific revision by specifying its number:

Command: # kubectl rollout undo deployment/web-deployment--to revision=<revision-number>

```
root@control-k8s:~# kubectl rollout undo deployment web-deployment --to-revision=4
deployment.apps/web-deployment rolled back
root@control-k8s:~#
```

5. Scale Down the Deployment:
   - Scale down the deployment to 3 replicas using file method.

```
root@control-k8s:~# vim deployment-example1.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-deployment
```

```
root@control-k8s:~# kubectl apply -f deployment-example1.yml
deployment.apps/web-deployment configured
root@control-k8s:~#
```

```
Every 2.0s: kubectl get all                control-k8s: Fri Jan 10 13:43:31 2025

NAME                                 READY   STATUS    RESTARTS   AGE
pod/web-deployment-7b76b89595-kd8l4  1/1     Running   0          29s
pod/web-deployment-7b76b89595-n89l9  1/1     Running   0          25s
pod/web-deployment-7b76b89595-wh58j  1/1     Running   0          23s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web-deployment       3/3     3            3           20h

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/web-deployment-66f498c855  0         0         0       69m
replicaset.apps/web-deployment-7b76b89595  3         3         3       20h
```

Daemon Set

In the series of Kubernetes deployment strategies, let's talk about **DaemonSet**, a resource type with its own specific use cases. Unlike Deployments, DaemonSets are not used for replicas, high availability, rolling updates, or rollback features—Deployments are better suited for those scenarios.

So why is a DaemonSet important in deployment strategies?

Real-World Use Case of DaemonSets:

DaemonSets are used when you need unique Pods that run **only once per node** in a Kubernetes cluster—no more, no less. This is particularly useful for tasks like:

- Monitoring nodes (e.g., collecting logs or metrics).
- Managing network configuration for nodes (e.g., running network proxies).

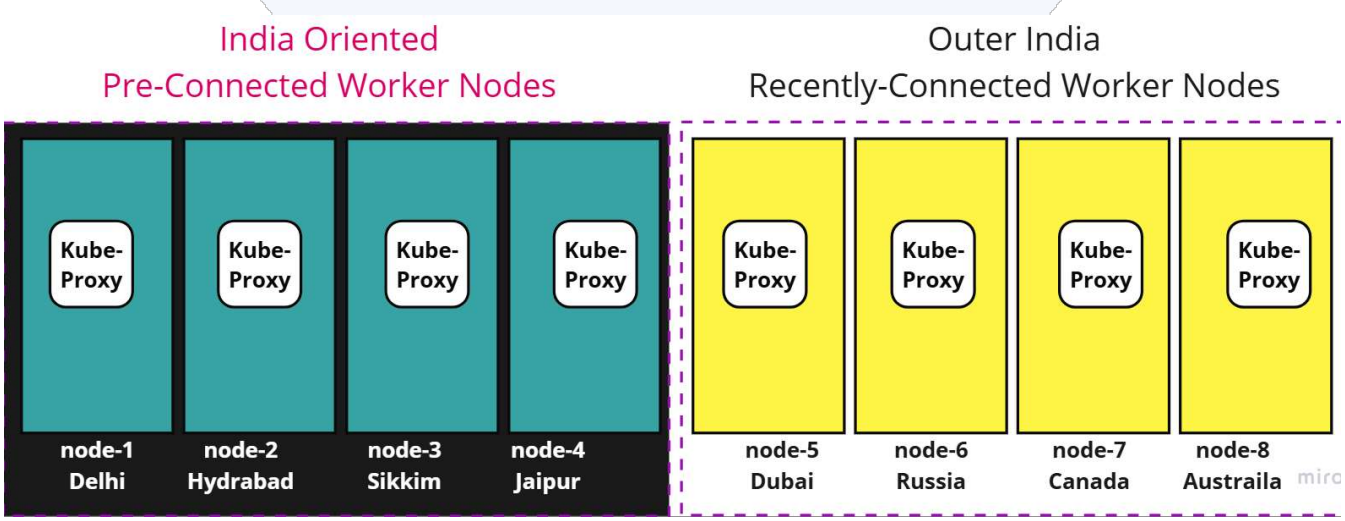
DaemonSets ensure that these essential Pods are automatically scheduled on **every node** in the cluster. When new nodes are added to the cluster, the DaemonSet automatically ensures that the required Pods are deployed on those nodes as well.

Example: For instance, Kubernetes often runs some DaemonSets in the kube-system namespace by default. These DaemonSets help manage critical components of the cluster, such as logging or networking services.

```
root@control-k8s:~# kubectl get daemonsets.apps -n kube-system
NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-proxy   3         3         3       3            3           kubernetes.io/os=linux   7d23h
weave-net    3         3         3       3            3           <none>                   7d23h
root@control-k8s:~#

root@control-k8s:~# kubectl get po -n kube-system -o wide
NAME                                    READY   STATUS    RESTARTS      AGE   IP               NODE
coredns-55cb58b774-5b474                1/1     Running   1 (46h ago)   8d    10.32.0.1        control-k8s
coredns-55cb58b774-j4lvc                1/1     Running   1 (46h ago)   8d    10.32.0.2        control-k8s
etcd-control-k8s                        1/1     Running   1 (46h ago)   8d    192.168.199.173  control-k8s
kube-apiserver-control-k8s              1/1     Running   1 (46h ago)   8d    192.168.199.173  control-k8s
kube-controller-manager-control-k8s     1/1     Running   2 (4h44m ago) 8d    192.168.199.173  control-k8s
kube-proxy-555xk                        1/1     Running   0             46h   192.168.199.149  node2-k8s
kube-proxy-brgrq                        1/1     Running   1 (46h ago)   8d    192.168.199.174  node1-k8s
kube-proxy-nn4hf                        1/1     Running   1 (46h ago)   8d    192.168.199.173  control-k8s
kube-scheduler-control-k8s              1/1     Running   2 (4h44m ago) 8d    192.168.199.173  control-k8s
weave-net-g7gbz                         2/2     Running   3 (46h ago)   8d    192.168.199.173  control-k8s
weave-net-kfktn                         2/2     Running   3 (46h ago)   8d    192.168.199.174  node1-k8s
weave-net-m5dj5                         2/2     Running   0             46h   192.168.199.149  node2-k8s
```

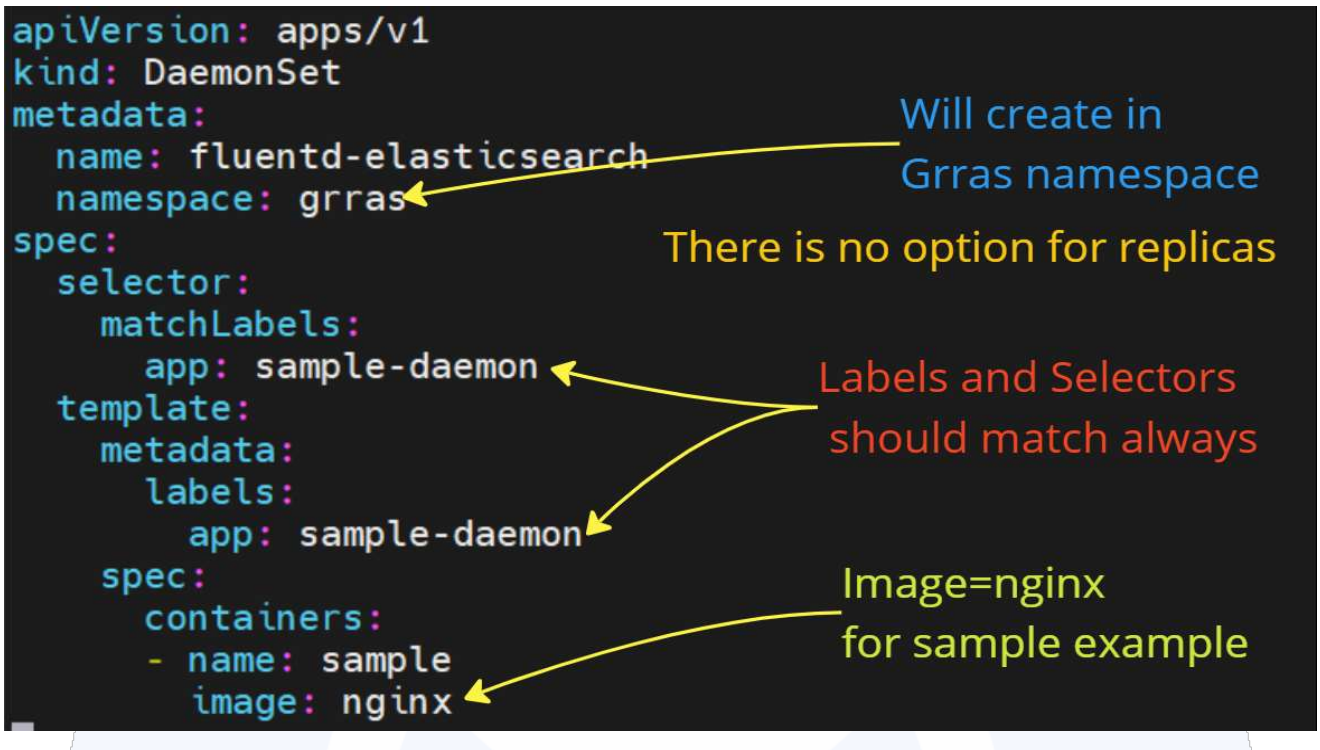
¶ When a new worker node is added to the cluster using the kubeadm join command, DaemonSet Pods are automatically scheduled on that node.

¶ See the below graphical diagram, where explains that DaemonSet is automatically created on all worker nodes (node-5 to node-8) that was recently added to the cluster.



How to Practice DaemonSet?

1. Create a DaemonSet YAML file with the necessary configuration.



DaemonSet is namespace oriented Kubernetes resource. You can check by
Command: #kubectl api-resources

```
root@control-k8s:~# kubectl api-resources | grep -i DaemonSet
daemonsets                        ds          apps/v1                                true         DaemonSet
```

2. Apply the YAML file to the cluster using the kubectl apply-f command.

```
root@control-k8s:~# kubectl apply -f daemonset.yml
daemonset.apps/fluentd-elasticsearch created
```

3. Verify the status of the DaemonSet and ensure it is running on all applicable nodes.

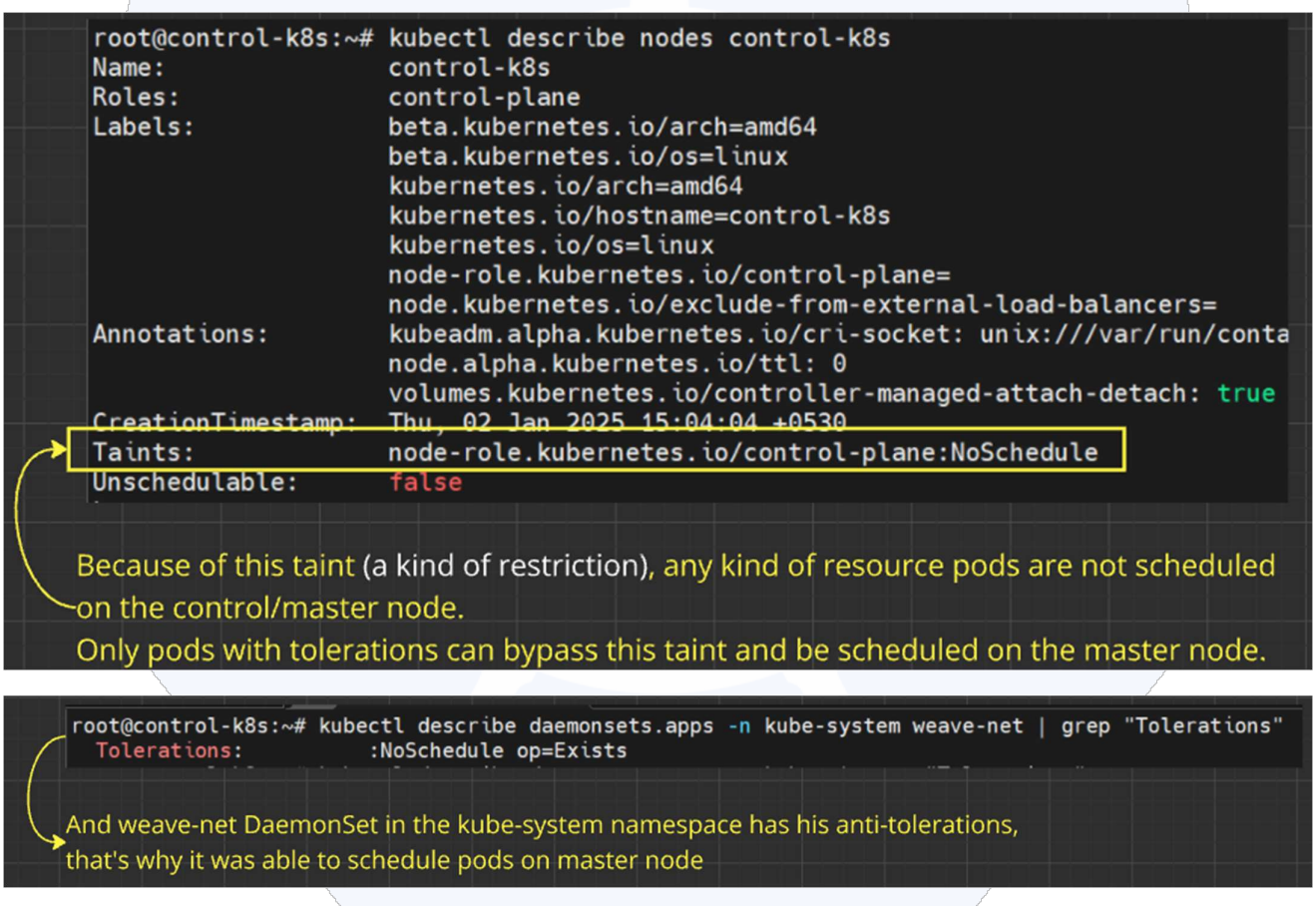
```
root@control-k8s:~# kubectl get daemonsets.apps -n grras
NAME                  DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluentd-elasticsearch   2         2         2       2            2           <none>          67s
```

```
root@control-k8s:~# kubectl get all -n grras
NAME                                READY   STATUS    RESTARTS   AGE
pod/fluentd-elasticsearch-cqkrv   1/1     Running   0          76m
pod/fluentd-elasticsearch-mj95w   1/1     Running   0          76m

NAME                                     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
GE
daemonset.apps/fluentd-elasticsearch   2         2         2       2            2
```

¶ Note: You might be wondering why the DaemonSet is only running on 2 nodes (the worker nodes) and not on the master node, even though we have a 3-node cluster. However, if you check the kube-proxy and weave-net DaemonSets, you'll see they are running on all nodes, including the master node.

¶ Hmmm… That's a valid question! The reason lies in taints and tolerations, which are advanced pod scheduling mechanisms. Don't worry, we'll cover these in detail in a later session.

¶ For now, here's a quick explanation: in the kube-proxy and weave-net DaemonSets, tolerations are configured. These tolerations allow the pods to bypass the taints on the master node, effectively removing the restrictions and letting them run on all nodes.



Static Pod

A **static pod** in Kubernetes is a type of pod that is directly managed by the **kubelet service** on a specific node, rather than being controlled by the Kubernetes **API server**, **controller manager**, or **scheduler**. Unlike regular pods, a static pods are not part of the Kubernetes control manager object like "Pod, RC, RS, Deployment, Daemon-Set" etc

Static pods are defined using configuration files (YAML or JSON) placed in a designated directory on the node, such as **/etc/kubernetes/manifests/.** The kubelet monitors this directory and automatically creates and manages the pods based on the configuration.

Some Key Characteristics:

1. Independent Management: Static pods operate independently of the Kubernetes control plane. They are created and managed by the kubelet without requiring instructions from the API server or other control plane components.
2. Node-Specific: Since they are tied to the kubelet on a specific node, static pods are only available on that node. If the node goes offline, the static pod also becomes unavailable.
3. Visibility: While static pods are managed by the kubelet, they can still appear in kubectl commands if the kubelet communicates with the API server and registers the pod status.
4. Mirror Pods: When a **static pod** is created by the kubelet on a node, it generates a corresponding **mirror pod** in the Kubernetes **API server** if the kubelet is configured to communicate with the API server.
5. What are Mirror Pods?

   Mirror pods are representations of static pods within the Kubernetes API server.

   These pods allow static pods to appear in kubectl outputs (e.g., kubectl get pods) as if they were regular pods managed by the control plane.

   Mirror pods cannot be directly managed or modified by users because their actual management happens through the kubelet and the static pod configuration files on the node.

6. Why are Mirror Pods Created?

   The purpose of mirror pods is to provide visibility and consistency within the Kubernetes cluster. Even though static pods are not managed by the Kubernetes scheduler or controller, their status and lifecycle can still be monitored via the API server.

¶ Practice Static Pod Scenario: Monitoring Tool Deployment

¶ Imagine you need a node-specific monitoring tool (e.g., node-monitor) that must run directly on node2, independent of the Kubernetes control plane. This is critical to ensure the tool remains functional even if the Kubernetes API server becomes temporarily unavailable.

Command: # ssh root@node2

```
root@control-k8s:~# kubectl get no
NAME          STATUS   ROLES           AGE     VERSION
control-k8s   Ready    control-plane   13d     v1.30.8
node1-k8s     Ready    <none>          13d     v1.30.8
node2-k8s     Ready    <none>          7d19h   v1.30.8
root@control-k8s:~# ssh root@node2-k8s
```

First find, what is static manifests file path on node ?

```
root@node2-k8s:~# hostname
node2-k8s
root@node2-k8s:~# systemctl status kubelet.service
● kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
    Drop-In: /usr/lib/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: active (running) since Wed 2025-01-08 16:58:49 IST; 1 week 0 days ago
       Docs: https://kubernetes.io/docs/
   Main PID: 3304 (kubelet)
      Tasks: 14 (limit: 4551)
     Memory: 38.4M
        CPU: 2h 5min 43.101s
    CGroup: /system.slice/kubelet.service
            └─3304 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
                    --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml
```

```
runtimeRequestTimeout: 0s
shutdownGracePeriod: 0s
shutdownGracePeriodCriticalPods: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

Then in yaml you will get the staticPodPath

First Check what is your kubelet config yaml

Put your pods yaml here !

miro

- ¶ You've decided to deploy a static pod with the following specifications:
- ¶ Pod Name: node-monitor
- ¶ Image: busybox (a lightweight image for testing or monitoring)
- ¶ Namespace: grras
- ¶ Node: This pod will run exclusively on node2 as static Pod.

```
root@node2-k8s:~# cd /etc/kubernetes
root@node2-k8s:/etc/kubernetes# ls
kubelet.conf  pki
root@node2-k8s:/etc/kubernetes# mkdir manifests
root@node2-k8s:/etc/kubernetes# cd manifests/
root@node2-k8s:/etc/kubernetes/manifests# pwd
/etc/kubernetes/manifests
root@node2-k8s:/etc/kubernetes/manifests# vim static-pod-example.yml
```

```
root@node2-k8s:~# vim /etc/kubernetes/manifests/static-pod-example.yml
apiVersion: v1
kind: Pod
metadata:
  name: node-monitor
  namespace: grras
  labels:
    app: monitoring
spec:
  containers:
  - name: node-monitor-container
    image: busybox:latest
    command:
    - sh
    - -c
    - "while true; do top -b -n 1; sleep 5; done"
```

Now check from master-node

Command: #kubectl get po-n <namespace>

```
Connection to node2-k8s closed.
root@control-k8s:~# kubectl get po -n grras
NAME                      READY   STATUS    RESTARTS   AGE
node-monitor-node2-k8s    1/1     Running   0          75s
root@control-k8s:~#
```

Try to describe and manage control-node.

Command: #kubectl describe po <pod-name>-n <namespace> | less

```
root@control-k8s:~# kubectl describe pod node-monitor-node2-k8s -n grras | less
Name:         node-monitor-node2-k8s
Namespace:    grras
Priority:     0
Node:         node2-k8s/192.168.199.149
Start Time:   Thu, 16 Jan 2025 13:09:14 +0530
Labels:       app=monitoring
```

Now Try to delete this pod from control-node using kubectl. See this is mirror copy

Command: #kubectl delete pod <pod-name>

```
root@control-k8s:~# kubectl delete pod node-monitor-node2-k8s
pod "node-monitor-node2-k8s" deleted
root@control-k8s:~# kubectl get po -n grras
NAME                    READY   STATUS    RESTARTS   AGE
node-monitor-node2-k8s  1/1     Running   1          3s
root@control-k8s:~#
```

And see your logs are generating by static-pod successfully By running this

command: #watch kubectl logs <pod-name>-n <namespace>

```
root@control-k8s:~# watch kubectl logs pods/node-monitorr-node2-k8s
Every 2.0s: kubectl logs pods/node-monitorr-node2-k8s

Mem: 3650444K used, 310988K free, 2532K shrd, 178292K buff, 2645536K cached
CPU:  0.0% usr  2.5% sys  0.0% nic 97.5% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.12 0.10 0.09 1/419 7
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
    1     0 root     S     4412  0.1   2  0.0 sh -c while true; do top -b -n 1; sleep
    7     1 root     R     4412  0.1   3  0.0 top -b -n 1
```

This was your static pod practical example.

INIT CONTAINERS

What is an Init Container in Kubernetes?

An **init container** is a special type of container in Kubernetes that runs **before the main application containers** (called service containers) in a pod. It is designed to perform **initialization tasks** required to prepare the environment or fulfill specific conditions for the service containers.

How Init Containers Work

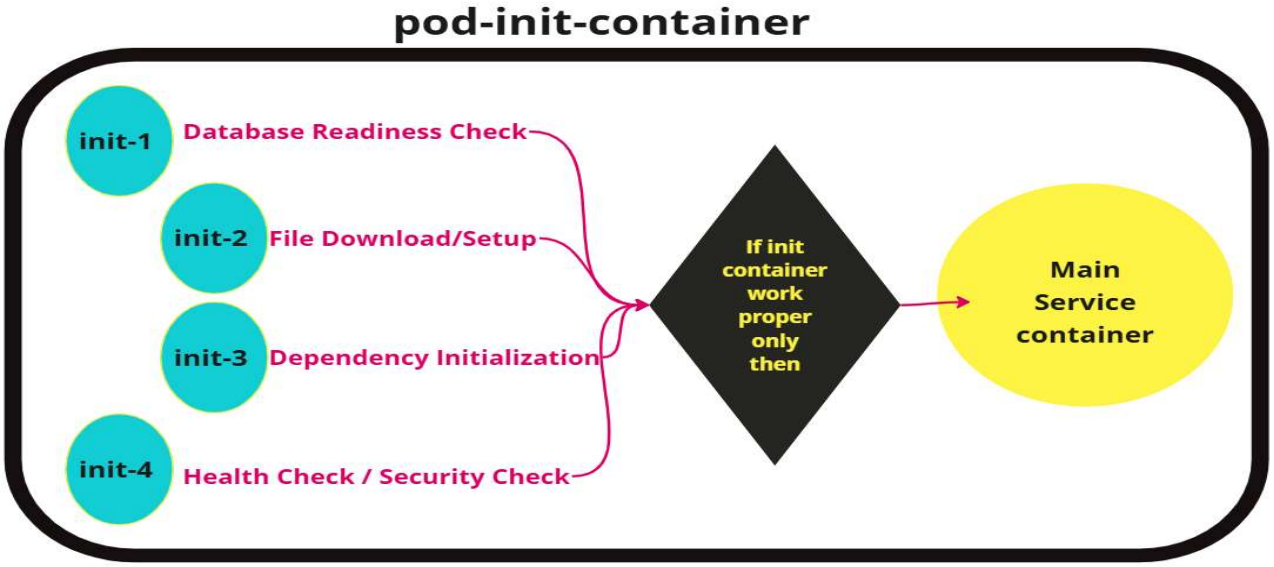
- A pod can have **one or more init containers**, defined in its specification.
- Init containers run **sequentially**, one after another. Each init container must complete successfully before the next one starts.
- After all init containers complete, the **main service containers** in the pod are started.

Why Use Init Containers?

1. Fulfil Multiple Conditions:
    - Init containers can ensure that specific preconditions are met before the main application starts.
      Examples:
        - Waiting for a database to be ready.
        - Downloading configuration files or secrets.
        - Setting up temporary resources.
2. Separation of Concerns:
    - Tasks like environment setup or dependency resolution are separated from the main application logic, improving modularity.
3. Reliability:

- If an init container fails, Kubernetes retries it automatically. The pod does not transition to the "Running" state until all init containers succeed.

In a pod multiple containers share the same IP address, volumes, storage, resource limits, and namespaces.



After the init containers complete successfully, the main service container runs, and the init containers are automatically removed from the pod

Let's understand init Container requirement with a good example Scenario

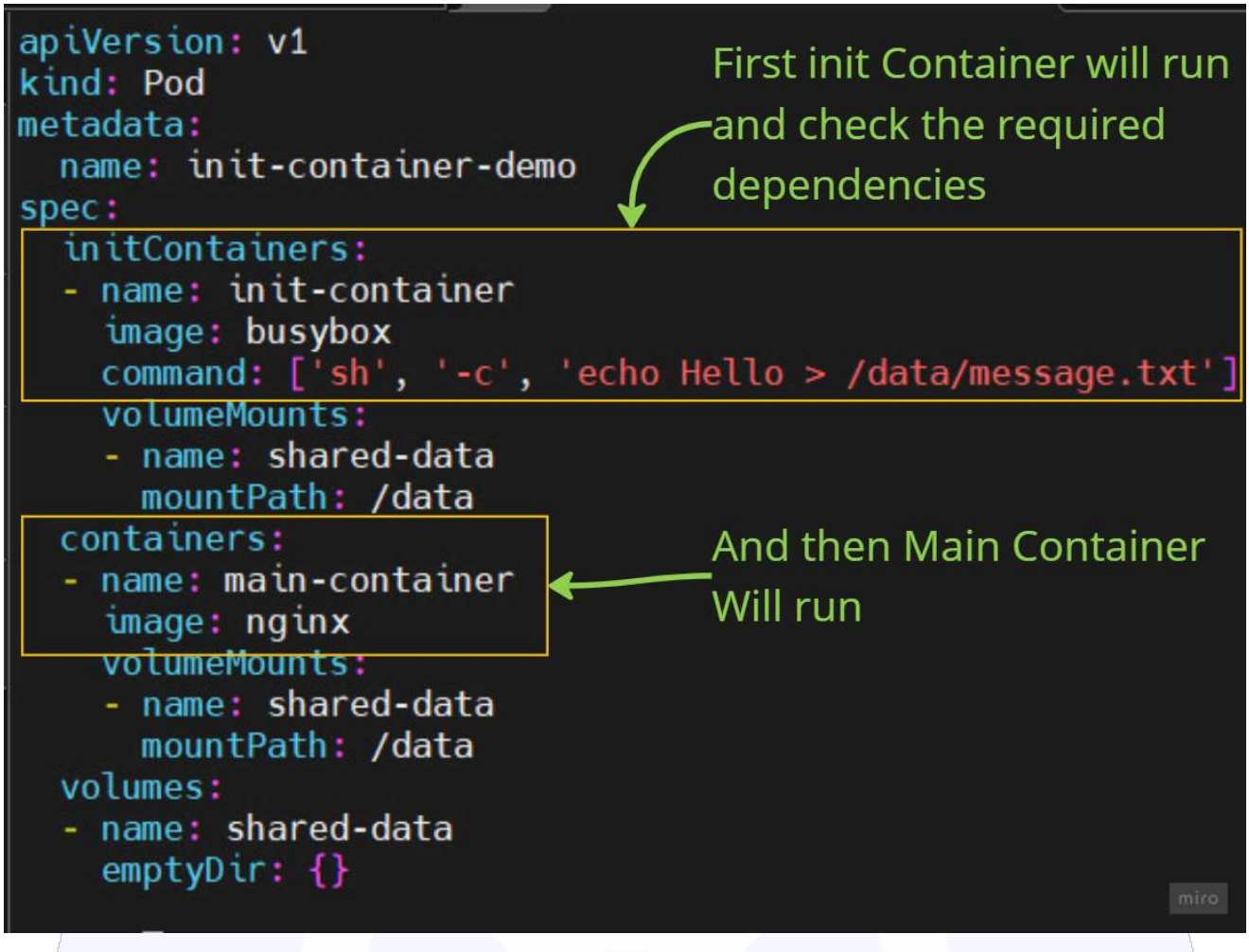
Imagine a pod that hosts a web application. The application depends on:

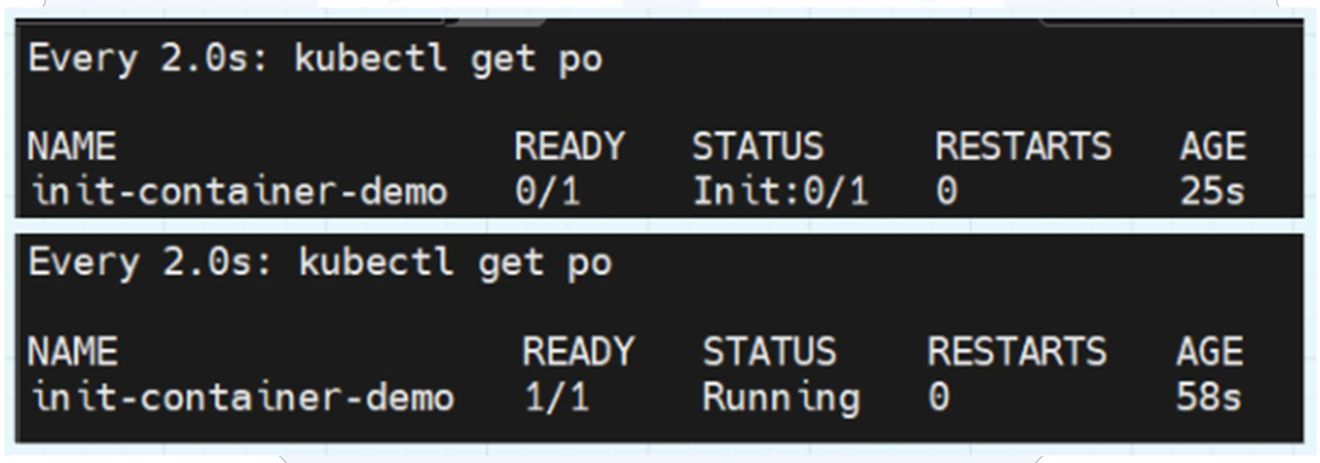
1. A database being available.
2. A configuration file being downloaded.

You can achieve this using init containers:

- The first init container checks if the database is reachable.
- The second init container downloads the configuration file.
- Only after these tasks are complete does the main service container (the web app) start.

YAML EXAMPLE:





Pod Scheduling

What is Pod Scheduling in Kubernetes?

Pod scheduling in Kubernetes is the process of assigning a pod to a specific node within a cluster. When you create a pod, Kubernetes doesn't immediately know where it should run. Instead, it relies on the scheduling process to evaluate the available nodes and pick the best one. This process ensures that workloads are efficiently distributed while meeting the pod's requirements and the cluster's constraints.

By default, Kubernetes uses the kube-scheduler, which is the built-in component responsible for automatically handling this task. It examines various factors like resource requirements, node availability, and scheduling rules to make its decisions.

The default Kubernetes scheduler is the built-in system that handles this task automatically. It looks at factors like:

- How much CPU and memory the pod needs.
- Which nodes have enough resources available.
- Any special rules you've set, like which node the pod must or must not run on.

¶ This doesn't mean that after custom or manual pod scheduling, the kube-scheduler stops working. No, that's not true. The kube-scheduler is still active but instead of making its own independent decisions about which node to run a pod on, it follows the user's instructions or conditions. By applying custom pod scheduling methods, you're essentially telling Kubernetes, "Run this pod on the node that meets my specific requirements or conditions.

Why is Pod Scheduling Different from Default kube-scheduler Scheduling?

While the default kube-scheduler works well for most scenarios, there are times when you need more control. Kubernetes gives you the ability to override or customize the scheduling process for specific requirements. Here's how custom pod scheduling differs from the default:

Here's how custom pod scheduling is different from the default process:

1. Custom Rules:
    - You can define specific rules to control which nodes a pod should or shouldn't run on.
    - For example:
        - "Only run this pod on nodes labeled gpu-enabled."
        - "Don't run this pod on nodes marked low-priority."
        - "Run this pod on that nodes, that match my requirement".
2. Manual Scheduling:
    - You can skip the scheduler altogether and directly assign a pod to a specific node using the nodeName field.

- For example: "Run this pod only on node-1."

¶ **Why It Matters:** The default scheduler is general-purpose, which is great for most workloads. However, when you need fine-grained control, custom scheduling gives you the flexibility to make sure pods run exactly where they're supposed to, based on your unique needs.

In Short

Types of Pod Scheduling

1) nodeName Based Pod Scheduling (Direct)
2) nodeSelector Based Pod Scheduling (Labels Based)
3) Taint & Tolerations
4) Affinity & Anti-Affinity

Let's understand In Detailed

Kubernetes provides various ways to control how and where pods are scheduled on nodes in a cluster. Below are the detailed types of pod scheduling methods:

1. nodeName Based Scheduling (Direct Scheduling)

This is the simplest form of scheduling where you directly assign a pod to a specific node by specifying its name in the nodeName field. The kube-scheduler is bypassed, and the pod is scheduled on the designated node without any further checks.

- How It Works:
  You specify the name of the target node directly in the pod specification.
- Use Case:
  This is mainly used for testing, debugging, or forcing specific pods to run on a particular node in controlled scenarios.
- Syntax of nodeName:

```yaml
spec:
  nodeName: node1-k8s
  containers:
  - name: nginx
    image: nginx:1.14.2
```

- How to Practice:

  Create a pod that should only be scheduled on the specific node **"node1-k8s".** To achieve this, we use the **nodeName scheduling** in the pod specification.

Check the node

Command: #kubectl get no

```
root@control-k8s:~# kubectl get no
NAME          STATUS   ROLES           AGE     VERSION
control-k8s   Ready    control-plane   18d     v1.30.8
node1-k8s     Ready    <none>          18d     v1.30.8
node2-k8s     Ready    <none>          4d19h   v1.30.8
```

Syntax of Pod with nodeName:

Command: # vim nodeNamepod.yml

```
root@control-k8s:~# vim nodeNamePod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: node1-k8s
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Under the Pod spec, at the pod level, we use the nodeName field to specify the scheduling rule.

Command: #kubectl create-f nodeNamepod.yml

```
root@control-k8s:~# kubectl create -f nodeNamePod.yml
pod/nginx created
```

Command: #kubectl get no-o wide

```
root@control-k8s:~# kubectl get po -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP          NODE        NOMINATED NODE   READINESS GATES
init-container-demo 1/1     Running   0          94m   10.40.0.1   node1-k8s   <none>           <none>
nginx               1/1     Running   0          77s   10.40.0.2   node1-k8s   <none>           <none>
root@control-k8s:~#
```

---

2. nodeSelector Based Scheduling (Labels-Based Scheduling)

Instead of forcefully scheduling pods on a specific node using nodeName, the nodeSelector field allows you to schedule pods on nodes that match specific key-value labels. This way, you can define the conditions based on node requirements. It provides a straightforward way to filter nodes based on their labels but doesn't support complex rules.

How It Works:

Nodes are pre-labeled with key-value pairs, and pods are set up to select nodes based on those labels. Labels are crucial because they help the kube-scheduler decide which node meets the conditions defined in the deployment manifest while creating the deployment.

¶ Info: If no node in the cluster has the required labels, the deployment will remain in a waiting state, as this is a hard type of scheduling and the pod cannot be scheduled elsewhere.

Use Case:

Ideal for ensuring workloads run on nodes with specific label characteristics, such as hardware types, environments (e.g., production), or region labels, which are provided earlier by the cluster administrator.

Syntax of nodeSelector with atteching only one key=value pairs:

Note: nodeSelector support AND opration, means both labes should present on singe node.

Two nodes labels not support in nodeSelector.

```
spec:
  nodeSelector:
    region: india
    env: test
  containers:
    - name: nginx
```

How to Practice:

Create a Deployment resource named **“web-deploy”** with **3 replicas**, ensuring that the pods are deployed on nodes that have the following key-value label pairs: **region=india, env=test**. You can achieve this by using the nodeSelector field in the pod specification.

Check Node labels:

Command: # kubectl get no –show-labels

```
root@control-k8s:~# kubectl get no --show-labels | grep -i -e env -e region
node1-k8s     Ready    <none>          19d     v1.30.8   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,env=test,kuberne
tes.io/arch=amd64,kubernetes.io/hostname=node1-k8s,kubernetes.io/os=linux,region=india
node2-k8s     Ready    <none>          4d20h   v1.30.8   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,env=prod,kuberne
tes.io/arch=amd64,kubernetes.io/hostname=node2-k8s,kubernetes.io/os=linux,region=usa
root@control-k8s:~#
```

¶ Info: In this image, you will see that labels are already attached to the specific nodes.

On node1 = env=test & region=india

On node2 = env=prod & region=usa

¶ And node1 has all the required labels as per requirement

Create deployment Manifests file mentioning nodeSelector with preferred key=labels:
Command: # vim web-deploy.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
        env: test
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Both should same

Define your Pod Scheduling type here

Run this deployment
Command: # kubectl create-f web-deploy.yml

And check the pods status (Pods will deployed on node1-k8s only because of labels)

```
root@control-k8s:~# kubectl create -f web-deploy.yml
deployment.apps/web-deploy created
```

Command: # kubectl get pods-o wide

```
root@control-k8s:~# kubectl get po -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP           NODE
web-deploy-654d564ffb-gtx2w   1/1     Running   0          32s   10.40.0.2    node1-k8s
web-deploy-654d564ffb-p6n59   1/1     Running   0          32s   10.40.0.1    node1-k8s
web-deploy-654d564ffb-v8rkp   1/1     Running   0          32s   10.40.0.3    node1-k8s
```

Command: # kubectl get deploy web-deploy

```
root@control-k8s:~# kubectl get deploy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
web-deploy   3/3     3            3           3m23s
```

Some of you might wonder if it's possible to use both types of scheduling (nodeName & nodeSelector) in a single manifest file. The answer is yes. You can specify both

nodeName and nodeSelector in the same manifest file, but nodeName will always take precedence over nodeSelector.

Why this happened:

Because nodeName directly forcefully specifies the exact node where the pod must be scheduled, bypassing the kube-scheduler entirely. On the other hand, nodeSelector relies on the kube-scheduler to match key value labels of nodes and determine where the pod should run.

Candidate Assignment Question:

Q1: Assign the label env=test to node1 and env=prod to node2. Then, schedule a pod using nodeSelector in such a way that the pod can run on both labeled nodes instead of just one.
Your question is "can you use operators with nodeSelector" Pod Scheduling ??

Answer:

- **NO,** nodeSelector does not support using operators like In, Equal, Exists, etc. It only supports single key-value matching.
- To cover multiple labels or achieve more complex scheduling logic, you need to use taints and tolerations instead.

Why Taints and Tolerations?

- **nodeSelector can only check one specific key-value pair.**
- If you want to schedule the pod on nodes with either env=test or env=prod, you cannot use nodeSelector. But you can use taints on the nodes and tolerations on the pod to allow scheduling on both nodes with the help of operators (In,NotIn, Exists).

Taints and Tolerations Based Scheduling:

Understanding the Need for Taints and Tolerations

Before diving into the concept of taints and tolerations, let's first understand the problem they solve and the use case in Kubernetes clusters.

The Challenge: Diverse Node Configurations

In a Kubernetes cluster, nodes are often not identical or same. They may have different configurations, capabilities, and locations. For example:

- Hardware Differences:
  Some nodes may have high-performance GPUs for AI or machine learning workloads. Others may have high CPU and RAM for compute-intensive tasks and some nodes may have SSD storage for fast data access, while others use SATA disks for general-purpose storage.
- Regional Variations:

Nodes may be distributed across different regions or availability zones, such as:

- region=north-india, south-india, east-india, west-india
- region=usa, russia, australia, japan,
- region=south-asia

- Specialized Nodes:

  Some nodes may be dedicated to specific workloads, such as database services or caching systems. So, In this diverse environment, not all pods can or should run on all nodes. For example:

  - A lightweight application doesn't need to run on an expensive GPU-based node.
  - A pod requiring SSD storage must not be scheduled on a node with only SATA disks.
  - A service specific to North India should only run on nodes in the north-india region.

  If Kubernetes doesn't enforce placement rules, pods will be scheduled randomly, leading to inefficient resource utilization and potential failures.

How Kubernetes Solves This Challenge

To ensure proper scheduling, Kubernetes provides two mechanisms:

- Node Labels:
  - Assign labels to nodes as per the node specification.
  - Labels are simple key-value pairs that describe the attributes of nodes.
  - Example:
    - region=india
    - hardware=gpu
    - storage=ssd
  - These labels help identify the characteristics of nodes, making it easier to match pods to the right nodes. if you remember in previous part where in nodeSelector based Pod scheduling we use these labels for provide conditions. That run this pod only on that node that have storage=ssd etc.

Limitation: Labels alone don't restrict pod placement. They work passively to define attributes but don't enforce rules.

- Taints and Tolerations:
  - Taints: Taint are a kind of flags applied only on nodes, that used to specifying to kube-scheduler that "**Hey listen I have this type of restriction, if any pod have my matching toleration that is a kind of bypass entry card, only that time you please schedule pod on this node otherwise not schedule.**"

Key Point to Remember

To allow a pod or deployment to run on a node with specific restrictions, you need to add the **right toleration** to the pod or deployment resource manifests file. This ensures that the Kubernetes kube-scheduler knows the pod is allowed to bypass the node's taint and run there.

"It's like a taint is a lock, and a toleration is the key. Only the right key (matching toleration) can unlock the lock (taint). Not every key will work."

- Tolerations: A kind of bypass entry card act as permissions in pods that allow them to bypass the node's restrictions and authorize to schedule on node.

We can understand now that, Taints and tolerations work together to provide strict control over pod placement.

Types of taint on nodes: (Taints are like Locks)

In Kubernetes, **taints** are applied to nodes to control which pods can run on them. They act like special rules that determine whether a pod can be scheduled on a node based on whether it has the proper **toleration** (like a special card that lets the pod bypass the rule). Kubernetes provides three types of taints, each serving a specific purpose:

1) NoSchedule
2) PreferNoSchedule
3) NoExecute

1) NoSchedule

This taint tells Kubernetes:

- "Don't schedule any new pods on this node unless the pod has the special card (toleration) to bypass this rule."
- Pods already running on the node will not be affected—they can stay.

Example: It's like a VIP area with a sign that says, "Only those with a special card can enter." Existing guests (pods) are allowed to stay, but new guests must have the special card to get in.

To practical this, let add taint first.

How to add "NoSchedule" taint on nodes?

Command: #kubectl taint node node1-k8s key=value:NoSchedule

```
root@control-k8s:~# kubectl taint node node1-k8s env=test:NoSchedule
node/node1-k8s tainted
```

```
root@control-k8s:~# kubectl describe node node1-k8s | grep -i taint
Taints:             env=test:NoSchedule
root@control-k8s:~#
```

How to remove “NoSchedule” taint from node? (Only add - at the last of command)

Command: #kubectl taint node node1-k8s key=value:NoSchedule-

```
root@control-k8s:~# kubectl taint node node1-k8s env=test:NoSchedule-
node/node1-k8s untainted
root@control-k8s:~#
```

```
root@control-k8s:~# kubectl describe node node1-k8s | grep -i taint
Taints:             <none>
root@control-k8s:~#
```

---

2) PreferNoSchedule

This taint is less strict and means:

- “Try not to schedule new pods here, but if there’s no other place, it’s okay to let them in.
- I mean avoid myself ASAP for pod scheduling, but if required then schedule on me”

Example: Think of it as a sign that reads, "Preferably, don’t enter unless it's necessary." Kubernetes will try to avoid scheduling new pods here, but will let them in if no other options are available.

How to add “PreferNoSchedule” taint on node

Command: #kubectl taint node node1-k8s key=value:PreferNoSchedule

```
root@control-k8s:~# kubectl taint node node2-k8s env=prod:PreferNoSchedule
node/node2-k8s tainted
root@control-k8s:~# kubectl describe node node2-k8s | grep -i taint
Taints:             env=prod:PreferNoSchedule
root@control-k8s:~#
```

How to remove “PreferNoSchedule” taint from node? (Only add - at the last of command)

Command: #kubectl taint node node1-k8s key=value:PreferNoSchedule-

```
root@control-k8s:~# kubectl taint node node2-k8s env=prod:PreferNoSchedule-
node/node2-k8s untainted
root@control-k8s:~# kubectl describe node node2-k8s | grep -i taint
Taints:             <none>
root@control-k8s:~#
```

---

3) NoExecute

This is the strictest taint. It works in two ways:

- "Do not schedule new pods unless they have the special card to bypass this taint."
- "Evict (remove) any existing pods that don't have the special card to stay here."

Example: Imagine a sign that says, "No entry without a special card—if you're already inside without one, you need to leave immediately." Only pods with the special card are allowed to remain or be scheduled on this node.

How to add "NoExecute" taint on node

Command: #kubectl taint node node1-k8s key=value:NoExecute

```
root@control-k8s:~# kubectl taint node control-k8s region=india:NoExecute
node/control-k8s tainted
root@control-k8s:~# kubectl describe node control-k8s | grep -i taint
Taints:             region=india:NoExecute
```

How to remove "NoExecute" taint from node? (Only add - at the last of command)

Command: #kubectl taint node node1-k8s key=value:NoExecute-

```
root@control-k8s:~# kubectl taint node control-k8s region=india:NoExecute-
node/control-k8s untainted
```

Check all taints status on node:

```
root@control-k8s:~# kubectl describe node | grep -i taint
Taints:             env=test:NoSchedule
Taints:             env=prod:PreferNoSchedule
Taints:             env=backup:NoExecute
root@control-k8s:~#
```

Types of Tolerations: (Only the right Key will able to open the right lock)

Tolerations are used on pods to match the taints on nodes. Similar to taints, tolerations also have three types. They work as follows:

1. **NoSchedule:** Allows the pod to run on a node with a NoSchedule taint.
2. **PreferNoSchedule:** The pod avoids nodes with this taint but can run there if no other nodes are available.
3. **NoExecute:** Allows the pod to stay on or run on a node with a NoExecute taint, preventing eviction.

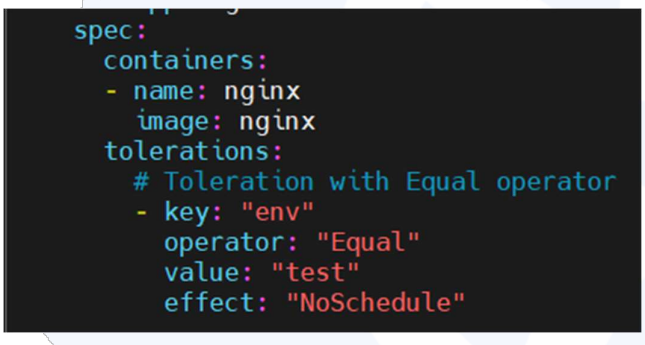
Operators with tolerations:

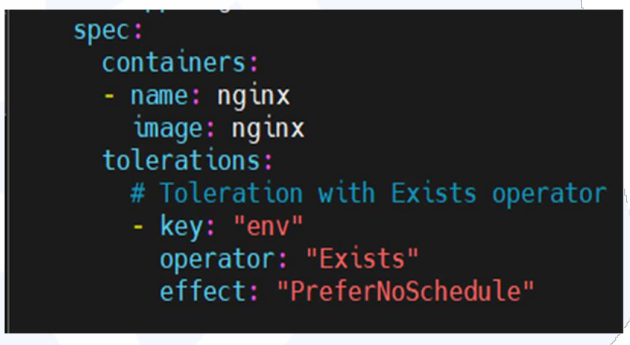
In Kubernetes, **tolerations** use two main operators: **Equal** and **Exists**. The **Equal** operator matches a taint's key and value exactly with the toleration's key and value. It ensures that only taints with the same key-value pair will be tolerated.

On the other hand, the **Exists** operator ignores the value and only requires the key to match. This means any taint with the specified key will be tolerated, regardless of its value. These operators help fine-tune which nodes a pod can be scheduled on based on the taints applied to the nodes.

If you do not use operators in tolerations, Kubernetes will assume the **Equal** operator by default. This means that the toleration will only match taints that have both the **same key and value** as the toleration. Without specifying an operator, Kubernetes will look for taints that exactly match the key-value pair, and the pod will only be scheduled on nodes that have a taint with that specific combination.

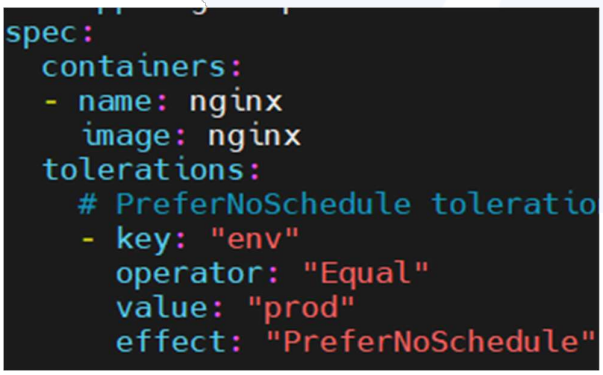
1. Create the NoSchedule toleration with both operators
- (NoSchedule toleration with Equal & Exists operator)

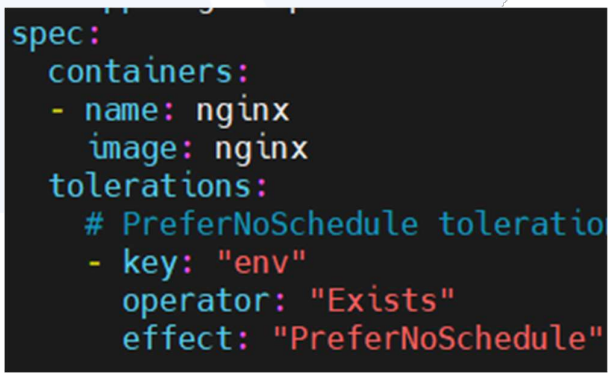




---

2. Create the PreferNoSchedule toleration with both operators
- (PreferNoSchedule toleration with Equal & Exists operator)





---

3. Create the NoExecute toleration with both operators
- (NoExecute toleration with Equal & Exists operator)

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
  tolerations:
    # NoExecute toleratio
    - key: "env"
      operator: "Equal"
      value: "backup"
      effect: "NoExecute"
```

```yaml
spec:
  containers:
  - name: nginx
    image: nginx
  tolerations:
    # NoExecute toleratio
    - key: "env"
      operator: "Exists"
      effect: "NoExecute"
```

Note:

In toleration, only the **"Equal"** and **"Exists"** operators will work with the toleration.

The **"In"** operator doesn't work, as it shows the following message

```
* spec.template.spec.tolerations[0].operator: Unsupported value: "In": supported values: "Equal", "Exists"
* spec.template.spec.tolerations[1].operator: Unsupported value: "In": supported values: "Equal", "Exists"
```

And same thing will work for other taint and tolerations included "PreferNoSchedule & NoExecute"

Syntax of PreferNoSchedule toleration:

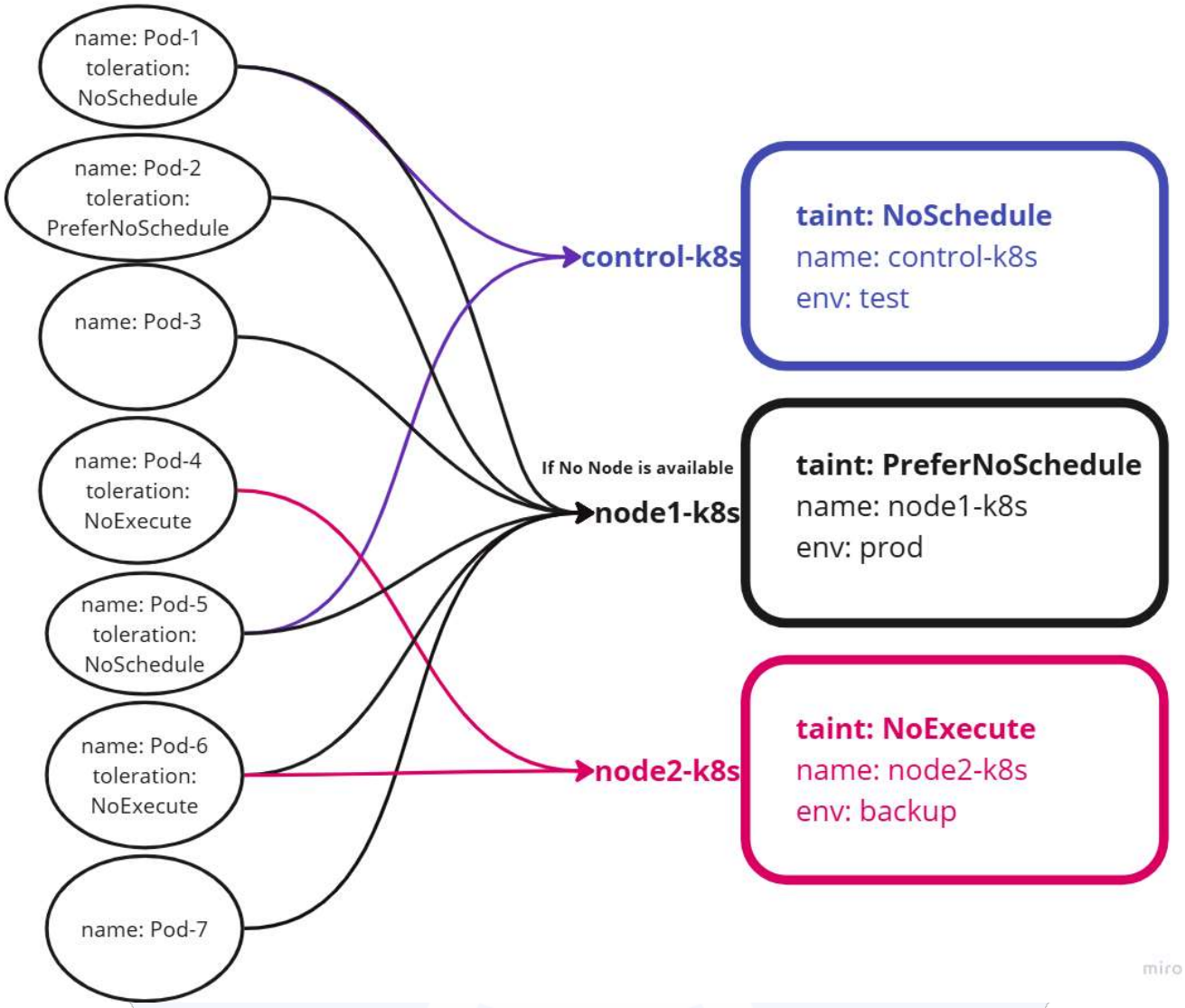
```yaml
  metadata:
    labels:
      app: nginx
  spec:
    tolerations:
      - key: "env"
        operator: "Equal"
        value: prod
        effect: "PreferNoSchedule"
    containers:
```

In both cases (taint available or not) pods will deployed on that nodes that has taint "PreferNoSchedule"

Syntax of NoExecute toleration:

```yaml
  spec:
    tolerations:
      - key: "env"
        operator: "Equal"
        value: "dev"
        effect: "NoExecute"
    containers:
    - name: nginx
```

Ref Image: Understand which types of tainted nodes are able to deploy pods based on their tolerations.



SR. NO	Case	Cordon	Taint	Toleration	Manual Pod Scheduling Possible?	Normal Pod शेड्यूल होगा?	DaemonSet शेड्यूल होगा?	पहले से चल रहे Pod का क्या होगा?
1	केवल Cordon	✅ हाँ	❌ नही	❌ नही	✅ हाँ (मैन्युअल फोर्स कर सकते है)	❌ नही	✅ हाँ	✅ रहेगा
2	केवल NoSchedule Taint	❌ नही	✅ NoSchedule	❌ नही	❌ नही	❌ नही	❌ नही	✅ रहेगा
3	केवल NoExecute Taint	❌ नही	✅ NoExecute	❌ नही	❌ नही	❌ नही	❌ नही	❌ हट जाएगा
4	केवल PreferNoSchedule Taint	❌ नही	✅ PreferNoSchedule	❌ नही	✅ हाँ	✅ हो सकता है	✅ हाँ	✅ रहेगा
5	Cordon + NoSchedule Taint	✅ हाँ	✅ NoSchedule	❌ नही	❌ नही	❌ नही	❌ नही	✅ रहेगा
6	Cordon + NoExecute Taint	✅ हाँ	✅ NoExecute	❌ नही	❌ नही	❌ नही	❌ नही	❌ हट जाएगा
7	Cordon + PreferNoSchedule Taint	✅ हाँ	✅ PreferNoSchedule	❌ नही	✅ हाँ	✅ हो सकता है	✅ हाँ	✅ रहेगा
8	NoSchedule Taint + Toleration	❌ नही	✅ NoSchedule	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा
9	NoExecute Taint + Toleration	❌ नही	✅ NoExecute	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा
10	PreferNoSchedule Taint + Toleration	❌ नही	✅ PreferNoSchedule	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा
11	Cordon + NoSchedule Taint + Toleration	✅ हाँ	✅ NoSchedule	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा
12	Cordon + NoExecute Taint + Toleration	✅ हाँ	✅ NoExecute	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा
13	Cordon + PreferNoSchedule Taint + Toleration	✅ हाँ	✅ PreferNoSchedule	✅ हाँ	✅ हाँ	✅ होगा	✅ हाँ	✅ रहेगा

Affinity and Anti-Affinity Based Scheduling

Affinity and anti-affinity give more advanced and flexible ways to control Pod scheduling compared to all previous basic methods like nodeName, nodeSelector, taints & tolerations, and cordon & uncordon.

What is Affinity and Anti-Affinity?

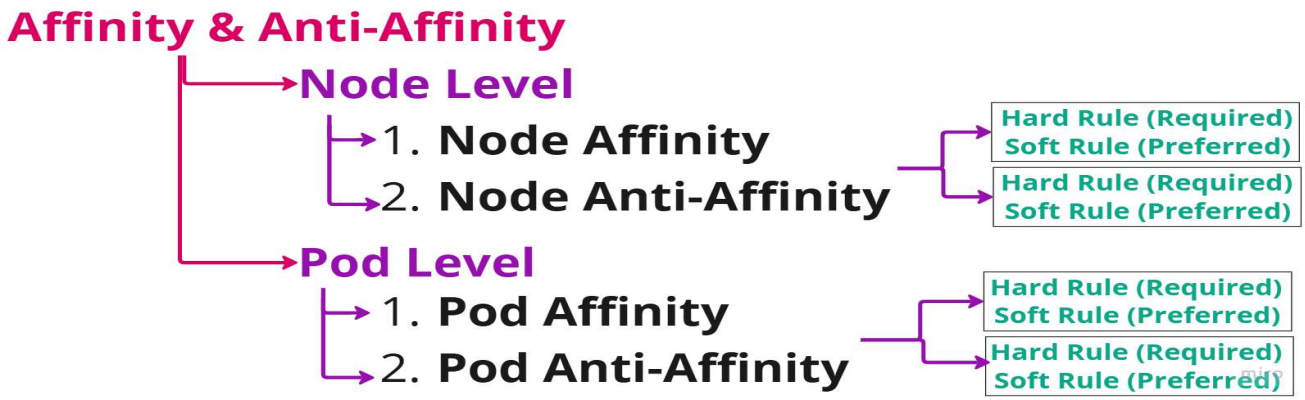
In simple words, **affinity and anti-affinity** are ways to tell Kubernetes **where a pod should or should not run**.

Let's understand this with a fun example:

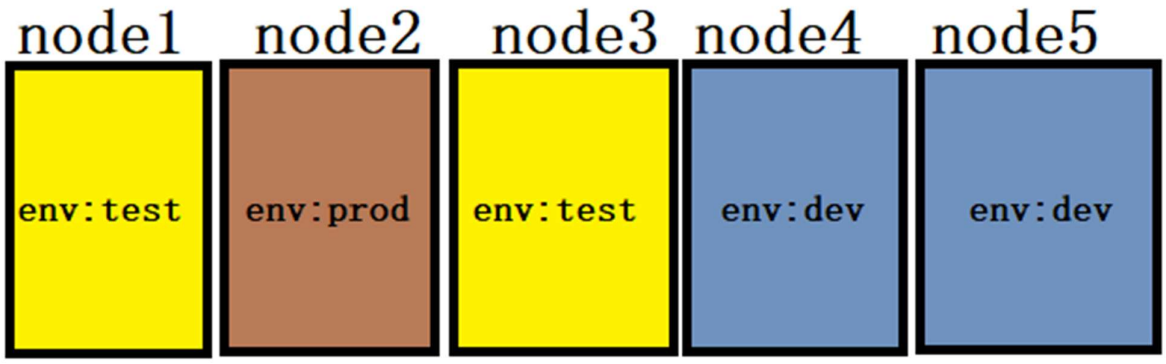
- **Affinity** means **"I want to stay with you"** or **"I want to be near you"**.
- **Anti-affinity** means **"I don't want to stay with you"** or **"I want to be away from you"**.

After that, we need to understand that these two methods can be applied at both the **node level** and the **pod level**. So, we have:

- Node Affinity (Node Level)
- Node Anti-Affinity (Node Level)
- Pod Affinity (Pod Level)
- Pod Anti-Affinity (Pod Level)



Now, let's understand these terms with a better example.



Suppose we have a **6-node**

- 1 Control Plane Node (Pre-tainted with NoSchedule)
- 5 Worker Nodes with the following labels:
    - Node 1 & Node 3 → env=test
    - Node 2 → env=prod
    - Node 4 & Node 5 → env=dev

Now, based on these labels, we can apply Affinity and Anti-Affinity rules to control where our Pods should or should not be scheduled.

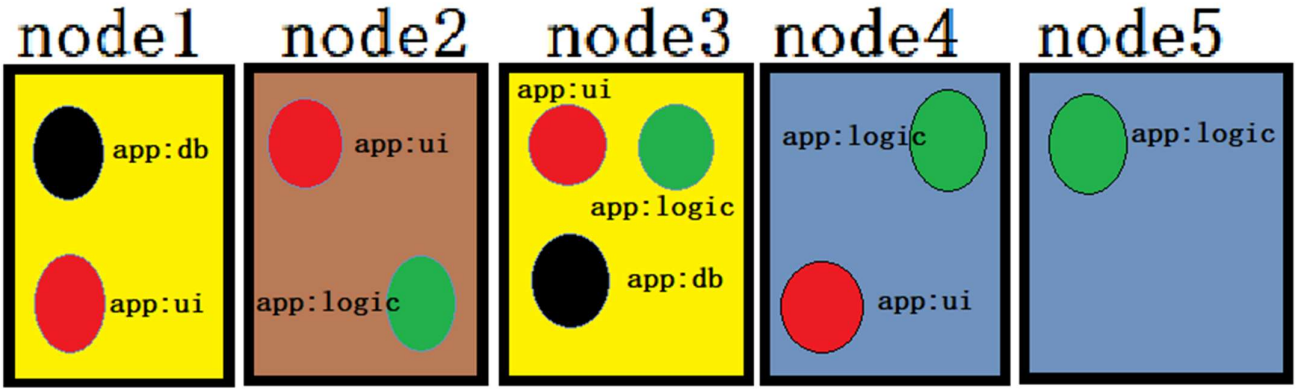
Node Affinity:

- Simple Explanation: A Pod or Deployment wants to schedule its replicas only on specific nodes that have already a particular label.
- Example: If we want our Pods to run only on nodes with the label env=dev, we will apply Node Affinity with env=dev.
- Rule Meaning: "I want to stay only with you" or "I want to be only near you."
- Result: Pods will be scheduled only on Node 4 & Node 5.

Node Anti-Affinity:

Simple Explanation: A Pod or Deployment wants to avoid certain nodes based on their labels.

- Example: If we apply Node Anti-Affinity to env=dev, the Pods will not be scheduled on Node 4 & Node 5.
- Rule Meaning: "I don't want to be with you."
- Result: Pods will be scheduled only on Node 1, Node 2, and Node 3, avoiding Node 4 & Node 5.



Hard and Soft Rules in Affinity & Anti-Affinity

Affinity and Anti-Affinity rules apply at **two levels**:

1 Node Level (Node Affinity & Node Anti-Affinity)

2 Pod Level (Pod Affinity & Pod Anti-Affinity)

Each level has **Hard (Required)** and **Soft (Preferred)** rules.

1 Node Level Rules

Hard Node Affinity (Required)

- Pod **must** be scheduled on specific nodes that match the rule.
- If no matching node is found, the Pod **will not be scheduled**.
- Implemented using: requiredDuringSchedulingIgnoredDuringExecution

Example:

- "I want to schedule my **backend Pods only on nodes with label env=prod**."
- If no node has env=prod, the backend Pods **won't be created**.

Soft Node Affinity (Preferred)

- Pod **should** be scheduled on specific nodes, **but it's not mandatory**.
- If no matching node is found, Kubernetes will schedule it on any available node.
- Implemented using: preferredDuringSchedulingIgnoredDuringExecution

Example:

- "I prefer to schedule my **backend Pods on nodes with env=prod, but if not available, place them anywhere**."
- If no node has env=prod, the backend Pods **can run on other nodes**.

Hard Node Anti-Affinity (Required)

- Pod **must not** be scheduled on specific nodes.
- If no alternative node is available, the Pod **won't be scheduled**.

- Implemented using: requiredDuringSchedulingIgnoredDuringExecution

Example:

- "I don't want to schedule my **frontend Pods on nodes with env=dev**."
- If all nodes have env=dev, the frontend Pods **won't be created**.

Soft Node Anti-Affinity (Preferred)

- Pod **should not** be scheduled on specific nodes, **but if needed, it can be**.
- Implemented using: preferredDuringSchedulingIgnoredDuringExecution

Example:

- "I prefer to **avoid scheduling frontend Pods on nodes with env=dev**, but if no other node is available, they can be placed there."
- If no other node is free, Kubernetes will still schedule the Pod on env=dev nodes.

---

2 Pod Level Rules

Hard Pod Affinity (Required)

- Pod **must** be scheduled on the same node as another specific Pod type.
- If no matching node exists, the Pod **won't be scheduled**.

Example:

- "I want to deploy **log Pods only on nodes where logic Pods (app=logic) are running**."
- If no node has app=logic, **log Pods won't be scheduled**.

Soft Pod Affinity (Preferred)

- Pod **should** be scheduled near a specific Pod type **if possible**.
- If no matching node is found, Kubernetes will still schedule it somewhere.

Example:

- "I prefer to deploy **log Pods on nodes where logic Pods (app=logic) are running, but if not available, deploy anywhere**."
- If no app=logic Pod exists, the **log Pods can run on other nodes**.

Hard Pod Anti-Affinity (Required)

- Pod **must not** be scheduled on the same node as another specific Pod type.
- If no alternative node exists, the Pod **won't be scheduled**.

Example:

- "I don't want to deploy **web Pods on nodes where database Pods (app=database) are running**."
- If all nodes have database Pods, **web Pods won't be scheduled**.

Soft Pod Anti-Affinity (Preferred)

- Pod **should not** be scheduled on the same node as another specific Pod type, **but if needed, it can be**.

Example:

- "I prefer to avoid **web Pods on nodes where database Pods (app=database) are running, but if no choice, they can be scheduled there**."
- If no other node is free, Kubernetes will still schedule web Pods on database nodes.

YAML File

Node Affinity (Pod runs only on Nodes with a specific label)

```
apiVersion: v1
kind: Pod
metadata:
  name: node-affinity-example
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - us-west-1
  containers:
  - name: nginx
    image: nginx
```

Node Anti-Affinity (Pod avoids Nodes with a specific label)

```
apiVersion: v1
kind: Pod
metadata:
  name: node-anti-affinity-example
spec:
  affinity:
    nodeAffinity:
```

```
      requiredDuringSchedulingIgnoredDuringExecution:
       nodeSelectorTerms:
       - matchExpressions:
         - key: disallowed-node
           operator: NotIn
           values:
           - "true"
  containers:
  - name: nginx
    image: nginx
```

Pod Affinity (Pod runs on the same Node as a specific Pod)

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-affinity-example
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - backend
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx
    image: nginx
```

Pod Anti-Affinity (Pod avoids Nodes where a specific Pod is running)

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-anti-affinity-example
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          matchExpressions:
          - key: app
```

```
            operator: In
            values:
            - frontend
        topologyKey: "kubernetes.io/hostname"
    containers:
    - name: nginx
      image: nginx
```

What is Topology Key?

When implementing Pod Affinity or Anti-Affinity, we use topologyKey.

- topologyKey defines how Pods should be scheduled based on a specific topology, when we are working on muti-node over Cloud service like AWS, AZURE, GCP or Multi-cloud using VPC cloud networking.
- The default value is kubernetes.io/hostname, ensuring that Pods are scheduled on the same Node.
- It can also be set to **zone, region,** or custom labels.

What does Weight: 1 mean?

When we use soft rules (preferred rules) in Pod Affinity or Anti-Affinity, we assign a weight to tell Kubernetes how important the rule is.

- Weight: 1 → The rule is not very important; if followed, it's good, but it's not mandatory.
- Weight: 100 → The rule is highly important, and Kubernetes will try harder to follow it.

In simple terms: Higher weight means higher priority.

Real-World Assignment

Imagine you are a Kubernetes administrator managing a cloud-based e-commerce application. The system has three major components:

1. Frontend: A web interface for customers.
2. Backend: Handles business logic and API calls.
3. Database: Stores customer data and transactions.

Your challenge:

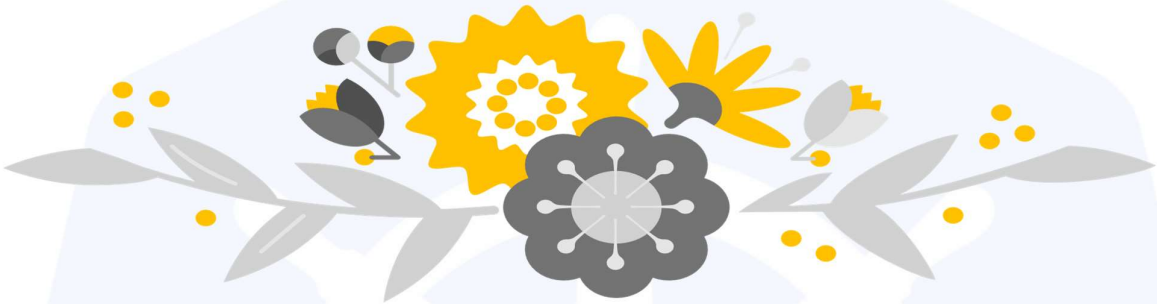
- Ensure that the backend and frontend run on the same Node for better performance.
- Ensure that database Pods do not run on the same Node for fault tolerance.
- Ensure that backend Pods are not scheduled on Nodes with low memory.
- Ensure that at least one replica of each component runs in each availability zone.

Steps:

1. Define Labels for Nodes: Label some Nodes as high-memory, low-memory, zone-a, and zone-b.
2. Apply Node Affinity: Ensure the backend only runs on high-memory Nodes.
3. Apply Pod Affinity: Ensure the frontend and backend run together.
4. Apply Pod Anti-Affinity: Ensure the database Pods do not run on the same Node.
5. Use topologyKey to distribute workloads across different zones.

Use the provided YAML templates and modify them to meet the assignment goals.

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**



ConfigMap and Secrets in Kubernetes

1. Introduction

In Kubernetes, we need to store some settings and data, such as:

- **Non-Sensitive Data** – Application settings, URLs, log levels, etc.
- **Sensitive Data** – Passwords, API Keys, tokens, certificates etc.

To store these types of data, Kubernetes provides **ConfigMap** and **Secret**.

2. Difference Between ConfigMap and Secret

Feature	ConfigMap	Secret
Data Type	Non-Sensitive Data	Sensitive Data
Storage	Plain Text	Base64 Encoded
Usage	Application Settings	Passwords, Tokens, API Keys

Note: Secrets are only Base64 encoded, not encrypted. To secure them, use **etcd encryption** and **RBAC**.

3. Ways to Create ConfigMap and Secret

Both can be created using:

1. **kubectl command**
2. **YAML file**

A. Creating ConfigMap and Secret via Command

Create a ConfigMap:

```
kubectl create configmap my-config --from-literal=APP_NAME=MyApp --from-literal=APP_VERSION=1.0 -n namespace
```

Create a Secret:

```
kubectl create secret generic my-secret --from-literal=DB_USER=admin --from-literal=DB_PASS=pass123 -n namespace
```

B. Creating ConfigMap and Secret via YAML

ConfigMap.yaml:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  APP_NAME: "MyApp"
  APP_VERSION: "1.0"
```

Secret.yaml:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  DB_USER: YWRtaW4=  # Base64 encoded "admin"
  DB_PASS: cGFzczEyMw==  # Base64 encoded "pass123"
```

How to encode data in Base64:

```
echo -n "admin" | base64  # Output: YWRtaW4=
```

```
echo -n "pass123" | base64  # Output: cGFzczEyMw==
```

4. How to Use ConfigMap and Secret?

A. Using as Environment Variables **(Only one key at a time)**

(Using a ConfigMap Key as an Environment Variable)

```
env:
- name: MY_ENV_VAR
  valueFrom:
    configMapKeyRef:
      name: my-config
      key: APP_NAME
```

Using a Secret Key as an Environment Variable **(Only one key at a time)**

```
env:
- name: DB_USER
  valueFrom:
    secretKeyRef:
      name: my-secret
      key: DB_USER
```

B. Using as Environment Variables (All config at a time)

(Using a configMapRef as an Environment Variable)

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-ref-pod
spec:
  containers:
  - name: my-container
    image: busybox
    envFrom:
    - configMapRef:
        name: my-config  #Access complete config Map
```

(Using a secretRef as an Environment Variable) **(All secret at a time)**

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-ref-pod
spec:
  containers:
  - name: my-container
    image: busybox
    envFrom:
    - secretRef:
        name: my-secret  # Access complete secret
```

Note: Using --from-file Option (Loading Data from a File)

If you have a file containing multiple key-value pairs, you can use the --from-file option.

Creating a ConfigMap from a File

kubectl create configmap my-config --from-file=config.txt

Example config.txt File:

```
APP_NAME=MyApp
APP_VERSION=1.0
```

Generated ConfigMap YAML:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  config.txt: |
    APP_NAME=MyApp
    APP_VERSION=1.0
```

Creating a Secret from a File

```
kubectl create secret generic my-secret --from-file=credentials.txt
```

Example credentials.txt File:

```
DB_USER=admin
DB_PASS=pass123
```

Generated Secret YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  credentials.txt: |
    RFIgVVNFUj1hZG1pbg==
    REJfUEFTUz1wYXNzMTIz
```

Now Mounting Entire File in a Pod (Using ConfigMap and Secret as Volumes)

D. Using as Volumes

You can attach and mount the entire ConfigMap or Secret as a volume.

Mounting ConfigMap as a Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-pod
spec:
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - name: config-volume
      mountPath: /config  # Mount point
  volumes:
  - name: config-volume
    configMap:
```

```
      name: my-config
```

Mounting Secret as a Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - name: secret-volume
      mountPath: /secrets  # Mount point
  volumes:
  - name: secret-volume
    secret:
      secretName: my-secret
```

About the topic Final words :-

- **ConfigMap** is used to store and manage non-sensitive data.
- **Secret** is used to store sensitive data securely.
- Both can be used as **Environment Variables, Command-line Arguments, and Volumes**.
- Use **configMapRef and configMapKeyRef** for ConfigMap and **secretRef and secretKeyRef** for Secrets.
- To secure Secrets, use **RBAC and etcd encryption** in Kubernetes.

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**

Kubernetes Persistent Volumes (PV), Persistent Volume Claims (PVC), and Storage Classes

1. Why Do We Need Persistent Volumes?

When a Pod is created in Kubernetes, it gets temporary storage. If that Pod is deleted or crashes, its data is also lost. However, sometimes we need storage that is persistent and keeps data safe. To solve this issue, Kubernetes provides **Persistent Volumes (PV)** and **Persistent Volume Claims (PVC)**.

---

2. What are Persistent Volume (PV) and Persistent Volume Claim (PVC)?

Persistent Volume (PV):

- **PV** is a persistent storage resource in a Kubernetes cluster.
- It can be created manually by an admin or dynamically using **Storage Classes**.
- PVs are independent of Pods, meaning that even if a Pod is deleted, the PV remains.

Persistent Volume Claim (PVC):

- **PVC** is a request made by a user for storage.
- It works similarly to how a Pod requests CPU and RAM.
- When a PVC is created, Kubernetes binds it to a suitable PV.

---

3. PV and PVC Workflow (Lifecycle)

The PV and PVC workflow consists of the following stages:

1. Provisioning (Creating Storage)

Static Provisioning (Manually Creating Storage)

Think of this as an IT admin pre-creating some hard drives (Storage):

- Drive A (100GB) - SSD (Fast)
- Drive B (200GB) - HDD (Standard)
- Drive C (500GB) - Backup

When a developer needs storage, they request a claim for one of the available drives. Similarly, in Kubernetes, an admin pre-creates PVs, and developers submit PVCs for them.

Dynamic Provisioning (Automatically Creating Storage)

Understand this with a Cloud Storage (AWS EBS) example:

- We create three types of storage in AWS:
    1. **Fast** (SSD, High IOPS)
    2. **Slow** (HDD, Standard performance)
    3. **Average** (Balanced SSD)
- We use **StorageClass** for this.
- When a user creates a PVC and selects a StorageClass (e.g., fast-storage), Kubernetes automatically provisions a new PV.

2. Binding (Linking PVC to PV)

- When a user creates a PVC, Kubernetes finds an available PV and binds them together.
- If no suitable PV is found, the PVC remains **unbound** until a new PV becomes available.

3. Using the Storage

- Once a PVC is bound to a PV, it can be used as storage in Pods.
- The Pod mounts it as a volume and can read/write data.

4. Reclaiming (Cleaning Up Storage)

- When a PVC is deleted, what happens to the PV depends on its **Reclaim Policy**:
    - **Retain:** Data remains, and manual cleanup is required.
    - **Delete:** PV and its data are deleted.
    - **Recycle:** Data is wiped, and PV can be reused (Deprecated).

---

4. How to Use PV and PVC in Kubernetes (for static provisioning)?

Create a Persistent Volume (PV) Example (As hostPath storage volume**)**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
```

```
    storage: 5Gi
  accessModes:
   - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: "/mnt/data"
```

Create a Persistent Volume (PV) Example (As nfs storage volume**)**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv01
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain  # "Recycle" is deprecated, use "Retain" or "Delete"
  storageClassName: slow
  nfs:
    path: /mnt/nfs-share1
    server: 172.17.0.2
```

Persistent Volume Claim (PVC) Example

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

```
  storageClassName: standard
```

Using PVC in a Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
      volumeMounts:
        - mountPath: "/data"
          name: storage
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: my-pvc
```

---

5. What is StorageClass?

StorageClass is used in Kubernetes to define different types of storage and enable **Dynamic Provisioning**.

StorageClass Example

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ext4
allowVolumeExpansion: true
```

---

6. Creating a Persistent Volume (PV) with Dynamic Provisioning

To create a PV dynamically, follow these steps:

Step 1: Define a StorageClass

A **StorageClass** is used to specify the type of storage and how it should be provisioned dynamically.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: dynamic-storage
provisioner: kubernetes.io/aws-ebs  # Cloud provider storage
parameters:
  type: gp2
  fsType: ext4
allowVolumeExpansion: true
```

Note:

This does not specify accessModes because it is determined at the PVC level.

The PVC **specifies the required access mode**.

Kubernetes will **provision a PV** that satisfies this request.

Step 2: Create a PVC (Persistent Volume Claim)

When a user creates a PVC and requests storage with a specific StorageClass, Kubernetes automatically provisions a PV.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi  # Requesting 10Gi storage
  storageClassName: dynamic-storage
```

Step-3: The Provisioned PersistentVolume (PV)

Once the PVC is created, Kubernetes dynamically provisions a PV **with the same access mode**:

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: pvc-12345678
spec:
 capacity:
  storage: 10Gi
 volumeMode: Filesystem
 accessModes:
  - ReadWriteOnce  # This is inherited from PVC
 persistentVolumeReclaimPolicy: Delete
 storageClassName: fast-storage
 csi:
  driver: ebs.csi.aws.com
  volumeHandle: vol-xyz123
```

Note: What is the csi: parameter in PV??

The **CSI (Container Storage Interface)** section in the **PersistentVolume (PV)** manifest is used when Kubernetes dynamically provisions storage using a **CSI driver**.

Breaking It Down

- **csi:** → This section tells Kubernetes that the storage is provisioned using the **CSI driver** instead of traditional methods like nfs, hostPath, or awsElasticBlockStore.
- **driver: ebs.csi.aws.com** → Specifies the CSI driver handling the storage. In this case, it's the **AWS EBS CSI driver**, which allows Kubernetes to manage AWS Elastic Block Store (EBS) volumes dynamically.
- **volumeHandle: vol-xyz123** → A unique identifier for the **actual backend storage volume** (in AWS, this would be the EBS volume ID). Kubernetes uses this to attach and detach the volume to/from nodes.
- The **access mode is set automatically** based on the PVC's request.

Step-4 : Use the PVC in a Pod

Once the PVC is created and dynamically bound to a PV, a Pod can use it.

```
apiVersion: v1
kind: Pod
metadata:
  name: dynamic-pod
spec:
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - mountPath: "/app-data"
          name: storage
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: dynamic-pvc
```

---

7. Access Modes of PV

Access Mode	Description
ReadWriteOnce (RWO)	Can be used for Read/Write by a single node.
ReadOnlyMany (ROX)	Can be used for Read by multiple nodes.
ReadWriteMany (RWX)	Can be used for Read/Write by multiple nodes.
ReadWriteOncePod (RWOP)	Can be used for Read/Write by a single Pod only.

---

8. Reclaim Policies

Reclaim Policy	Action
Retain	Data remains, and manual cleanup is required.
Delete	PV and its data are deleted.

Reclaim Policy	Action
Recycle (Deprecated)	Data is wiped, and PV can be reused.

---

9. Expanding Persistent Volume (PV)

If you need more storage, you can expand a PVC **(if allowVolumeExpansion: true is set)**.

Expanding a PVC Example

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: expandable-pvc
spec:
  accessModes:
   - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi  # Previously 5Gi, now increased to 10Gi
  storageClassName: fast-storage
```

Types of Storage Volumes in Kubernetes

Kubernetes provides different types of storage volumes based on various use cases. Some volumes are created directly within the pod, while others require **Persistent Volume (PV)** and **Persistent Volume Claim (PVC)** for management.

---

1. emptyDir (Temporary Storage)

- Created when the pod starts and deleted when the pod is removed.
- **Use Case:** Caching or temporary data storage.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example
spec:
  containers:
    - name: app
```

```
    image: busybox
    volumeMounts:
     - mountPath: /data
       name: temp-storage
  volumes:
   - name: temp-storage
     emptyDir: {}
```

---

2. hostPath (Using Node's File System)

- Uses the host machine's filesystem for storage.
- **Use Case:** Log storage or node-specific data sharing.

Example:

```
apiVersion: v1
kind: Pod
metadata:
 name: hostpath-example
spec:
 containers:
  - name: app
    image: busybox
    volumeMounts:
     - mountPath: /data
       name: host-storage
 volumes:
  - name: host-storage
    hostPath:
     path: /mnt/data
     type: DirectoryOrCreate
```

---

3. nfs (Network File System)

- A network-based storage that multiple pods can share.
- **Use Case:** Shared data or log file storage.

Persistent Volume (PV):

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: nfs-pv
```

```
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  nfs:
    path: /exported/path
    server: 192.168.1.100
```

Persistent Volume Claim (PVC):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 5Gi
```

Pod Example using PVC:

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-pod
spec:
  containers:
    - name: app
      image: busybox
      volumeMounts:
        - mountPath: /data
          name: nfs-storage
  volumes:
    - name: nfs-storage
      persistentVolumeClaim:
        claimName: nfs-pvc
```

---

4. persistentVolumeClaim (PVC for Dynamic Storage)

- Used for dynamically provisioned storage.
- **Use Case:** Stateful applications.

PVC Example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-example
spec:
  accessModes:
   - ReadWriteOnce
  resources:
   requests:
    storage: 10Gi
```

Pod Example using PVC:

```
apiVersion: v1
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
   - name: app
     image: busybox
     volumeMounts:
      - mountPath: /data
        name: my-storage
  volumes:
   - name: my-storage
     persistentVolumeClaim:
       claimName: pvc-example
```

---

5. configMap & secret (Configuration & Secure Storage)

- **Use Case:** Storing configuration files or sensitive data (e.g., API keys, credentials).

ConfigMap Example:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
```

```
data:
  config.json: |
    {
      "setting": "value"
    }
```

Pod using ConfigMap as a volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: app
      image: busybox
      volumeMounts:
        - mountPath: /etc/config
          name: config-storage
  volumes:
    - name: config-storage
      configMap:
        name: app-config
```

---

6. csi (Container Storage Interface - Cloud Storage)

- Used for cloud storage solutions like AWS EBS, Google Persistent Disk, etc.

PVC for CSI Storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: gp2
```

Pod using CSI-based PVC:

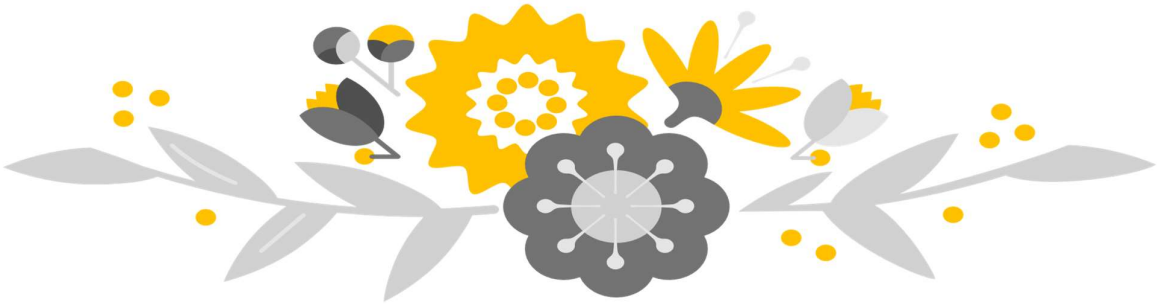
```
apiVersion: v1
kind: Pod
metadata:
  name: csi-pod
spec:
  containers:
    - name: app
      image: busybox
      volumeMounts:
        - mountPath: /data
          name: csi-storage
  volumes:
    - name: csi-storage
      persistentVolumeClaim:
        claimName: csi-pvc
```

Do We Need to Create a PV for Every Storage Type?

Storage Type	PV Required?	PVC Required?	StorageClass Required?
emptyDir	❌	❌	❌
hostPath	❌	❌	❌
nfs	✅	✅	✅ (for dynamic provisioning)
aws-ebs	✅ (for static provisioning)	✅	✅
csi	❌ (if dynamically provisioned)	✅	✅

Conclusion of the topic:

- If you are using emptyDir or hostPath, **you don't need a PV**.
- If you are using nfs, csi, or **cloud-based storage**, **you need PV and PVC**.
- **Dynamically provisioned storage (e.g., csi, cloud storage)** does not require manually creating a PV; PVC alone is enough.



Kubernetes Resource Quota (A Simple strong Student Guide)

What is Resource Quota?

Resource Quota is a method in Kubernetes to **control resource usage** at the project level (**Namespace** level).

There are two types of Resource Quota:

1. **Computing Level Quota** – Controls CPU and RAM (Memory) usage.
2. **Object Level Quota** – Limits the number of **Pods, Deployments, Services, PVCs**, etc.

Why is Resource Quota Applied at the Namespace Level?

In Kubernetes, every resource is created inside a **Namespace**. A **Namespace** defines the boundary for project resources. So, **Resource Quota is always set at the Namespace level** to manage resource limits effectively.

Why is Resource Quota Important?

Imagine a **Kubernetes Cluster** where multiple teams work together. If one team uses **too many resources**, other teams **may not get enough**. So **Resource Quota** prevents this by ensuring fair distribution.

Problem: One team uses all the resources.

Solution: Quota limits resource usage per team.

Example:

- Team A (Amazon) gets limited resources.
- Team B (Facebook) gets its own fair share.

Namespaces and Resource Quota

In Kubernetes, a **Namespace** divides a cluster into smaller sections, providing **logical isolation** at the project level. When a **Resource Quota** is applied to a Namespace, it ensures that all resources within that Namespace adhere to the defined limits. However, it does **not** impact other projects, as each operates in its own separate Namespace.

For example, if a **cluster has a total capacity of 32GB RAM and 16 CPU Cores**, the administrator can allocate resources as follows:

- **Team A** gets **20GB RAM and 10 CPU Cores**.
- **Team B** gets **10GB RAM and 4 CPU Cores**.
- **2GB RAM and 2 CPU Cores** are reserved for future use.

Example:

Team A → Namespace = amazon
Team B → Namespace = facebook
Each team can create resources **only within their assigned Resource Quota**, ensuring fair distribution of cluster resources.

Types of Resource Quota

As discussed earlier, there are two main types of Resource Quota:

1. **Computing Quota** – Controls CPU and Memory (RAM).
2. **Object Quota** – Limits the number of Kubernetes objects like **Pods, Deployments, Services, PVCs**, etc.

Type-1 (Object Resource Quota) – Kubernetes object Resources

How Does Object Quota Work?

A **Object Quota** sets a **limit on the maximum resources** that can be used in a **Namespace**. It controls computing resource like **CPU, RAM, storage, and the number of objects (Pods, Services, Deployments, etc.)**.

Example: Setting a Object Quota for the "AMAZON" Namespace

```
apiVersion: v1
kind: ResourceQuota
```

```
metadata:
  name: team-a-quota
spec:
  hard:
    pods: "10"            # Maximum 10 Pods
    services: "5"         # Maximum 5 Services
    deployments: "10"     # Maximum 10 deployments
    secrets: "15"         # Maximum 15 Secrets
    requests.cpu: "2"  # Max 2 CPU requests
    requests.memory: "4Gi"  # Max 4GB RAM requests
    limits.cpu: "4"               # Max 4 CPU usage
    limits.memory: "8Gi"       # Max 8GB RAM usage
    requests.storage: "100Gi"          # Max 100GB Storage
    persistentvolumeclaims: "5"      # Max 5 PVCs
```

What Does This Mean for the Namespace? When this Resource Quota is applied:

- A maximum of 10 Pods can be created.
- CPU and Memory usage cannot exceed the defined limits.
- Storage usage cannot exceed 100GB.
- Only 5 PVCs (Persistent Volume Claims) can be created.
- Only 15 Secrets can be stored.
- Only 10 Deployments are allowed.

What if limit try to crossed (How Does Resource Quota Work?)

- Admin sets a Resource Quota inside a Namespace.
- Teams create resources (Pods, Services) inside the Namespace.
- Kubernetes checks if the request follows the Quota limits.
- If the request exceeds the limit, Kubernetes blocks it.

Example: If team-a tries to create an **11th Pod**, it will get an **error:**

Error: exceeded quota: team-a-quota, requested: pods=1, used: pods=10, limited: pods=10

Meaning: Cannot create more than 10 Pods**.**

Type-2 (Compute Resource Quota) – CPU & Memory Limits

Compute Quota is a type of Resource Quota that specifically controls CPU and Memory (RAM) usage in a Namespace. It ensures that no team or application overuses computing resources, keeping the cluster balanced and efficient.

Resource Quota controls CPU and Memory in two ways:

Requests – The minimum CPU and RAM a Pod needs.
Limits – The maximum CPU and RAM a Pod can use.

How Does Compute Quota Work? Step by Step

1. **Admin sets a Compute Quota in a Namespace.**
2. **Teams create Pods and other resources.**
3. **Kubernetes checks if CPU and Memory requests are within limits.**
4. **If a request exceeds the limit, Kubernetes blocks it.**

Example: Setting a Compute Quota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
spec:
  hard:
    requests.cpu: "2"  # Minimum 2 CPU required
    requests.memory: "4Gi"  # Minimum 4GB RAM required
    limits.cpu: "4"  # Maximum 4 CPU allowed
    limits.memory: "8Gi"  # Maximum 8GB RAM allowed
```

What This Means:

- Minimum CPU a Pod can request: **Up to 2 CPUs in total.**
- Minimum RAM a Pod can request: **Up to 4GB in total.**
- Maximum CPU a Pod can use: **Up to 4 CPUs in total.**
- Maximum RAM a Pod can use: **Up to 8GB in total.**

Important:

- **requests.cpu** & **requests.memory** → Define minimum guaranteed resources for Pods.
- **limits.cpu** & **limits.memory** → Define maximum allowed usage for Pods.

Meaning:

- **CPU Requests** = Minimum CPU allocation.
- **CPU Limits** = Maximum CPU allocation.

What is compute measurement unit:

1 CPU = 1000m (millicores)
1 GB RAM = 1024Mi

What Happens If Limits Are Exceeded?

Example: If a Pod requests more than 2 CPUs or tries to use more than 4 CPUs, Kubernetes will block the request and show an error.

Error: exceeded quota: compute-quota, requested: cpu=3, used: cpu=2, limited: cpu=2

Meaning: The request exceeds the allowed CPU limit in this Namespace.

Priority-Based Quota – High-Priority Pods

Sometimes, **important Pods** need extra resources. For this, **Priority-Based Quota** is used.

Why is Priority-Based Quota Used?

Imagine you are running a **shared Kubernetes cluster** where multiple teams and applications use the same resources. Some applications are more important than others. For example:

- A **payment service** must always work, even if resources are low.
- A **report generation service** can wait if needed.

To **ensure that critical applications always get the resources they need**, **Priority-Based Quota** is used.

How It Works (In Simple Terms)

- **Admins assign different priority levels** (High, Medium, Low).
- **Each priority level gets a fixed amount of resources (CPU, RAM, Pods).**
- **High-priority applications always get their reserved resources first.**
- **If resources are low, low-priority applications may be stopped or delayed.**

Real-Life Example

Think of it like **boarding a flight**:

1. **High-Priority (Business Class)** – Always gets a seat, even if the flight is full.
2. **Medium-Priority (Premium Economy)** – Gets a seat if available.
3. **Low-Priority (Economy Class)** – May be denied a seat if the flight is overbooked.

Similarly, **Kubernetes ensures high-priority applications always run**, while low-priority applications may be limited if resources are tight.

Example of Priority-Based Quota (Commented for Easy Understanding)

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high          # Resources reserved for high-priority applications
  spec:
    hard:
      cpu: "1000"            # High-priority apps can use up to 1000 CPUs
      memory: "200Gi"        # Can use up to 200GB RAM
      pods: "10"             # Only 10 high-priority pods allowed
    scopeSelector:
```

```
    matchExpressions:
    - operator: In
      scopeName: PriorityClass
      values: ["high"]          # Applied only to high-priority pods

- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-medium           # Resources reserved for medium-priority applications
  spec:
    hard:
      cpu: "10"                 # Medium-priority apps can use up to 10 CPUs
      memory: "20Gi"            # Can use up to 20GB RAM
      pods: "10"                # Only 10 medium-priority pods allowed
    scopeSelector:
      matchExpressions:
      - operator: In
        scopeName: PriorityClass
        values: ["medium"]      # Applied only to medium-priority pods

- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-low              # Resources reserved for low-priority applications
  spec:
    hard:
      cpu: "5"                  # Low-priority apps can use up to 5 CPUs
      memory: "10Gi"            # Can use up to 10GB RAM
      pods: "10"                # Only 10 low-priority pods allowed
    scopeSelector:
      matchExpressions:
      - operator: In
        scopeName: PriorityClass
        values: ["low"]         # Applied only to low-priority pods
```

Meaning:

- High-Priority Pods get their own limits.
- Up to 10 High-Priority Pods are allowed.
- High-priority apps always get resources first.
- Low-priority apps may be paused if resources are low.
- Ensures fair and efficient resource usage in Kubernetes.

How to Use Priority-Based Resource Quota in a Deployment or Pod?

To apply **Priority-Based Resource Quota** to a **Deployment or Pod**, you need to use **PriorityClass**.

- **Use PriorityClass in a Deployment**
  Now, apply **PriorityClass** in a **Deployment** to ensure it gets the right resource quota.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: high-priority-app
  namespace: example-namespace
spec:
  replicas: 2
  selector:
    matchLabels:
      app: important-app
  template:
    metadata:
      labels:
        app: important-app
    spec:
      priorityClassName: high-priority        # Assigning high-priority
      containers:
      - name: app-container
        image: nginx
        resources:
          requests:
            cpu: "500m"                       # Requesting 0.5 CPU
            memory: "512Mi"                   # Requesting 512MB RAM
          limits:
            cpu: "1"                          # Max usage 1 CPU
            memory: "1Gi"                     # Max usage 1GB RAM
```

- **Use PriorityClass in a Pod (Without Deployment)**
  If you want to use **PriorityClass** in a **Pod directly**, use this:

```
apiVersion: v1
kind: Pod
metadata:
  name: medium-priority-pod
  namespace: example-namespace
spec:
  priorityClassName: medium-priority  # Assigning medium-priority
  containers:
  - name: app-container
    image: nginx
    resources:
      requests:
        cpu: "200m"
        memory: "256Mi"
      limits:
        cpu: "500m"
        memory: "512Mi"
```

Kubernetes Updated on Resource Quota (Things you have to know)

Extended Resource Quota

Since Kubernetes 1.10, **Resource Quota** has been extended to support **special hardware resources** like **GPU, FPGA, and other custom hardware**.

For example, if you want to limit the number of GPUs used in a **Namespace**, you can define a quota like this:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gpu-quota
spec:
  hard:
        requests.cpu: "4" requests.memory: "16Gi"
        requests.nvidia.com/gpu: "4"  # Maximum 4 GPUs can be used in namespace
```

Meaning:

- The Namespace where this quota is applied **cannot use more than 4 GPUs**.
- This feature is useful for workloads that require **machine learning, AI processing, or high-performance computing** while ensuring fair resource distribution.

Storage Resource Quota

Storage Resource Quota is part of the **ResourceQuota object**, but it specifically deals with **storage-related limits**, unlike standard **object quotas** that manage Kubernetes objects (like Pods, Services, and PVC counts).

Let's break it down clearly:

Where does Storage Resource Quota fit?

- **Object Quota:** Controls **how many** Kubernetes objects (Pods, PVCs, Services, etc.) can exist.
- **Compute Quota:** Controls **how much** CPU, Memory, and special resources (like GPU) can be used.
- **Storage Quota:** Controls **how much storage (Persistent Volume Claims - PVCs)** can be used.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
  namespace: my-namespace       # Replace with your actual namespace
spec:
  hard:
    pods: "10"                  # Max 10 Pods allowed
    services: "5"               # Max 5 Services allowed
    persistentvolumeclaims: "5" # Max 5 PVCs allowed
    requests.cpu: "2"           # Max 2 CPU requests
    requests.memory: "4Gi"      # Max 4GB RAM requests
    limits.cpu: "4"             # Max 4 CPU usage
```

```
limits.memory: "8Gi"          # Max 8GB RAM usage
requests.storage: "100Gi"     # Max 100GB total storage usage
configmaps: "10"              # Max 10 ConfigMaps allowed
secrets: "15"                 # Max 15 Secrets allowed
```

Kubernetes Resource Quota Assignment – Professional Quiz

Instructions:

This quiz is designed to test your understanding of **Kubernetes Resource Quotas** (Computing Quota & Object Quota) in real-world scenarios. Read each question carefully and think about the best possible answer before proceeding.

---

1. Understanding Namespace-Level Resource Quotas

- Your company manages multiple teams working on different projects within a shared Kubernetes cluster. To prevent one team from consuming excessive resources, you decide to implement Resource Quotas.
- **Why is Resource Quota always applied at the Namespace level instead of the cluster level? How does this help in managing resources effectively?**

---

2. Handling Resource Limits for Pods

- A development team is working in a Namespace where the following quota is applied: pods: "10" Now when They attempt to deploy an 11th Pod.
- **What will happen in this case? How does Kubernetes enforce this quota limit?**

---

3. Compute Resource Quota Enforcement

- A Kubernetes administrator has set up the following Compute Resource Quota for a Namespace:

```
requests.cpu: "4"
limits.cpu: "8"
requests.memory: "8Gi"
limits.memory: "16Gi"
pods: "5"
```

- Currently, the team has already created **5 Pods**, but there is still CPU and Memory available.
- **Can they create an additional Pod? Why or why not?**

---

4. Monitoring and Troubleshooting Resource Quotas

- As a Kubernetes administrator, you need to monitor the current resource usage in a specific Namespace and identify whether it is approaching its quota limits.
- **Which kubectl command would you use to check the existing Resource Quota and its current usage?**

---

5. Requests vs. Limits in Compute Quotas

- Your team is configuring a **Compute Resource Quota** for their Namespace, but they are confused about the difference between requests and limits.
- **In a Kubernetes Compute Quota, what is the difference between "requests" and "limits" for CPU and Memory? Why is this distinction important?**

---

6. Object Count Quota Enforcement

- A Namespace has the following Object Quota applied:

  pods: "10"

  services: "5"

  secrets: "15"
- Currently, the team has deployed **9 Pods, 5 Services, and 10 Secrets**.
- **How many more Pods, Services, and Secrets can they create before hitting the quota limit?**

---

7. Handling Storage Resource Quotas

- A team is deploying applications that require Persistent Volumes. Their Namespace has the following Storage Resource Quota:

  requests.storage: "500Gi"

  persistentvolumeclaims: "10"
- Currently, they have **6 PersistentVolumeClaims (PVCs)**, each requesting **50Gi** of storage.
- **Can they create 5 more PVCs, each requesting 50Gi of storage? Why or why not?**

---

8. Impact of Resource Quotas on High-Priority Pods

- Your team is deploying High-Priority Pods in a cluster that already has a strict Resource Quota applied. However, the High-Priority Pods are getting rejected due to quota restrictions.
- **How can you configure Resource Quotas to ensure High-Priority Pods get the necessary resources even in a constrained environment?**

---

9. Debugging Quota-Related Issues

- A developer reports that their Pod creation request is failing due to a quota limit error. You need to investigate the issue.
- **Which Kubernetes command would you use to check recent quota-related errors and understand why the request is failing?**

---

10. Scaling and Resource Quotas

- Your team is planning to scale an application by adding more Pods. However, they discover that the Namespace has hit its Resource Quota limits.
- **What are the possible ways to allow more Pods to be created while maintaining fair resource distribution among teams?**

Essential Kubernetes Commands for Managing Resource Quotas

- **Create a Compute Resource Quota (CPU & Memory) from Command Line**

```
kubectl create resourcequota compute-quota --namespace=<namespace> \
  --hard=requests.cpu=2,limits.cpu=4,requests.memory=4Gi,limits.memory=8Gi
```

Creates a quota that limits CPU and Memory usage in the specified Namespace.

- **Create an Object Count Quota (Pods, Services, PVCs) from Command Line**

```
kubectl create resourcequota object-quota --namespace=<namespace> \
  --hard=pods=10,services=5,persistentvolumeclaims=5,secrets=15
```

Creates a quota that limits the number of Kubernetes objects like Pods, Services, PVCs, and Secrets.

- **List All Resource Quotas in a Namespace**

```
kubectl get resourcequota --namespace=<namespace>
```

Displays all existing Resource Quotas in the specified Namespace.

- **View Detailed Information About a Specific Resource Quota**

kubectl describe resourcequota <quota-name> --namespace=<namespace>

Shows quota limits, current resource usage, and remaining capacity.

- **Check Node-Level Resource Quota**

```
kubectl describe node <node-name>
```

Shows CPU, memory, and other allocated resources at the **node level**.

- **Check the Current Resource Usage in a Namespace**

```
kubectl top pod --namespace=<namespace>
```

Displays real-time CPU and memory usage of Pods.

- **Apply a New Resource Quota from a YAML File**

  kubectl apply -f <resource-quota.yaml> --namespace=<namespace>

Creates or updates a Resource Quota from a YAML definition.

- **Edit an Existing Resource Quota**

  kubectl edit resourcequota <quota-name> --namespace=<namespace>

Opens the quota configuration for editing in a text editor.

- **Delete a Resource Quota**

  kubectl delete resourcequota <quota-name> --namespace=<namespace>

Removes the Resource Quota from the Namespace.

- **Check Recent Quota-Related Errors (Events)**

  kubectl get events --namespace=<namespace>

Lists all recent events, including Resource Quota violations.

- **Verify Resource Requests and Limits for a Pod**

  kubectl get pod <pod-name> -o yaml --namespace=<namespace>

Displays detailed YAML output to check if the Pod is within quota limits.

- **Simulate a Quota Violation Before Applying a Deployment**

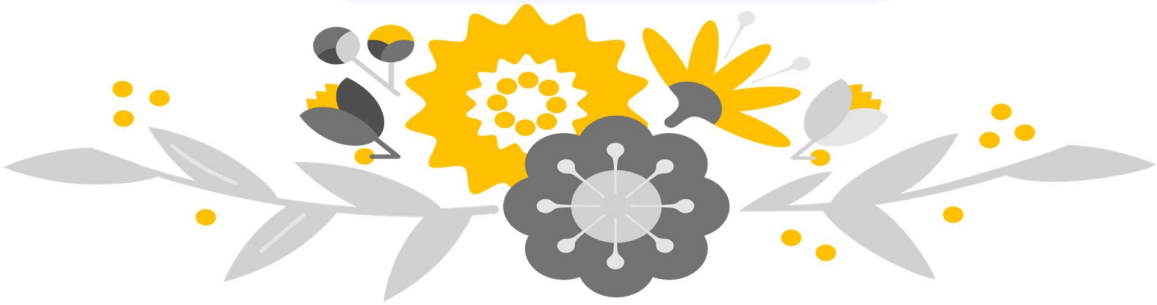
  kubectl create -f <deployment.yaml> --dry-run=server --namespace=<namespace>

Checks if a new deployment would exceed the quota without actually applying it.

- **Increase Resource Quota Limits (By Applying an Updated YAML)**

  kubectl apply -f updated-quota.yaml --namespace=<namespace>

Allows increasing limits when more resources are required.



Kubernetes Sidecar Containers & BusyBox (A Simple Student Guide)

Point 1: What is a Kubernetes Sidecar Container?

1. What is a Sidecar Container?

Sometimes, along with the main container, we need an extra container to help it. This extra container is called a **Sidecar Container**. It runs alongside the main container and provides additional features to improve performance and functionality of main container without losing main containers quality and performance.

Think of it like this:
Main Container = Your application (e.g., a web server)
Sidecar Container = A helper that manages logs, monitoring, or syncing data

2. Why is a Sidecar Container needed?

If your main container is designed to do only one job but needs extra features, instead of modifying it, you can add a Sidecar Container.

Example: If a container is running a web app, but you need to store log files, you can use a separate Sidecar Container for log storage.

3. Use Cases of Sidecar Containers

- **Log Management** – To store and transfer logs
- **Security** – To enhance the security of the main container
- **Proxy Server** – To optimize networking
- **Data Backup & Sync** – To send data to cloud or storage
- **Monitoring** – To check the health of the container

4. Benefits of Sidecar Containers

- You can add new features without modifying the main container
- It improves microservices architecture
- Helps create scalable and repeatable solutions

- Makes logging, monitoring, and security easier

5. How to create and use a Sidecar Container?

You need a Pod YAML file with two containers:

1**: Main Container** (Nginx Web Server)

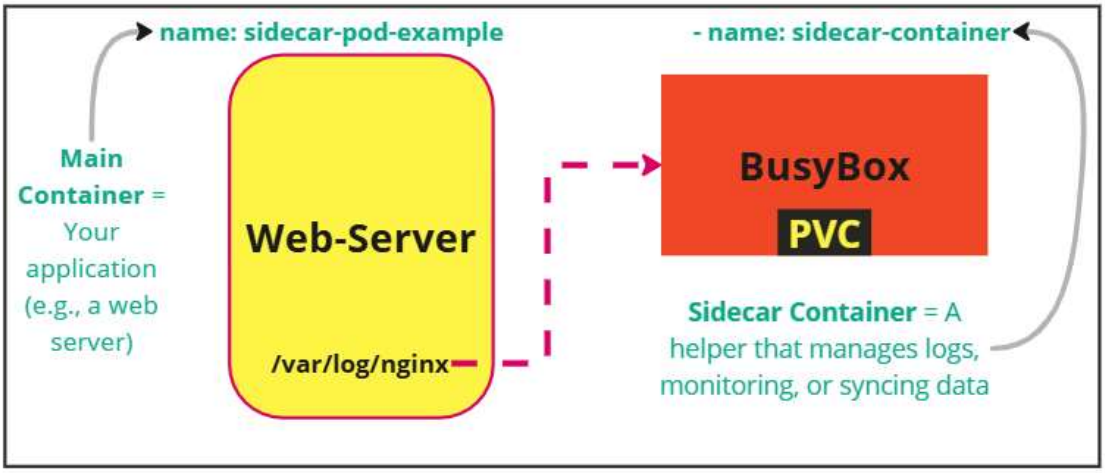
2: **Sidecar Container** (BusyBox Log Forwarder)

Definition

Sometimes, along with the main container, we need an extra container to help it. This extra container is called a **Sidecar Container**.

It runs alongside the main container and provides additional features to improve performance and functionality of main container without losing main containers quality and performance.

Diagram



Manifests

```
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-pod-example
spec:
  containers:
    - name: main-container
      image: nginx
      volumeMounts:
        - name: shared-logs
          mountPath: /var/log/nginx

    - name: sidecar-container
      image: busybox
      command: ["/bin/sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep 10; done"]
      volumeMounts:
        - name: shared-logs
          mountPath: /var/log/nginx

  volumes:
    - name: shared-logs
      persistentVolumeClaim:
        claimName: pvc2
```

The **main container (nginx)** writes logs to /var/log/nginx.

The **sidecar container (busybox)** reads the same logs from /var/log/nginx and prints them every 10 seconds.

pvc volume "pvc2" is using here to store data persistently

Note:

Let's understood the line below in **command** section, that runs inside the busybox container in your

Kubernetes Pod:

command: ["/bin/sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep 5; done"]

Breaking it Down:

1. **/bin/sh** → This starts a shell (sh) inside the container.
2. **-c** → This tells the shell to execute the following command as a script.
3. **while true; do ... done** → This is an infinite loop that keeps running continuously.
4. **cat /var/log/nginx/access.log** → Reads and prints the contents of the Nginx access log.
5. **sleep 5** → Waits for 5 seconds before running the next iteration of the loop.

What This Does in the Pod:

- The busybox container keeps reading (cat) the Nginx access logs every 5 seconds.
- This is useful for debugging because it allows you to see real-time logs.
- The loop ensures the container doesn't exit immediately after running the command once.

How does this work?

- **Main Container (Nginx)** serves the website
- **Sidecar Container (BusyBox)** continuously checks and stores Nginx logs

To apply this in Kubernetes run the manifests file:
kubectl apply -f sidecar.yaml

To check logs:
kubectl logs -f sidecar-example -c sidecar-container

Point 2: What is a BusyBox Image?

BusyBox is a very lightweight **Linux-based utility tool** that includes basic Unix commands like ls, cp, mv, echo, cat, grep, wget, etc. without any heavy program or utility tools. It is called "**The Swiss Army Knife of Embedded Linux**" because it provides multiple Unix commands in a single binary. It has 1mb size max. so this is lightweight and fast.

Use Cases of BusyBox Image

1. Testing & Debugging

You can use BusyBox for testing in Docker/Kubernetes:

```
docker run -it busybox sh
```

This opens a lightweight Linux shell where you can run Unix commands.

2. Getting Shell Access in a Container

```
kubectl run -it my-busybox --image=busybox -- sh
```

This creates a temporary pod where you can run commands inside the container.

3. Checking Network Connectivity

```
kubectl run busybox --image=busybox --restart=Never -- ping google.com
```

This checks if your Kubernetes pod is connected to the internet.

4. Data Processing & File Operations

```
kubectl run my-busybox --image=busybox --restart=Never -- cat /var/log/app.log
```

This reads log files inside the container.

5. Using BusyBox as a Sidecar Container

BusyBox can be used as a Sidecar to process data from another container:

```
containers:
- name: sidecar-container
  image: busybox
  command: ["/bin/sh", "-c", "while true; do cat /var/log/app.log; sleep 10; done"]
```

This continuously monitors the /var/log/app.log file.

Benefits of BusyBox Image

- **Lightweight** – Only about **1MB** in size
- **Fast** – Loads quickly and uses fewer resources
- **Ideal for Embedded Systems** – Works well in low-memory and low-CPU environments
- **Complete Unix Toolset** – Includes sh, wget, ping, echo, cat, vi, etc.

Point 3: FAQs (Frequently Asked Questions)

1. Can we use the Main Container for Sidecar tasks?

Defiantly we can, it is possible, but it is not a good practice.

Problems

- If the **Main Container** (e.g., Nginx, PostgreSQL) handles logging, monitoring, or networking, maintenance becomes difficult.

- The **Single Responsibility Principle (SRP)** suggests that each container should do only one job.
- Updating the Main Container becomes harder if it performs multiple tasks.

2. Why is a Sidecar Container better?

- It **does not affect the Main Container**.
- The **application and Sidecar Container can be developed separately**.
- If you only need to change **log forwarding or monitoring**, you can update the Sidecar instead of modifying the Main Container.
- This approach is **better for Microservices and DevOps architectures**.

3. Why use BusyBox instead of other Base OS images?

1. Lightweight and Fast

- **BusyBox** is **less than 1MB** in size, while Base OS images (Ubuntu, Alpine, Debian) can be **100MB+**.
- A **smaller image means faster loading, less storage, and lower RAM usage**.

2. Complete Unix Command Set

- BusyBox includes essential Unix tools (ls, cat, ping, wget, grep, sh).
- We don't need a full Base OS, just basic commands.

3. Secure and Minimal

- BusyBox **only includes necessary commands**, reducing security risks.
- Other Base OS images may have unnecessary packages, increasing security vulnerabilities.

4. Best for Kubernetes and Containers

- BusyBox is a **scratch-based image** (basic and minimal), making it **perfect for containerized applications**.
- It is **ideal for Sidecar Containers, Debugging Pods, and Testing**.

4. Can we use alternatives instead of BusyBox?

Defiantly we can, but BusyBox is **the best and simplest option**. Why because of its size and minimalist nature with complete required tools options.

Alpine Linux – Lightweight (~5MB) but heavier than BusyBox

Debian Slim – Secure but larger (~20MB+)

Ubuntu Minimal – Secure and stable but heavy (~29MB+)

Scratch – **Zero-size image** (no built-in tools) but not good for debugging

If you need **Shell Access and Basic Commands in a lightweight container, BusyBox is the best choice**!

My Conclusion

- **Sidecar Containers** help improve the functionality of the Main Container without modifying it.
- They are used for **logging, monitoring, proxy, data sync, and security**.
- Kubernetes **Sidecar Containers** are defined in a **YAML file** with shared **volumes**.
- **BusyBox** is a **lightweight** Linux-based image, ideal for **debugging, networking, file processing, and Sidecar Containers**.
- It is **fast, minimal, and secure**, making it the **best choice for Kubernetes and Docker environments**.
- Using the **Sidecar Pattern** keeps containers modular, scalable, and easy to manage.

Now have some practice

Practice-1: Create an Nginx web server pod with a sidecar container that prints logs every 2 seconds. This will demonstrate how a sidecar container can be used for log processing in Kubernetes.

Answer:

Step-1 Here's the YAML configuration to create a **Pod** with two containers:

1. **Nginx container** - Runs the web server and stores logs in /var/log/nginx.
2. **BusyBox sidecar container** - Reads and prints the logs from /var/log/nginx every 2 seconds.

   # vim sidecar-pod-example.yml

   ```
   apiVersion: v1
   kind: Pod
   metadata:
   ```

```
  name: nginx-sidecar-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: log-volume
      mountPath: /var/log/nginx
  - name: log-sidecar
    image: busybox
    command: ["/bin/sh", "-c"]
    args: ["while true; do cat /var/log/nginx/access.log; sleep 2; done"]
    volumeMounts:
    - name: log-volume
      mountPath: /var/log/nginx
  volumes:
  - name: log-volume
    emptyDir: {}
```

Manifests yaml file Explanation:

- **Shared Volume (emptyDir)**: Both containers share /var/log/nginx using an emptyDir volume.
- **Nginx Container**: Writes access logs to /var/log/nginx/access.log.
- **BusyBox Sidecar**: Reads the logs from the shared volume and prints them every 2 seconds.

Now question may come, "If **logs are already stored in a volume**, you can view them directly from the volume mount point. So, **why do we need a sidecar container to print logs using:**

args: ["while true; do cat /var/log/nginx/access.log; sleep 2; done"]

So answer would be "You are right If you just need to view logs, use kubectl logs or kubectl exec but If you need to automatically process, filter, stream, or forward logs, a sidecar container is useful."

Step-2: See the logs from sidecar

 # kubectl exec -it nginx-sidecar-pod -c nginx-container -- cat /var/log/nginx/access.log

```
root@control:~# kubectl exec -it nginx-sidecar-pod -c nginx-container -- cat /var/log/nginx/access.log
127.0.0.1 - - [03/Mar/2025:07:39:33 +0000] "GET / HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
```

Real-World Projects (Sidecar Containers in Kubernetes:)

Assignment: Implementing Sidecar Containers in Kubernetes

Objective

The goal of this assignment is to understand and implement the **sidecar container pattern** in Kubernetes. Each project demonstrates how a sidecar container enhances the primary application by providing additional functionalities like log forwarding, database backup, monitoring, and dynamic configuration updates.

Project 1: Log Forwarding with Fluentd

Use Case

In microservices environments, efficient log management is crucial. This project implements a sidecar pattern where an **Nginx web server** writes logs, and a **Fluentd container** collects and forwards them to an external system.

Kubernetes Pod Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-fluentd-pod  # Pod name
spec:
  volumes:
    - name: log-volume
      emptyDir: {}
  containers:
    - name: nginx  # Primary container: Nginx (Web Server)
      image: nginx
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
    - name: fluentd  # Sidecar container: Fluentd (Log Processor)
      image: fluent/fluentd
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
```

How It Works?

- Nginx writes logs to /var/log/nginx.
- Fluentd reads these logs and forwards them to an external system.

Service Manifest (ClusterIP):

```
apiVersion: v1
kind: Service
metadata:
  name: fluentd-service
spec:
  selector:
    app: nginx-fluentd
  ports:
    - protocol: TCP
      port: 24224
      targetPort: 24224
  type: ClusterIP
```

Access Fluentd Logs:

```
kubectl port-forward svc/fluentd-service 24224:24224
```

Project 2: Automated Database Backup

Use Case

Automated backups are essential for databases running inside Kubernetes. This project introduces a **sidecar container** that periodically backs up **MySQL database data** to a shared volume.

Kubernetes Pod Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: mysql-backup-pod
spec:
  volumes:
    - name: db-backup
      nfs:
        server: <NFS_SERVER_IP>
        path: /exported/path
  containers:
    - name: mysql
      image: mysql:5.7
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "root"
      volumeMounts:
        - name: db-backup
          mountPath: /var/backups
    - name: backup-sidecar
      image: busybox
      volumeMounts:
        - name: db-backup
          mountPath: /var/backups
      command: ["/bin/sh", "-c", "while true; do cp -r /var/lib/mysql /var/backups; sleep 3600;
done"]
```

How It Works?

- MySQL stores data in its primary volume.
- The sidecar runs an automated backup process every hour.

Service Manifest (ClusterIP):

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-backup-service
spec:
  selector:
    app: mysql-backup
  ports:
    - protocol: TCP
      port: 3306
      targetPort: 3306
  type: ClusterIP
```

Access MySQL Database:

```
kubectl port-forward svc/mysql-backup-service 3306:3306
```

Project 3: Application Monitoring with Prometheus

Use Case

Observability is key in modern applications. This project integrates **Prometheus as a sidecar container** to monitor application metrics in real-time.

Kubernetes Pod Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-monitoring-pod
spec:
  containers:
    - name: my-app
      image: myapp:latest
      ports:
        - containerPort: 8080
    - name: prometheus-exporter
      image: prom/prometheus
      ports:
        - containerPort: 9090
```

How It Works?

- The application runs on port 8080.
- The Prometheus sidecar scrapes and exposes application metrics.

Service Manifest (NodePort):

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus-exporter-service
spec:
  selector:
    app: app-monitoring
  ports:
    - protocol: TCP
      port: 9090
      targetPort: 9090
      nodePort: 30090
  type: NodePort
```

Access Prometheus Dashboard:

```
kubectl get svc prometheus-exporter-service
```

Then, open:

```
http://<NODE_IP>:30090/metrics
```

Project 4: Dynamic Configuration Updates

Use Case

Many applications require **dynamic configuration updates** without restarting. This sidecar container **fetches updates periodically** and updates the config file.

Kubernetes Pod Manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: config-sync-pod
spec:
  volumes:
    - name: config-volume
      emptyDir: {}
  containers:
    - name: my-app
      image: my-app:latest
      volumeMounts:
        - name: config-volume
          mountPath: /etc/app/config
    - name: config-updater
      image: busybox
      volumeMounts:
        - name: config-volume
          mountPath: /etc/app/config
      command: ["/bin/sh", "-c", "while true; do echo 'updated' > /etc/app/config/config.yaml;
sleep 60; done"]
```

How It Works?

- The application reads configuration from /etc/app/config/config.yaml.
- The sidecar updates this file every 60 seconds.

Service Manifest (ClusterIP):

```
apiVersion: v1
kind: Service
metadata:
  name: config-sync-service
spec:
  selector:
    app: config-sync
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
  type: ClusterIP
```

Access Configuration Update Logs:

```
kubectl logs -f config-sync-pod -c config-updater
```

Deployment & Testing

Apply All Manifests:

```
kubectl apply -f fluentd-service.yaml
kubectl apply -f mysql-backup-service.yaml
kubectl apply -f prometheus-exporter-service.yaml
kubectl apply -f config-sync-service.yaml
```

Check Running Services:

```
kubectl get svc
```

Delete Any Service If Needed:

```
kubectl delete -f fluentd-service.yaml
kubectl delete -f mysql-backup-service.yaml
kubectl delete -f prometheus-exporter-service.yaml
kubectl delete -f config-sync-service.yaml
```

Now, verify that all your **sidecar containers** are correctly configured and functional!

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**

Service, kube-proxy & CNI (Easy Kubernetes Guide)

What is a service ?

A **Service** in Kubernetes helps expose an application running in your cluster by providing a single, stable network endpoint. This ensures that even if your application runs in multiple Pods (which may change dynamically), clients can always access it reliably.

Why Do We Need a Service?

1. **Pods Are Temporary (Ephemeral)** – Kubernetes creates and destroys Pods as needed, so their IP addresses change dynamically.
2. **Consistent Access** – If an application has multiple backend Pods, a Service provides a fixed way to reach them.
3. **Load Balancing** – Services distribute traffic among multiple Pods automatically.

How a Service Works?

- A **Service** groups a set of Pods using **labels and selectors**.
- It gives them a **stable network identity (ClusterIP) instead of IP**.
- Clients can use this identity to interact with the backend Pods.

For example, assume we have a **backend application** running in 3 Pods. Instead of connecting to individual Pods, we create a Service that directs traffic to all healthy backend Pods.

Types of Services

1. ClusterIP (Default)

- Exposes the Service only within the cluster.
- Not accessible from outside.
- Example: Internal communication between microservices.

Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Here, all Pods with label **app=my-backend** will be reachable at port **80** via the Service.

---

2. NodePort

- Exposes the Service on a specific port of each Node.
- Accessible from outside the cluster using **NodeIP:NodePort**.
- Useful for debugging or basic external access.

Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-backend
```

```
  ports:
   - port: 80
     targetPort: 9376
     nodePort: 30007  # This port is accessible on every node
```

If the cluster node has IP **192.168.1.10**, then you can access the service at:

http://192.168.1.10:30007

---

3. LoadBalancer

- Used when running Kubernetes on cloud providers (AWS, GCP, Azure).
- It provisions an **external load balancer** to distribute traffic.
- Suitable for public-facing applications.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
    app: my-backend
  ports:
    - port: 80
      targetPort: 9376
```

- If the cloud provider assigns **192.0.2.127** as the external IP, users can access it via:

http://192.0.2.127

---

4. ExternalName

- Maps a Service to an external DNS name.
- No traffic is routed inside the cluster; instead, it resolves to an external resource.

  Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-database
spec:
  type: ExternalName
  externalName: db.example.com
```

When applications inside the cluster query **my-database**, they get redirected to **db.example.com**.

Other Important Concepts

1. Multi-Port Services

- Services can expose multiple ports for different functionalities.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-backend
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

This allows HTTP traffic on **port 80** and HTTPS traffic on **port 443**.

---

2. Headless Services

- A Service without a ClusterIP (clusterIP: None).
- Used for direct Pod-to-Pod communication (e.g., databases).

```
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
spec:
  clusterIP: None
  selector:
    app: my-backend
  ports:
    - port: 80
      targetPort: 9376
```

Applications query DNS and get direct Pod IPs instead of a single service IP.

---

3. Service Discovery

There are **two ways** clients inside the cluster can discover Services:

1. **Environment Variables:** Kubernetes automatically sets environment variables for each Service.
2. **DNS Resolution:** Services are registered in the cluster's DNS and can be accessed using my-service.default.svc.cluster.local.

Now In the Background

Some of the most important things you need to know:

Understanding Kubernetes Networking: kube-proxy and CNI Plugin

In Kubernetes, a **Service** allows applications to communicate with each other reliably, even when the underlying Pods are created and destroyed dynamically. But how does this communication actually happen? This is where two important networking components come into play:

1. **kube-proxy** – It ensures that traffic from a **Service** reaches the correct **Pod**.
2. **CNI Plugin (Container Network Interface)** – It enables internal networking between **Pods**, even when they are on different nodes.

Without these two networking components, a **Service** would not be able to route traffic to application Pods, and Pods wouldn't be able to communicate across the cluster. Let's break them down in simple terms.

---

1. What is kube-proxy?

kube-proxy is a network service that makes sure traffic from a **Service** reaches the right **Pod**. It acts like a traffic controller inside the Kubernetes cluster.

How kube-proxy Works?

- Each **Service** in Kubernetes gets a stable IP address.
- When a request comes to a Service, **kube-proxy** finds a healthy **Pod** behind that Service.
- It forwards the request to the selected Pod.
- If a Pod goes down, **kube-proxy** automatically redirects traffic to another healthy Pod.

Why is kube-proxy Important?

- Ensures that traffic from Services reaches the correct Pod.
- Handles **load balancing** between multiple Pods.
- Works on **every node** to manage network traffic.

Example: Imagine a website where users visit www.myapp.com. This request reaches a **Service** in Kubernetes. The **kube-proxy** decides which backend **Pod** should handle the request and forwards the traffic accordingly.

---

2. What is a CNI Plugin?

The **CNI Plugin (Container Network Interface)** is responsible for providing **network communication between Pods**.

How CNI Plugin Works?

- When a **Pod** is created, the CNI plugin assigns it a unique **IP address**.
- It ensures that all **Pods** can communicate with each other, even if they are running on different nodes.
- It makes sure that all **same-label** Pods (like replicas of a Deployment) can talk to each other easily.

Why is CNI Plugin Important?

- Ensures **all Pods can communicate** inside the cluster.
- Provides networking across **different nodes**.
- Helps maintain internal **localhost-style** communication between similar Pods.

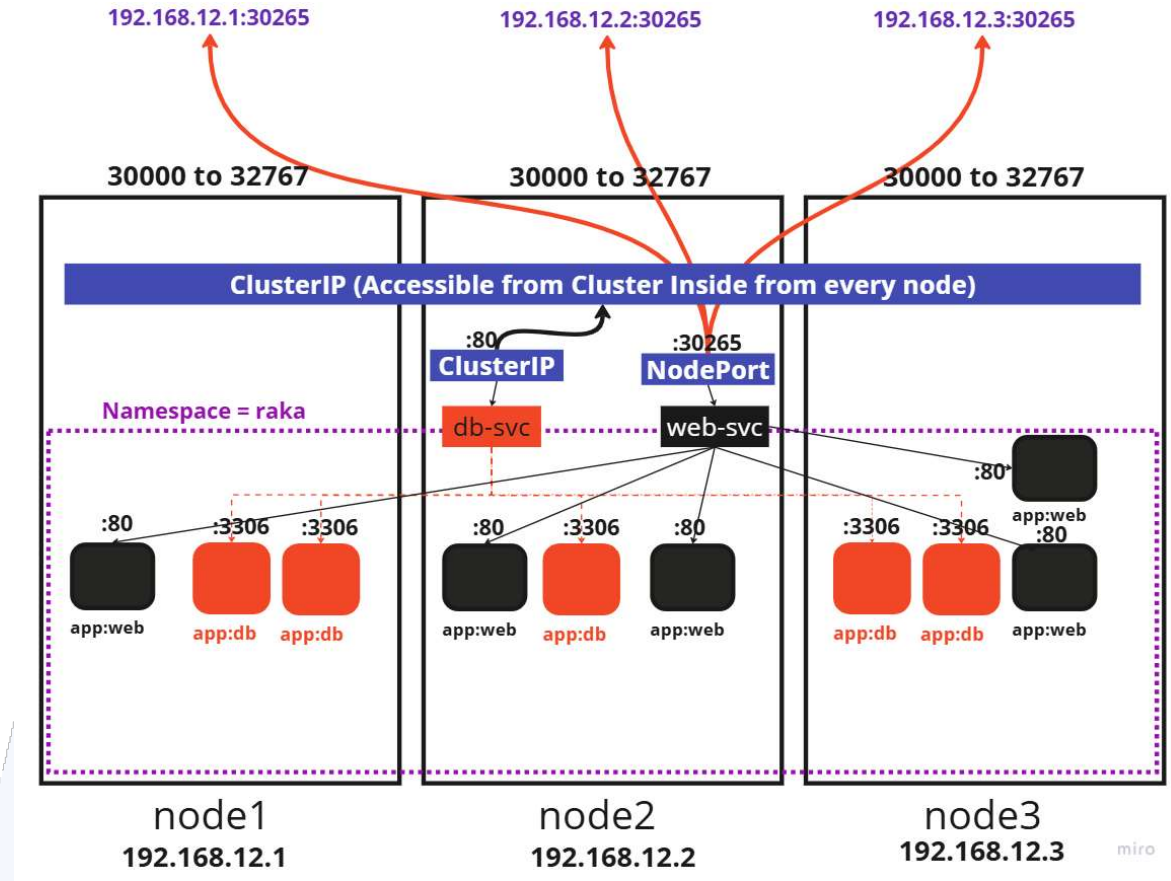
Example: If you have multiple replicas of a backend application (app: backend), the CNI plugin makes sure they can talk to each other smoothly.

Difference Between kube-proxy and CNI Plugin

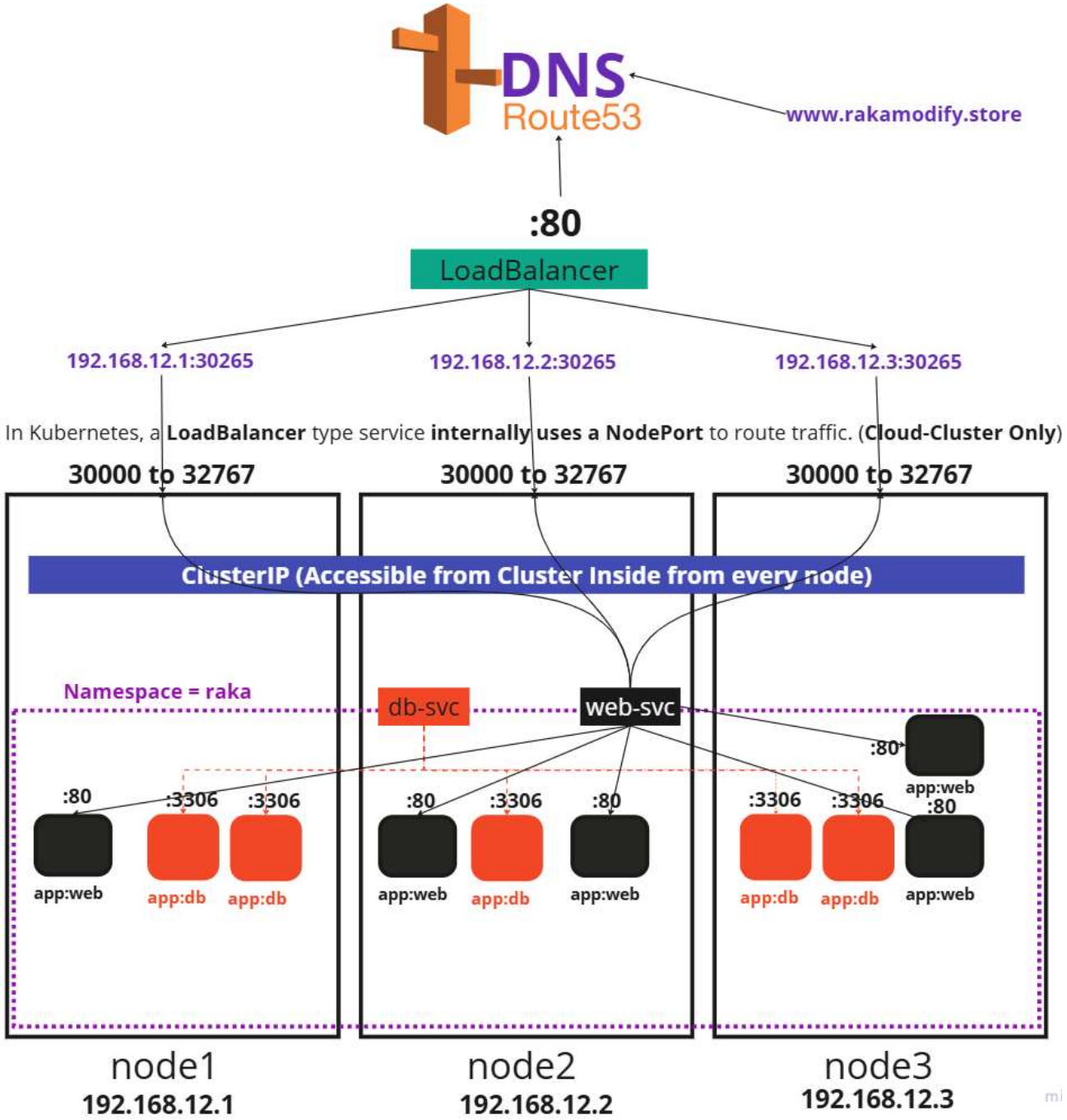
Feature	kube-proxy	CNI Plugin
Purpose	Connects Services to Pods	Connects Pods to each other
Works at	**Node level**	**Cluster level**
Manages	Service traffic routing	Pod-to-Pod networking
Technology	Uses iptables or IPVS	Uses plugins like Flannel, Calico, etc.

By understanding these two, you can better manage Kubernetes networking and troubleshoot connectivity issues easily.

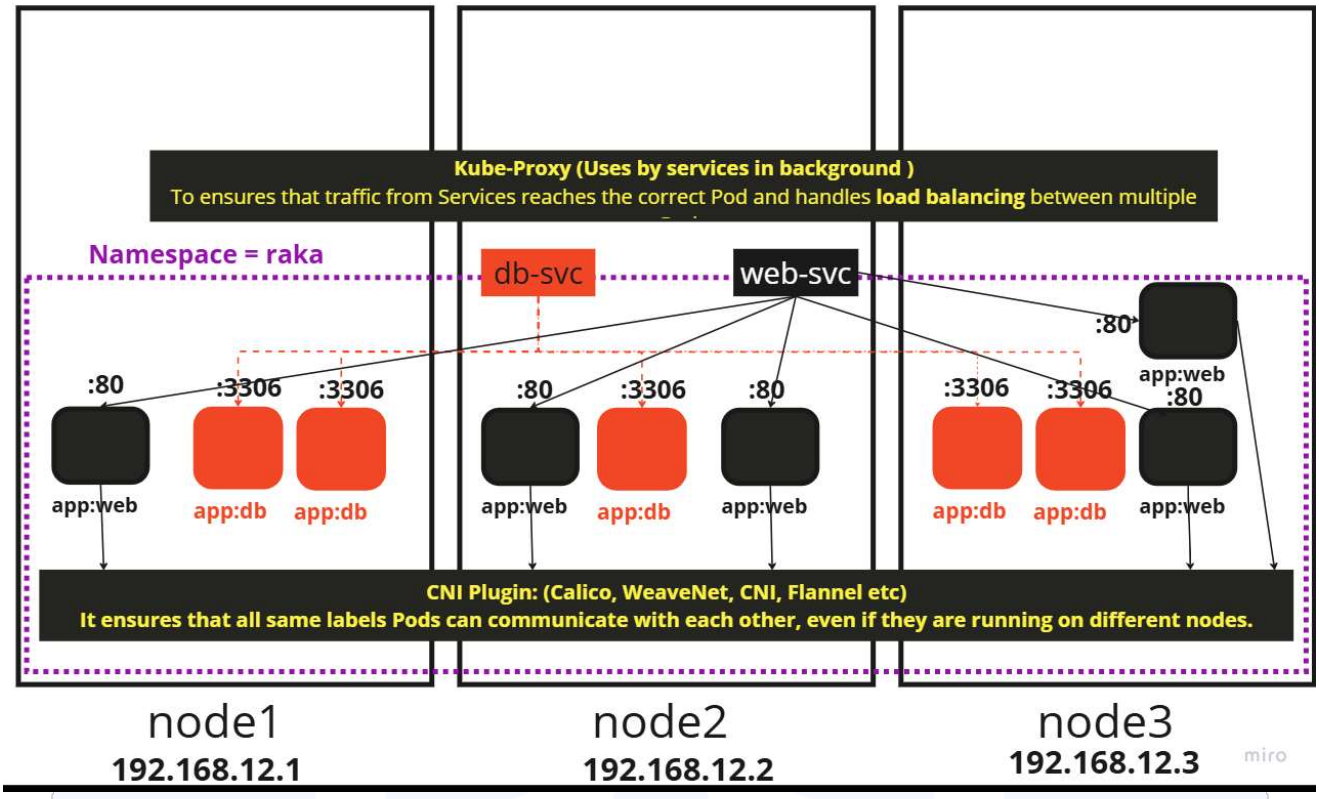
1. Diagram of ClusterIP and NodePort Service



2. Diagram of ClusterIP and NodePort Service



3. Diagram of Kube-Proxy and CNI plugin



My Final Thoughts

- Services ensure stable and reliable communication between Pods.
- Different Service types (ClusterIP, NodePort, LoadBalancer, ExternalName) cater to different needs.
- Load balancing, service discovery, and networking policies enhance Kubernetes' power.
- **kube-proxy** is like a **traffic manager** that connects Services to the right Pods.
- **CNI Plugin** ensures that all Pods can communicate internally, just like how computers in a network talk to each other.

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**

Creating an Authentication Configuration File in Kubernetes Using API and Custom Certification Keys

Introduction

In a Kubernetes cluster, granting access to multiple users with different levels of permissions is crucial for security and operational efficiency. By creating a separate user with customized authentication, we can implement Role-Based Access Control (RBAC) to ensure that only authorized personnel can perform specific actions.

Why Do We Need a New User?

- **Security**: Isolating user permissions prevents unauthorized access and accidental misconfigurations.
- **Least Privilege Principle**: Assigning minimal required permissions reduces security risks.
- **Multi-Tenant Access**: Different teams or users can have controlled access to the cluster.
- **Auditing & Monitoring**: Individual user accounts help track actions in logs and audits.
- **Service Accounts vs. Human Users**: Some tasks require dedicated service accounts, while human users need controlled authentication.

RBAC Requirements for User Access

When creating a new Kubernetes user, you should define their access level using RBAC policies:

1. **ClusterRole**: Defines what actions a user can perform cluster-wide (e.g., view, edit, admin access).
2. **Role**: Defines permissions within a specific namespace.
3. **RoleBinding**: Grants a user or group access to a namespace.
4. **ClusterRoleBinding**: Assigns a ClusterRole to a user, giving them permissions across the cluster.

This document provides a step-by-step guide to configuring a new Kubernetes user with authentication, signing certificates, configuring kubeconfig, and setting up RBAC-based permissions.

This document provides a step-by-step guide to configuring a new Kubernetes user with proper authentication and access control. The process includes generating keys, signing certificates, configuring kubeconfig, and setting up permissions for the new user.

Step 1: Create a Working Directory

- First, create a new directory where all required files will be stored.

  mkdir /rakesh

  cd /rakesh

Step 2: Copy Kubernetes CA Files

- Copy the Kubernetes Certificate Authority (CA) files to the newly created directory.

  ```
  cp /etc/kubernetes/pki/ca.* /rakesh
  ```

Step 3: Generate User Private Key and CSR

- Generate a private key for the new user:

  ```
  openssl genrsa -out rakesh.key 2048
  ```

- Create a Certificate Signing Request (CSR) using the generated private key:

  ```
  openssl req -new -key rakesh.key -out rakesh.csr
  ```

Step 4: Verify the Generated Files

- Check the created files using:

  ```
  ls
  ```

- Expected output:

  ```
  ca.crt    ca.key    rakesh.key    rakesh.csr
  ```

Step 5: Sign the CSR with Kubernetes CA

- Sign the CSR to generate the user certificate valid for 365 days:

  ```
  openssl x509 -req  -in rakesh.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out rakesh.crt –days 365
  ```

- Verify the files:

  ```
  ls
  ```

- Expected output:

  ```
  ca.crt    ca.key    rakesh.key    rakesh.csr  rakesh.crt
  ```

Step 6: Copy Kubernetes Configuration File

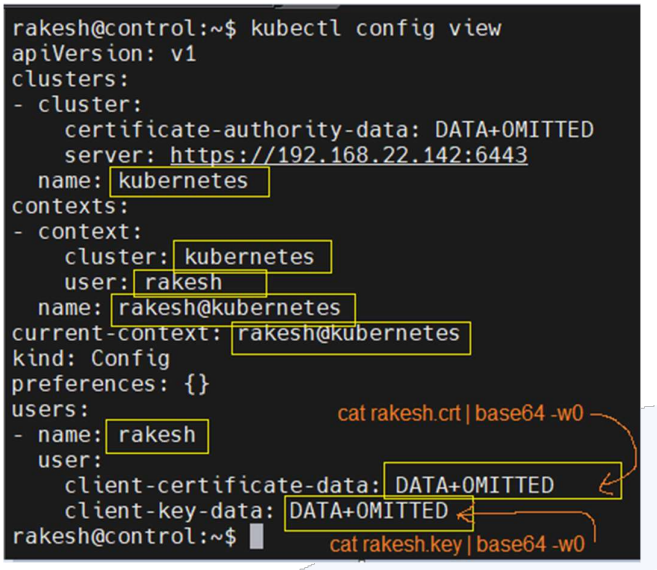
- Copy the Kubernetes admin configuration file for the new user:

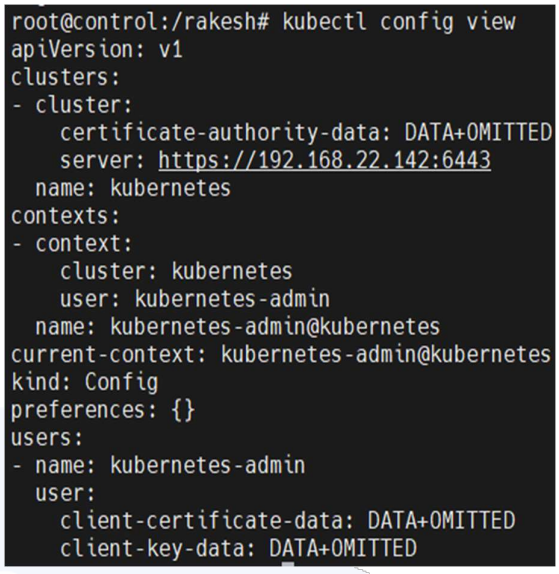
  ```
  cp /etc/kubernetes/admin.conf  /rakesh/config
  ```

Step 7: Encode Certificate and Key in Base64

- Convert the user certificate and private key to base64 encoding:

  ```
  cat rakesh.crt | base64 -w0
  cat rakesh.key | base64 -w0
  ```





Note: Create your own config file and change the config file values with the user's new value included users .key and .crt encoded hash values.

Step 8: Create a New Linux User

- Add a new system user:

  useradd rakesh

Step 9: Set Up kubeconfig for the User

- Create the .kube directory for the new user:

  mkdir -p /home/rakesh/.kube
- Copy the kubeconfig file:

  cp /rakesh/config /home/rakesh/.kube/config
- Set the correct ownership for the config file:

  chown rakesh:rakesh /home/rakesh/.kube/config

Step 10: Switch to the New User and Verify Configuration

- Switch to the newly created user:

  su – rakesh

  kubectl config view
- Run the following command to check if the Kubernetes cluster is accessible:

  kubectl get no

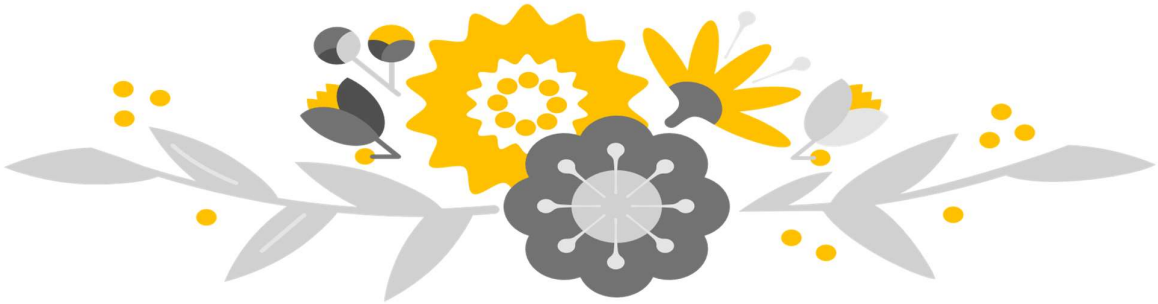
If user has specific role or cluster-role level authorization, it will list all Kubernetes nodes. Otherwise, it will show below message that showing that "User" are not able to list the resource. Which mean this is user authentication issue. This is Kubernetes role authorization issue.

```
rakesh@control:~$ kubectl get no
Error from server (Forbidden): nodes is forbidden: User "rakesh" ca
nnot list resource "nodes" in API group "" at the cluster scope
rakesh@control:~$
```

The new user rakesh is now successfully configured to interact with the Kubernetes cluster using kubectl. This setup ensures secure authentication through certificate-based authentication.

Note: **Step by step understand the Certificate Files process.**

1. **Why is rakesh.key (Private Key) Created First?**
   - The private key (rakesh.key) is the first file created because it is used to generate the certificate signing request (CSR).
   - It acts as a unique identifier for the user and should be kept secure.
2. **Why is rakesh.csr (Certificate Signing Request) Created Next?**
   - The CSR (rakesh.csr) is generated using the private key (rakesh.key).
   - It is a request that asks the Certificate Authority (CA) to issue a signed certificate.
   - This CSR contains information like the user's identity but is not yet trusted until signed.
3. **Why is rakesh.crt (Signed Certificate) Created at Last?**
   - The signed certificate (rakesh.crt) is created after the CSR is approved and signed by the CA.
   - This certificate verifies that the user is authenticated and allowed to interact with the Kubernetes cluster.
4. **What is the Role of ca.crt and ca.key?**
   1. **ca.crt (CA Public Certificate)**
      - This is the public certificate of the Kubernetes Certificate Authority.
      - It is used to verify that the user's certificate (rakesh.crt) is signed by a trusted CA.
   2. **ca.key (CA Private Key)**
      - This is the private key of the Kubernetes Certificate Authority.
      - It is used to sign CSRs and issue valid user certificates.
      - **This file is highly sensitive and should be securely stored.**



RBAC (Role-Based Access Control)

(A Powerful Student RBAC Guide)

Introduction

Role-Based Access Control (RBAC) is a security mechanism in Kubernetes that helps administrators control who can access cluster resources and what actions they can perform. It provides a way to assign permissions to users, groups, and service accounts to ensure secure operations in a Kubernetes environment.

RBAC allows fine-grained access management by defining roles and bindings that determine which resources can be accessed and modified. It is essential for maintaining security and preventing unauthorized access to critical components in a cluster.

1. Key Concepts in RBAC

RBAC in Kubernetes consists of the following main components:

A. Role

A Role defines a set of permissions within a specific namespace. It is used to grant Access to certain resources.

B. ClusterRole

A ClusterRole is similar to a Role but applies cluster-wide rather than within a single namespace.

C. RoleBinding

A RoleBinding associates a Role with a user, group, or service account, granting them the specified permissions within a namespace.

D. ClusterRoleBinding

A ClusterRoleBinding associates a ClusterRole with a user, group, or service account, granting them the specified permissions across the entire cluster.

---

2. Understanding Verbs and Resources in RBAC

A. Verbs in RBAC

Verbs define the actions that can be performed on Kubernetes resources. The common verbs are:

Verb	Description	Example Command
get	View a resource	kubectl get pods
list	View all resources of a type	kubectl get pods -n mynamespace
watch	Monitor resources for changes	watch kubectl get pods
create	Create a new resource	kubectl create deployment myapp --image=nginx
update	Modify an existing resource	kubectl edit deployment myapp
patch	Partially update a resource	kubectl patch pod mypod -p '{"metadata":{"labels":{"env":"prod"}}}'
delete	Remove a resource	kubectl delete pod mypod

Wildcard (*) can be used to grant all permissions.

B. Resources in RBAC

Resources are the Kubernetes objects on which actions can be performed. Some key resources are:

Resource	Description	Example Command
pods	Running application containers	kubectl get pods
deployments	Manages pods and scaling	kubectl get deployments
services	Network exposure of applications	kubectl get services
configmaps	Stores configuration data	kubectl get configmaps
secrets	Stores sensitive information	kubectl get secrets
namespaces	Logical separation in the cluster	kubectl get namespaces
nodes	Physical/virtual machines in the cluster	kubectl get nodes
persistentvolumes	Storage resources	kubectl get persistentvolumes
ingresses	Load balancer and routing rules	kubectl get ingresses

Wildcard (*) can be used to refer to all resources.

---

3. Creating RBAC Roles and RoleBindings

Creating a Role for a User in a Specific Namespace

```
kubectl create role gip-admin \
  --verb="*" \
  --resource="*" \
  --namespace=gip
```

This command creates a Role named gip-admin in the gip namespace, granting full permissions on all resources.

Creating a RoleBinding for a User

```
kubectl create rolebinding rakesh-gip-binding \
  --role=gip-admin \
  --user=rakesh \
  --namespace=gip
```

This binds the gip-admin Role to the user rakesh, allowing him to perform all actions in the gip namespace.

Creating a RoleBinding for a Group

```
kubectl create rolebinding devops-team-rolebinding \
  --role=gip-admin \
  --group=devops-team \
  --namespace=gip
```

This binds the gip-admin Role to the group devops-team, allowing all users in that group to perform actions in the gip namespace.

Creating a RoleBinding for a Service Account

```
kubectl create rolebinding monitoring-sa-rolebinding \
  --role=gip-admin \
  --serviceaccount=monitoring:monitoring-sa \
  --namespace=gip
```

This binds the gip-admin Role to the Service Account monitoring-sa in the monitoring namespace.

Verifying Role-Based Permissions

```
kubectl auth can-i create pods --namespace=gip --as=rakesh
kubectl auth can-i delete deployments --namespace=gip --as=rakesh
kubectl auth can-i list services --namespace=gip --as=rakesh
```

If the output is "yes", it means the permissions are correctly assigned.

---

4. Creating a ClusterRole and ClusterRoleBinding in Kubernetes

Creating a ClusterRole

```
kubectl create clusterrole cluster-admin-role \
  --verb="*" \
  --resource="*"
```

This command creates a **ClusterRole** named cluster-admin-role, granting full permissions on **all resources across the cluster**.

Creating a ClusterRoleBinding for a User

```
kubectl create clusterrolebinding rakesh-clusterrolebinding \
  --clusterrole=cluster-admin-role \
```

```
--user=rakesh
```

This binds the cluster-admin-role to the **user rakesh**, giving him full cluster-wide access.

Creating a ClusterRoleBinding for a Group

```
kubectl create clusterrolebinding devops-team-clusterrolebinding \
  --clusterrole=cluster-admin-role \
  --group=devops-team
```

This binds the cluster-admin-role to the **group devops-team**, allowing all users in that group to have full cluster-wide access.

Creating a ClusterRoleBinding for a Service Account

```
kubectl create clusterrolebinding monitoring-sa-clusterrolebinding \
  --clusterrole=cluster-admin-role \
  --serviceaccount=monitoring:monitoring-sa
```

This binds the cluster-admin-role to the **Service Account monitoring-sa in the monitoring namespace**, allowing it full cluster-wide access.

E. Verifying Role-Based Permissions

```
kubectl auth can-i create pods --as=rakesh
kubectl auth can-i delete deployments --as=system:serviceaccount:monitoring:monitoring-sa
kubectl auth can-i list services --as=system:serviceaccount:monitoring:monitoring-sa
kubectl auth can-i create nodes --as=devops-team
```

If the output is **"yes"**, it means the **permissions are correctly assigned**.

---

5. Some Example RBAC Scenarios (Verbs and Resources)

A. Read-Only Access to Pods and Services

```
kubectl create role viewer \
  --verb="get","list" \
  --resource="pods","services" \
  --namespace=myproject
```

This grants read-only access to pods and services in the myproject namespace.

B. Full Access to All Resources in a Namespace

```
kubectl create role admin \
  --verb="*" \
  --resource="*" \
  --namespace=myproject
```

This allows full control over all resources in myproject namespace.

C. ConfigMap and Secret Editing Permissions

```
kubectl create role config-editor \
  --verb="get","update" \
```

```
--resource="configmaps","secrets" \
--namespace=myproject
```

This lets a user view and update ConfigMaps and Secrets in myproject namespace.

---

5. Service Account

A. What is a Service Account?

A **Service Account** is a special type of account in Kubernetes. It is used by **applications running inside Pods** to communicate securely with the Kubernetes **API server**.

B. How is a Service Account Different from a Normal User Role/ClusterRole?

- **User accounts** are for humans (like developers, administrators).
- **Service accounts** are for **applications inside Pods**.
- Service accounts can have **Roles/ClusterRoles**, but they are mainly used by **Pods to access Kubernetes resources**.

C. Can We Assign a Role/ClusterRole to a Service Account?

Yes! You can assign a **Role (namespace-level)** or a **ClusterRole (cluster-wide)** to a service account using **RoleBinding or ClusterRoleBinding**.

D. Is a Service Account Namespace-Based or Cluster-Based?

- By default, **a service account is namespace-based**.
- A service account from one namespace **cannot** be used in another namespace **unless granted access using ClusterRoleBinding**.

E. Why Do We Need a Service Account?

1. To allow Pods to securely access the API server.
2. To control permissions for specific workloads (applications).
3. To automate tasks without using user credentials.

---

6. Service Account Practical Example: (Monitoring DaemonSet Pod with a Service Account)

Imagine you have a **Monitoring Pod** that needs to **view Pods, Nodes, and Services** in Kubernetes. So for this we have to create a service account for the monitoring Pod

- **Creating a Role and RoleBinding for a Service Account**

Create a Service Account:

```
kubectl create serviceaccount monitoring-sa --namespace=monitoring
```

Create a Role for Monitoring:

```
kubectl create role monitoring-role \
  --verb=get,list,watch \
  --resource=pods,nodes,services \
  --namespace=monitoring
```

Bind the Role to the Service Account:

```
kubectl create rolebinding monitoring-rb \
  --role=monitoring-role \
  --serviceaccount=monitoring:monitoring-sa \
  --namespace=monitoring
```

Use the Service Account in a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: monitoring-pod
  namespace: monitoring
spec:
  serviceAccountName: monitoring-sa
  containers:
   - name: monitoring-containe
     image: monitoring-image
```

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**

Ingress Controller & Ingress Resource (A Strong Student Guide to K8s Networking)

First Understand the Problem Statement (Understanding the Need for Ingress)

Imagine you have a Kubernetes cluster running multiple services, such as a frontend that handles user interactions, a backend that processes requests, and a database that stores information. To make these services accessible to users outside the cluster, you need to expose them.

One way to expose services is by using NodePort, which assigns a unique port on every node. However, this approach has a major drawback—each service requires an open port, increasing security risks by making cluster nodes directly accessible from the internet. Another option is using

a LoadBalancer, which provides external access by creating a dedicated cloud-based load balancer for each service. While this method is effective, it becomes expensive and difficult to manage as the number of services grows.

Managing multiple NodePorts or LoadBalancers for different services leads to security vulnerabilities, increased operational complexity, and high costs. Instead of exposing every service separately, a more efficient approach is needed.

And the solution? **Ingress!**

Ingress acts as a single entry point that routes external requests to the appropriate services based on defined rules. Instead of assigning a separate LoadBalancer or NodePort for each service, Ingress allows controlled and secure access while simplifying traffic management, reducing costs, and improving security.

Prerequisites Before Using Ingress in Your Cluster

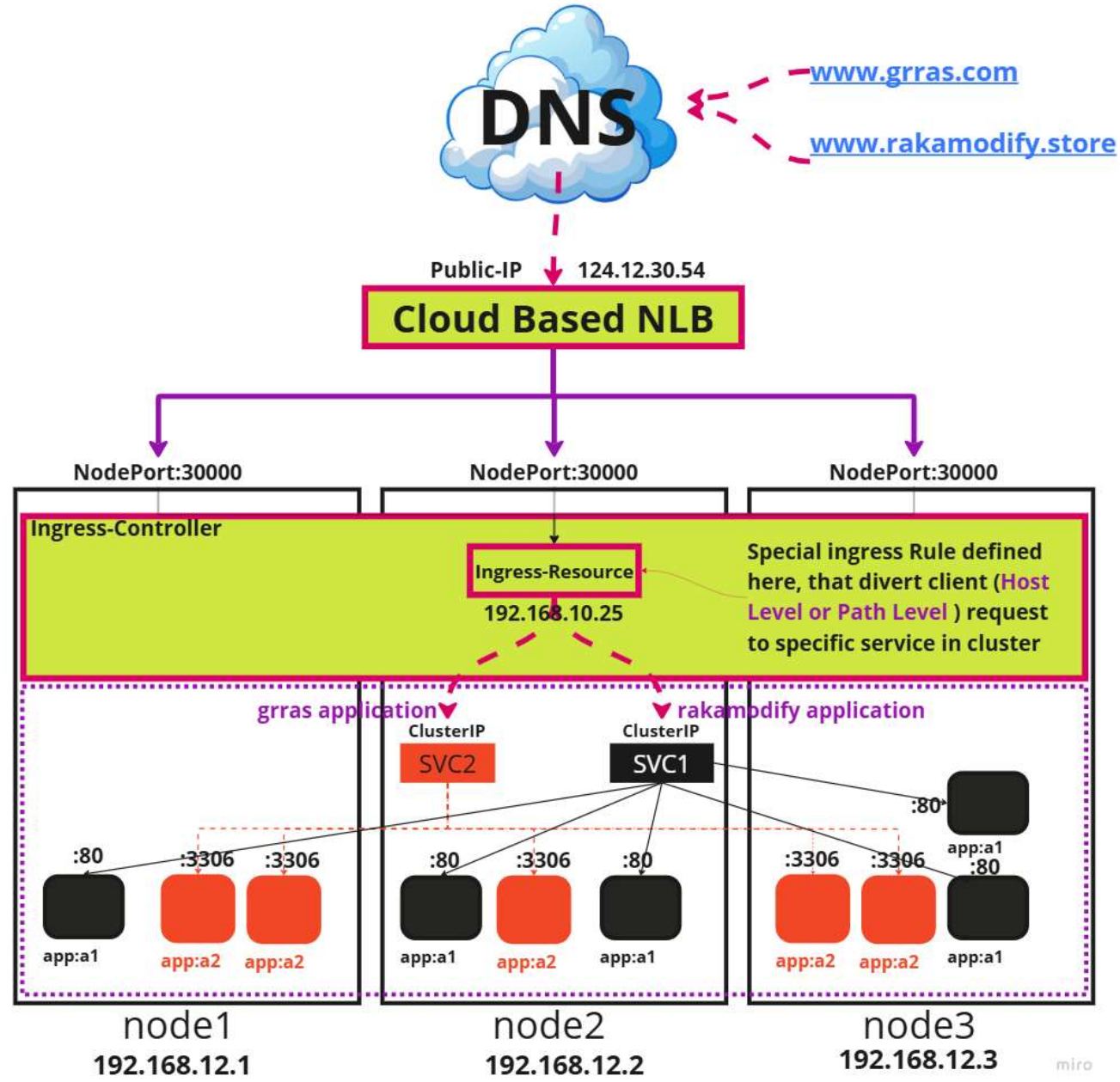
Before implementing Ingress to manage external traffic in your Kubernetes cluster, there are two essential components you must have in place. Without these, Ingress cannot function properly.

- **Ingress Controller** – This is the actual component that processes incoming traffic and enforces the routing rules defined in Ingress Resources. Kubernetes does not come with an Ingress Controller by default, so you need to install and configure one separately.
- **Ingress Resource** – This is a Kubernetes object that defines the rules for routing traffic, such as which requests should be sent to which service based on domain names or URL paths. However, an Ingress Resource alone does not handle traffic; it requires an Ingress Controller to function.

Without an Ingress Controller, Ingress Resources are just configurations that Kubernetes cannot act upon. In the next section, we will dive deeper into both **Ingress Controller** and **Ingress Resource** to understand how they work together to manage external traffic efficiently.

I have created an **image diagram below** to visually explain how **Ingress and Ingress Controller function with two deployment services inside your Kubernetes cluster.**

Ex: Image Diagram:



What is an Ingress Controller?

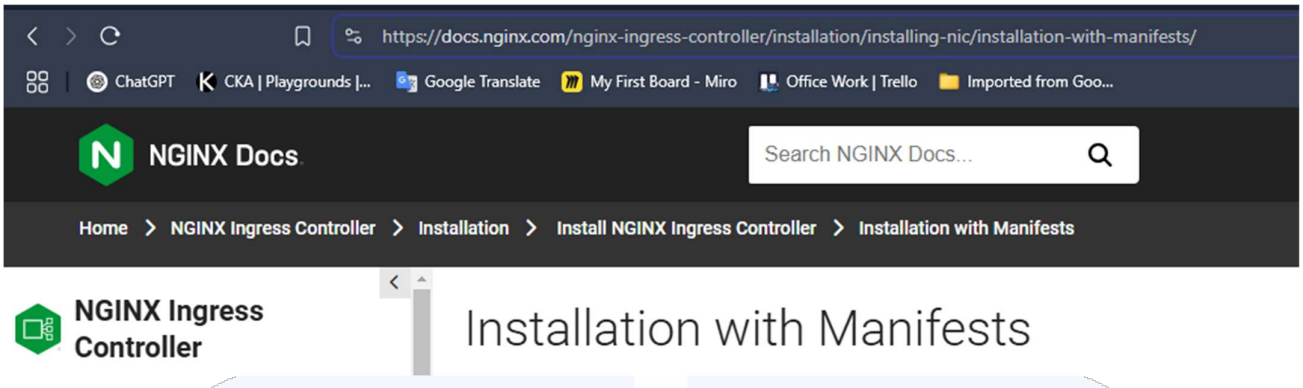
In Kubernetes Ingress Resource alone cannot process requests. It requires an **Ingress Controller** to implement and enforce the rules defined in Ingress.

Ingress Controller is a special pod that reads Ingress Resources and routes traffic accordingly.

- **Some Popular Ingress Controllers in Market:**
  - **NGINX Ingress Controller** (widely used)
  - **Traefik Ingress Controller**
  - **HAProxy Ingress Controller**
  - **Istio Gateway**

How to install Ingress Resource in your Kubernetes cluster?

Ingress Controller helps manage external traffic in a Kubernetes cluster. We are following the official nginx documentation for this Link: **https://docs.nginx.com/ .**

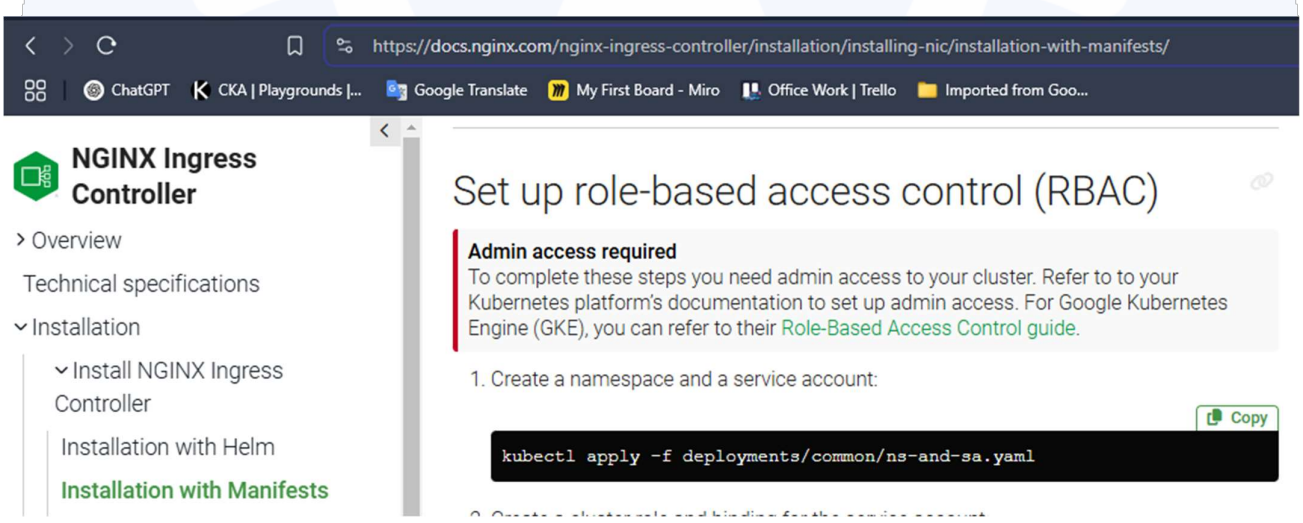


There are multiple ways to install the **NGINX Ingress Controller**:

1. **Installation with Helm**
2. **Installation with Manifests** (Recommended in this guide)
3. **Installation with NGINX Ingress Operator**
4. **Upgrade to NGINX Ingress Controller 4.0.0**

For detailed steps, follow the official documentation:

https://docs.nginx.com/nginx-ingress-controller/installation/installing-nic/installation-with-manifests/



What is an Ingress Resource?

Ingress Resource is a Kubernetes object that defines rules for routing HTTP and HTTPS traffic to the correct service inside the cluster.

It allows:

- URL-based routing
- Load balancing
- SSL/TLS termination
- Redirects and rewrites

Real-World project: (Traffic Flow in Kubernetes with Ingress)

Scenario Overview

In our Kubernetes cluster, two projects are running with the following domains:

- **www.grras.com** → Should be routed to **SVC2** (backend-service)
- **www.rakammodify.store** → Should be routed to **SVC1** (frontend-service)

How Does the Request Flow?

1. A client requests either www.grras.com or www.rakammodify.store.
2. The request first goes to the Public DNS (Domain Name System), which resolves the domain name to the public IP of the Network Load Balancer (NLB).
3. The NLB forwards the request to the attached Kubernetes cluster nodes using a NodePort service.
4. The Ingress Controller, running on the cluster, receives the request.
5. Based on the Ingress rules, the request is routed:
    - If the request is for www.grras.com, it is forwarded to SVC2.
    - If the request is for www.rakammodify.store, it is forwarded to SVC1.
6. The respective Pods behind the services process the request and respond back using the same route in reverse.

What is written in this traffic rule (Inside Ingress Resource)

This is an Ingress resource for routing traffic in the Kubernetes cluster.
It is created in the "amazon" namespace.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: project-ingress     # Name of the Ingress resource
  namespace: amazon      # Namespace where this Ingress is applied
spec:
  rules:
  - host: www.grras.com     # When a request comes for this domain
    http:                   # Rule 1: Handles traffic for www.grras.com
      paths:
      - path: /             # Routes all traffic ("/" means root path)
        pathType: Prefix    # Ensures the path-based routing
        backend:
          service:
```

```
        name: svc2          # Traffic is forwarded to the service "svc2"
        port:
          number: 80        # The service port it should use
  - host: www.rakammodify.store      # When a request comes for this domain
    http:                   # Rule 2: Handles traffic for www.rakammodify.store
      paths:
      - path: /             # Routes all traffic for this domain
        pathType: Prefix    # Path-based routing
        backend:
          service:
            name: svc1      # Traffic is forwarded to the service "svc1"
            port:
              number: 80    # The service port it should use
```

 # kubectl create -f ingress-resource.yml

Why Should we Use Ingress?

When deploying applications in Kubernetes, exposing services externally is a crucial challenge. Let's explore why Ingress is the best solution.

What was the problem without Ingress

1. **Using NodePort:**
   a) When you expose an application using NodePort, each service requires a unique port.
   b) If you have multiple services, you must open multiple NodePorts, which exposes your cluster nodes directly to the internet.
   c) More open NodePorts = More security risks (like open doors for hackers).
2. **Using LoadBalancer for Each Service:**
   a) Every LoadBalancer service creates a new cloud-based Load Balancer, which is expensive.
   b) More services = More LoadBalancers = High cost & complex setup.

What is the solution: Ingress!

Ingress acts as a smart gateway that solves these issues efficiently.

1. **Smart Traffic Routing**
   - Ingress allows you to route traffic based on domain names or URL paths instead of assigning a NodePort to every service.
   - Example:
     - www.grras.com → Forwarded to svc2
     - www.rakammodify.store → Forwarded to svc1
2. **Cost-Effective**

- Instead of requiring a separate LoadBalancer for every service, Ingress works with a single LoadBalancer and intelligently directs traffic.
- Fewer LoadBalancers = Less cost & easy maintenance.

3. **Improved Security**
   - Since Ingress handles traffic internally, you don't need to expose multiple NodePorts.
   - Fewer open ports = Less attack surface = Better security.
   - Also supports TLS termination (HTTPS), making communication secure.
4. **Simplified Configuration**
   - **Ingress provides powerful features like:**
     - URL rewriting (e.g., /old-path → /new-path)
     - Redirections (e.g., http → https)
     - Traffic splitting (e.g., A/B testing).

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**

Setting Up Kubernetes v1.30.2 Cluster Using Kubeadm on Ubuntu 22.04 LTS with 12 Easy Steps

---

What is Prerequisites

- Ubuntu 22.04 LTS installed on all nodes
- Internet access
- User with sudo privileges

Step 1: Disable Swap on All Nodes

```
swapoff -a
sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
```

Step 2: Enable IPv4 Packet Forwarding

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.ipv4.ip_forward = 1
EOF
sudo sysctl –system
```

Step 3: Verify IPv4 Packet Forwarding

```
sysctl net.ipv4.ip_forward
```

Step 4: Install Containerd

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc


echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME")
stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null


sudo apt-get update && sudo apt-get install -y containerd.io
systemctl enable --now containerd
```

Step 5: Install CNI Plugin

```
wget https://github.com/containernetworking/plugins/releases/download/v1.4.0/cni-plugins-linux-
amd64-v1.4.0.tgz
mkdir -p /opt/cni/bin
tar Cxzvf /opt/cni/bin cni-plugins-linux-amd64-v1.4.0.tgz
```

Step 6: Configure Networking

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
```

```
net.ipv4.ip_forward = 1
EOF

sudo sysctl –system
```

Step 7: Modify containerd Configuration for Systemd Support

```
sudo containerd config default | sudo tee /etc/containerd/config.toml
sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' /etc/containerd/config.toml
sudo systemctl restart containerd
```

Step 8: Install Kubernetes Components

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.30/deb/Release.key | sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.30/deb/ /" | sudo tee /etc/apt/sources.list.d/kubernetes.list
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo systemctl enable --now kubelet
```

Step 9: Initialize the Cluster on the Master Node

```
sudo kubeadm init
```

Step 10: Configure kubectl for the Master Node

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Step 11: Deploy a Network Plugin (Calico)

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

Step 12: Join Worker Nodes

Get the join command from the master node and run it on each worker node:

```
kubeadm token create --print-join-command
```

On worker nodes, run the output command received from the master.

Step 13: Verify Cluster Status

```
kubectl get nodes
```

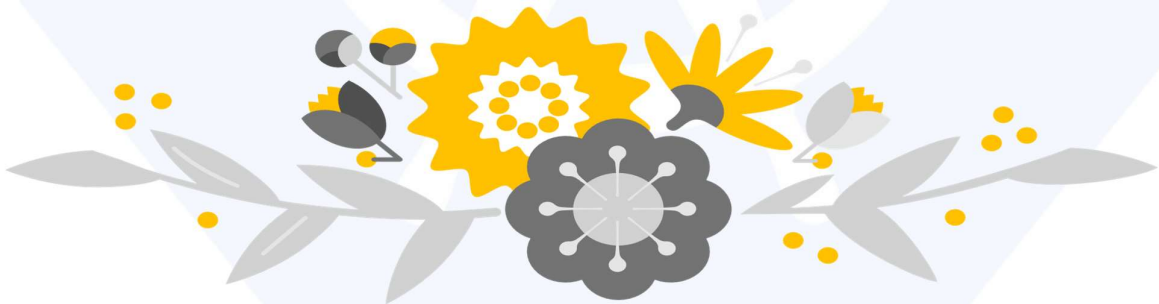
```
root@control:~# kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
control   Ready    control-plane   22d   v1.30.9
node1     Ready    <none>          22d   v1.30.9
node2     Ready    <none>          22d   v1.30.9
root@control:~#
```

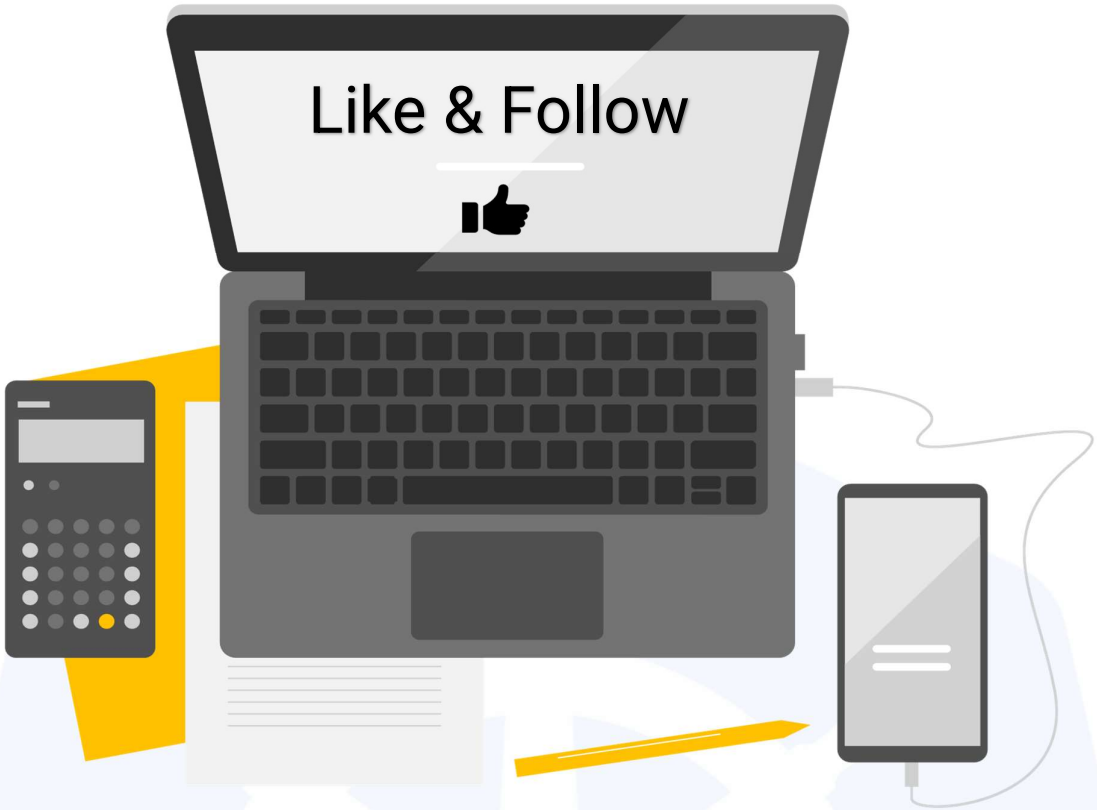
Your Kubernetes cluster is now set up and ready to use!

Kubectl get all -n kube-system

```
root@control:~# kubectl get all -n kube-system
NAME                                    READY   STATUS             RESTARTS         AGE
pod/coredns-55cb58b774-bpkmn            0/1     Running            7 (6h5m ago)     10d
pod/coredns-55cb58b774-qzdbz            0/1     Running            7 (6h5m ago)     10d
pod/etcd-control                        1/1     Running            16 (6h6m ago)    22d
pod/kube-apiserver-control              1/1     Running            16 (6h6m ago)    22d
pod/kube-controller-manager-control     1/1     Running            17 (6h6m ago)    22d
pod/kube-proxy-k4qgm                    1/1     Running            16 (6h6m ago)    22d
pod/kube-proxy-lj68f                    1/1     Running            15 (6h5m ago)    22d
pod/kube-proxy-t2qb8                    1/1     Running            15 (6h5m ago)    22d
pod/kube-scheduler-control              1/1     Running            17 (6h6m ago)    22d
pod/weave-net-gdjhr                     1/2     CrashLoopBackOff   1149 (22s ago)   22d
pod/weave-net-hd8cl                     2/2     Running            43 (6h6m ago)    22d
pod/weave-net-vl2xs                     2/2     Running            37 (6h3m ago)    22d

NAME               TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns   ClusterIP   10.96.0.10    <none>        53/UDP,53/TCP,9153/TCP   22d

NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
daemonset.apps/kube-proxy   3         3         3       3            3           kubernetes.io/os=linux   22d
daemonset.apps/weave-net    3         3         2       3            2           <none>                   22d

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns   0/2     2            0           22d

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-55cb58b774   2         2         0       22d
root@control:~#
```

Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**





Follow this LinkedIn channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**