**Kubernetes:**

**Arctitucture:**

*Control plane components:*

Kube-Api-server:

The **Kube-API-Server** is the central management component of Kubernetes. It acts as a RESTful interface between users, tools, and the cluster, handling all API requests to manage Kubernetes objects like pods, services, and nodes. It validates, processes, and persists API objects in etcd, ensuring the cluster's desired state is maintained.

**Kubernetes Scheduler**:

The **Kubernetes Scheduler** is a control plane component responsible for assigning unscheduled pods to suitable nodes in the cluster. It evaluates various factors like resource availability (CPU, memory), node labels, taints, tolerations, and affinity rules to determine the best node for each pod, ensuring optimal resource utilization and workload balancing.

**ETCD:**

**etcd** is a distributed key-value store used by Kubernetes to store all cluster data, including configuration, state, and metadata. It serves as the source of truth for the cluster, enabling consistency and coordination across nodes. Kubernetes components interact with etcd to retrieve or update the desired and current state of the cluster.

**Kubernetes Controller Manager**:

The **Kubernetes Controller Manager** is a control plane component that runs various controllers to manage the state of the cluster. Controllers monitor the desired state defined in the API server and work to maintain it by making necessary adjustments. Examples include the **Node Controller** (for node health), **Replication Controller** (for maintaining pod replicas), and **Endpoint Controller** (for updating service endpoints).

*Worker node:*

**Kubelet:**

The **Kubelet** is an agent that runs on each Kubernetes node. It ensures that containers in pods are running as expected by communicating with the control plane and managing container lifecycle on the node. It retrieves pod specifications from the API server, launches containers using the container runtime, monitors their health, and reports node and pod status back to the control plane.

**Kube-proxy:**

**Kube-proxy** is a network component that runs on each Kubernetes node. It manages network rules to enable communication between pods and services within the cluster and with external clients. It uses IP tables or user space proxying to forward traffic to the appropriate pod based on service definitions, ensuring load balancing and connectivity.

Kind Installation:

Install docker

# apt-get install docker.io

# sudo usermod -aG docker ubuntu

# sudo newgrp docker

Install kubectl

# chmod +x kubectl

# mv kubectl /usr/local/bin/kubectl

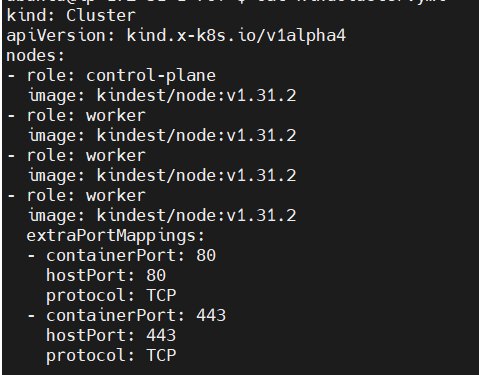
Install kind:

# For AMD64 / x86\_64

[ $(uname -m) = x86\_64 ] && curl -Lo ./kind <https://kind.sigs.k8s.io/dl/v0.26.0/kind-linux-amd64>

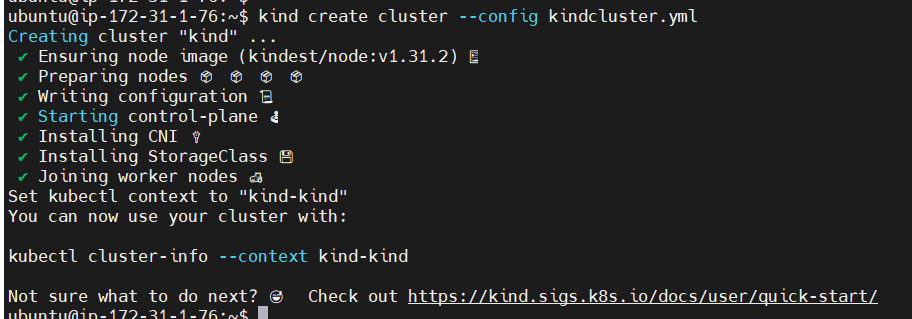
# mv kind /usr/local/bin/kind

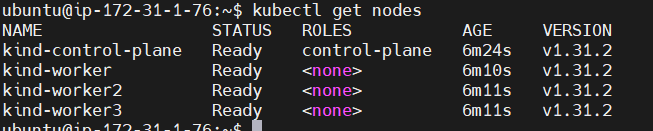
Kindcluster.yml



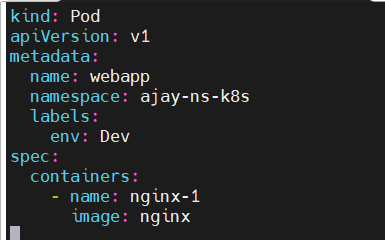
# kind create cluster --config kindcluster.yml

# kind get clusters





**Pod:** A **Pod** is the smallest deployable unit in Kubernetes. It represents one or more tightly coupled containers that share the same network namespace, storage volumes, and configuration. Pods are used to host application workloads and ensure containers can communicate easily with each other. Each pod runs on a single node and provides a consistent environment for its containers.



# kubectl run busybox -n=ajay-ns-k8s --image=busybox:latest

# kubectl apply -f pod.yml

# kubectl get pods -n ajay-ns-k8s –watch

# kubectl get pods

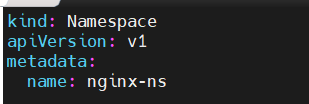
# kubectl run nginx --image=nginx

# kubectl describe pod newpods-dmj8x

# kubectl get pods -o wide

**Namespace:**

A **Namespace** in Kubernetes is a logical partition used to organize and isolate resources within a cluster. It allows multiple teams or projects to share the same cluster without interference by providing scope for resource names and access controls. Commonly used namespaces include default, kube-system, and kube-public, while custom namespaces can be created as needed.



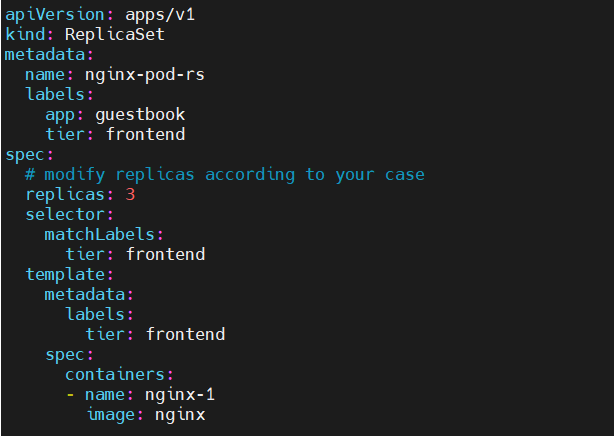
# kubectl apply -f name\_file.yaml

Or # kubectl create namespace ajay-ns-k8s

# kubectl get ns

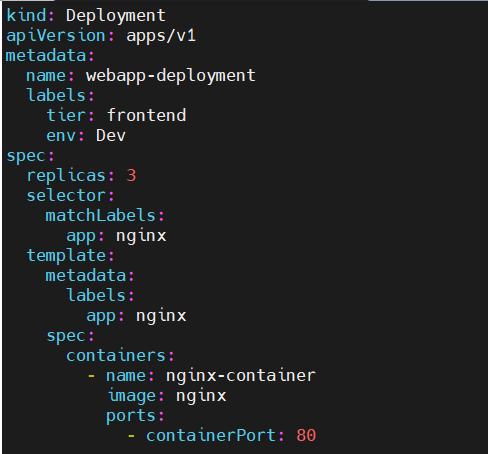
**ReplicaSet:**

A **ReplicaSet** in Kubernetes ensures that a specified number of identical pod replicas are running at all times. It monitors pod health and automatically replaces failed or terminated pods to maintain the desired state. ReplicaSets are commonly used with Deployments to manage scaling and rolling updates of applications.

****

**Deployments**:

A **Deployment** in Kubernetes is a higher-level abstraction that manages ReplicaSets and pods. It provides declarative updates for applications, allowing you to define the desired state of your application, such as the number of replicas, container images, and update strategies. Deployments enable easy scaling, rolling updates, rollbacks, and self-healing for applications.



# kubectl apply -f Deployment.yml

# kubectl get pods –watch

# kubectl get deployments

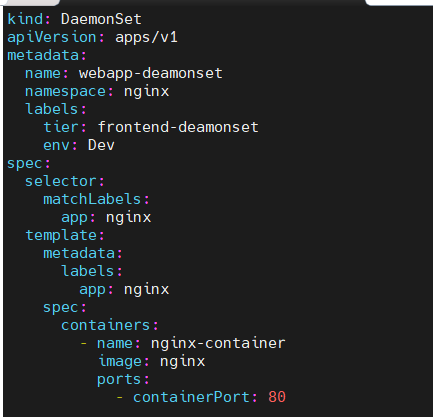
Create eployment by command:

# kubectl create deployment backend-deployment --image=nginx --replicas=2

# kubectl create deployment backend-deployment --image=nginx --replicas=2 --dry-run=client -o yaml > dep.yaml

**DaemonSet**:

It is typically used for background tasks such as monitoring, logging, or network management, where a copy of the pod needs to be present on each node. When a new node is added to the cluster, the DaemonSet automatically schedules the pod on the new node.



# kubectl get deamonset -n kube-system

# kubectl get ds -n kube-system

# kubectl get pods -n kube-system

# kubectl delete ds fluentd-elasticsearch-z9w7n -n kube-system

# kubectl get ds -n kube-system

# kubectl delete ds fluentd-elasticsearch -n kube-system

**Jobs**:

A **Job** in Kubernetes is a resource that manages the execution of a specific task or batch processing. It ensures that a set number of pods successfully complete their work. Once the pods successfully finish their task, the Job is considered complete. Jobs are commonly used for tasks like backups, data processing, or other one-time or scheduled operations. They can also be configured to retry failed pods until the task is successfully completed.

# kubectl apply -f Jobs.yaml

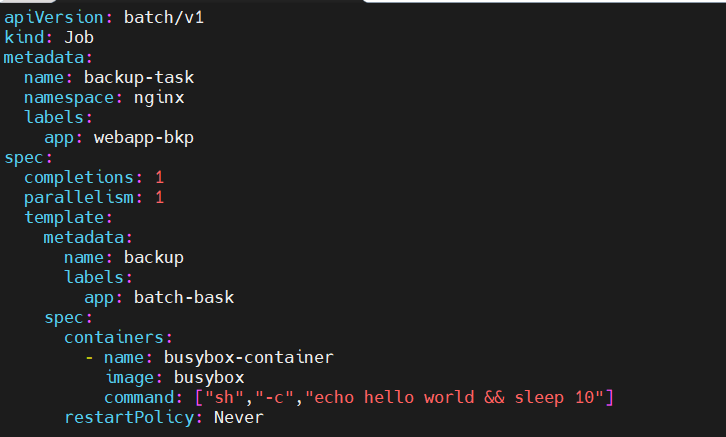
# kubectl get pods

# kubectl logs demo-job-wbbzl

# kubectl get pods

# kubectl get Jobs

# kubectl delete Jobs demo-job



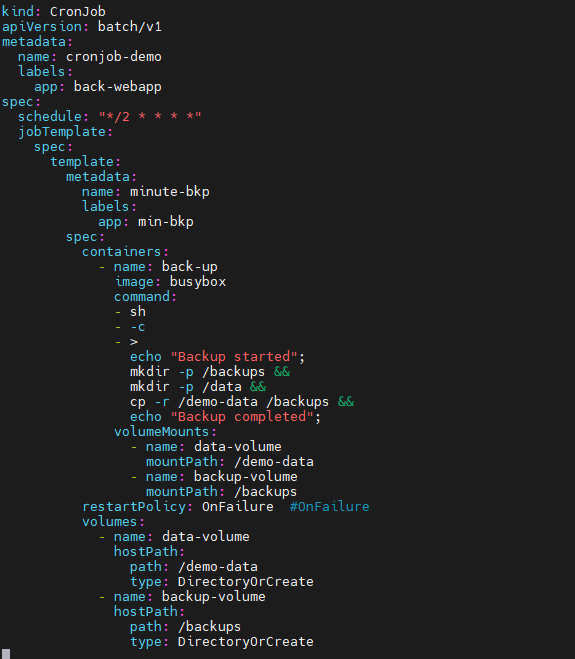
**Cronjobs**:

A **CronJob** in Kubernetes is a resource that runs jobs on a scheduled basis, similar to cron jobs in Unix-based systems. It allows you to specify time-based schedules using cron syntax (e.g., every hour, daily, weekly). CronJobs are ideal for recurring tasks like backups, report generation, or regular maintenance jobs. Each CronJob creates a Job at the specified time and ensures it runs to completion.

# kubectl apply -f cronjob.yml

# kubectl get cronjobs

# kubectl delete -f cronjob.yml



**Volumes:**

**Persistent Volumes (PV) and Persistent Volume Claims (PVC):**

**Persistent Volume (PV):**

A **Persistent Volume (PV)** in Kubernetes is a resource that provides storage in the cluster for persistent data. It is independent of a pod's lifecycle and enables data to persist even if the pod using the storage is deleted.

Persistent Volumes (PV) and Persistent Volume Claims (PVC) in Kubernetes are mechanisms to manage persistent storage for your applications.

A storage resource provisioned by an administrator or dynamically provisioned using a StorageClass.

It is a cluster-wide resource and is independent of pods.

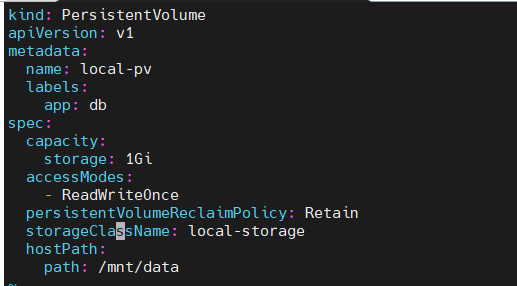
Defines storage details like capacity, access modes, and storage backend (e.g., NFS, AWS EBS, GCE PD).

**Persistent Volume Claim (PVC):**

A request for storage by a user or application.

It is bound to a PV that meets its requirements (e.g., size, access modes).

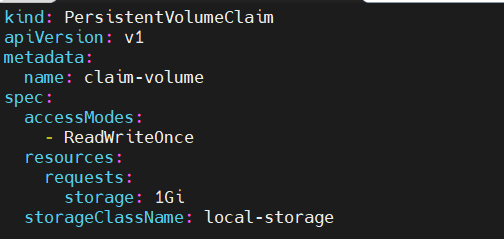
Acts as an interface for pods to use the storage.



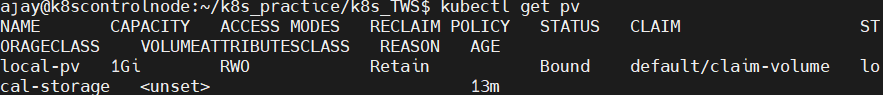


We can see the STATUS is Available

# kubectl apply -f claimpersistentvolume.yml

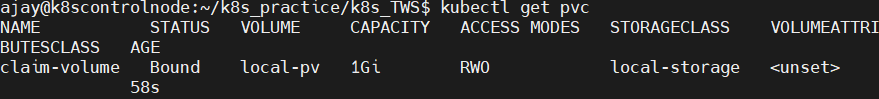


# kubectl get pv

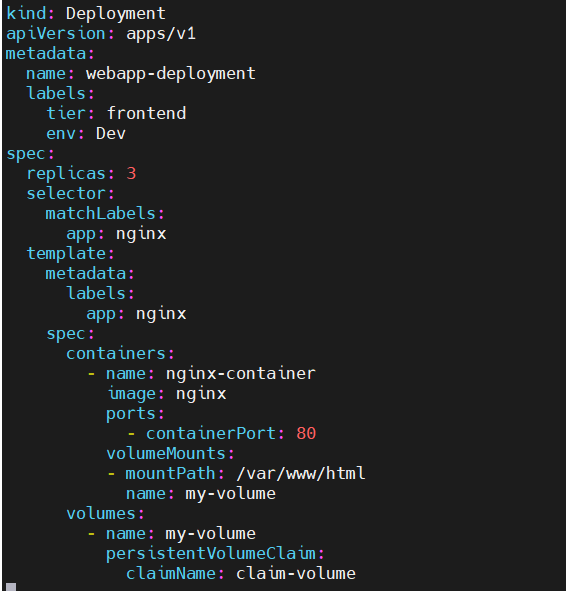


We can see the STATUS is Bound

# kubectl get pvc



Attach volume to Deployment:



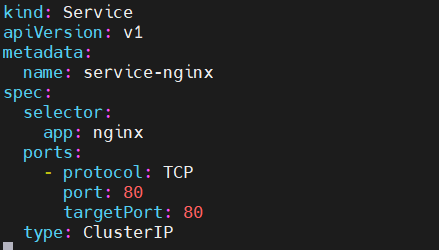
**Service**:

A **Service** in Kubernetes is an abstraction that defines a logical set of pods and a policy to access them, typically through a stable IP address or DNS name. It provides network connectivity to the pods, ensuring that even as pods are created or terminated, the service remains accessible. There are different types of services:

1. **ClusterIP**: Exposes the service on a cluster-internal IP. This is the default type, and the service is only accessible within the cluster.
2. **NodePort**: Exposes the service on a static port on each node's IP, allowing external access to the service.
3. **LoadBalancer**: Creates an external load balancer (typically cloud-provided) to expose the service to the outside world.
4. **ExternalName**: Maps the service to an external DNS name, enabling external services to be accessed via a Kubernetes service.

Services enable communication between different components of a Kubernetes application.

Services: NodePort, ClusterIp, and Load-Balancer.



Here, we are expose deployment, using label selector: app: nginx

All labels with app: nginx can be access with port 80.

# kubectl port-forward service/service-nginx -n default 80:80 --address=0.0.0.0

# sudo -E kubectl port-forward service/service-nginx -n default 80:80 --address=0.0.0.0

Mini Project:

Step 1: Create Docker image

# docker build -t notes-app-k8s .

# docker image tag notes-app-k8s:latest shajay3007/ask-notes-app-k8s:latest

# docker run -d -p 8000:8000 --name notes-app shajay3007/ask-notes-app-k8s:latest

Access and check the container functionality.

Configure the Docker Hub account:

# docker login -u shajay3007

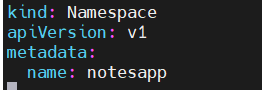
# password: <PAT Token>

# docker push shajay3007/ask-notes-app-k8s:latest

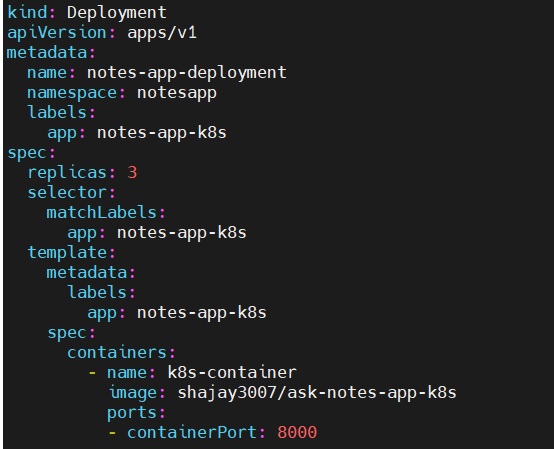
Setp 2:

Create a Yaml files for k8s:

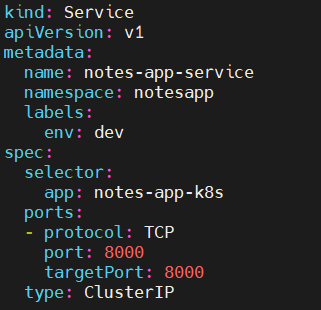
Namespace.yml



Deployment.yml



Service.yml

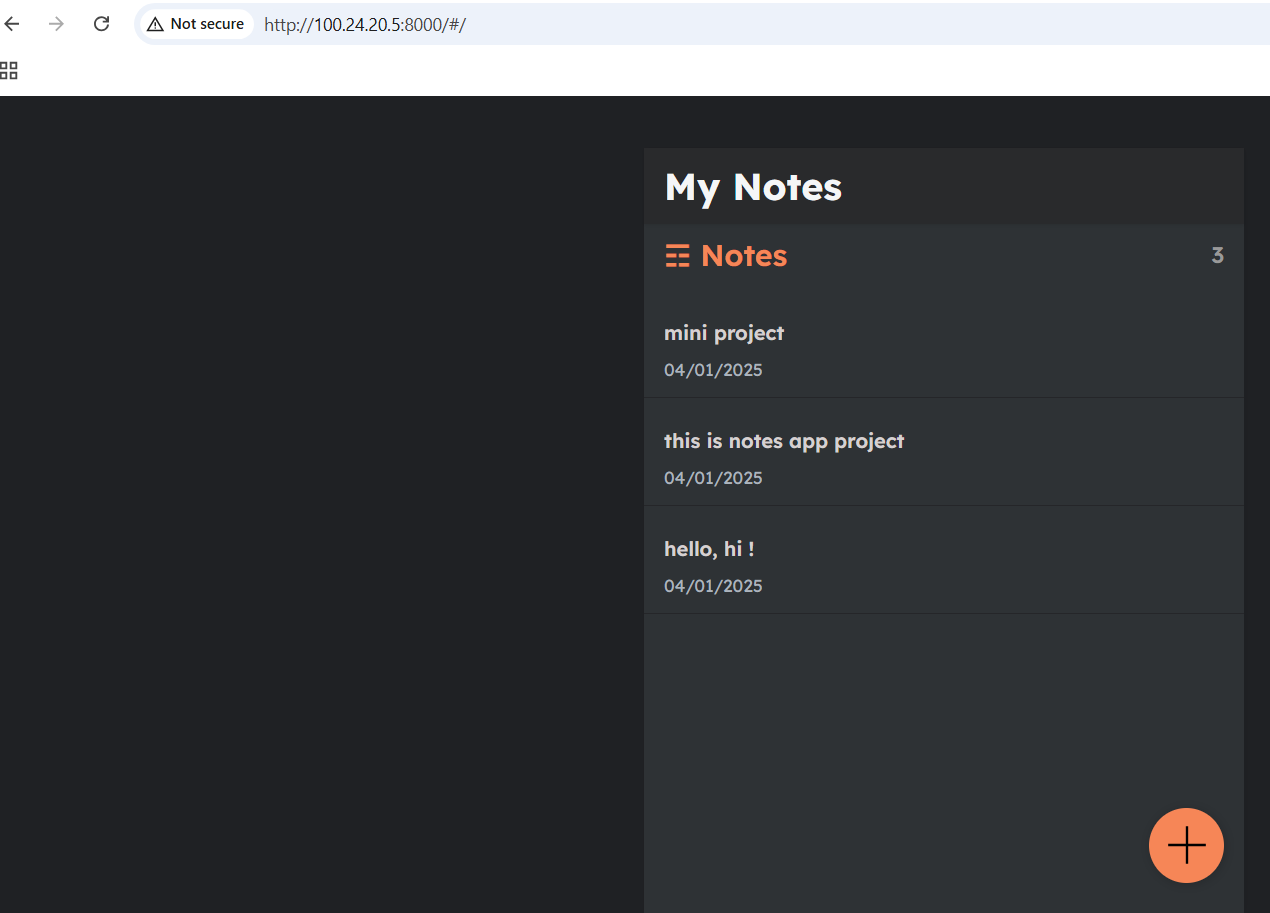


# kubectl apply -f namespace.yml

# kubectl apply -f deployment.yml

# kubectl apply -f service.yml

# kubectl port-forward service/notes-app-service -n notesapp 8000:8000 --address=0.0.0.0



***Ingress:***

Ingress in Kubernetes is a resource that manages external HTTP and HTTPS access to services within a Kubernetes cluster. It acts as a smart router, enabling traffic routing based on hostnames, URLs, or paths, providing advanced features like load balancing, SSL termination, and path-based routing.

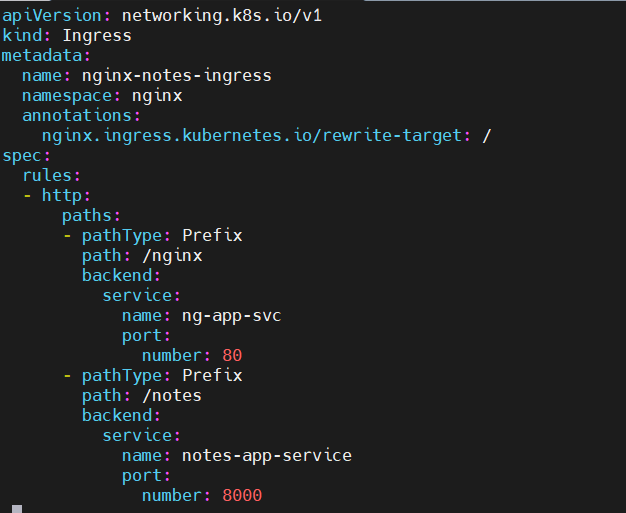
1. **Purpose**:
   * Exposes multiple services using a single external IP address.
   * Centralizes and simplifies traffic management.
2. **Components**:
   * **Ingress Resource**: Specifies the routing rules.
   * **Ingress Controller**: Implements the Ingress resource, processing traffic and routing it to services (e.g., NGINX, Traefik, HAProxy).
3. **Features**:
   * Path-based and hostname-based routing.
   * SSL/TLS termination for HTTPS traffic.
   * Load balancing across backend services.
   * Customizable routing rules with annotations.
4. **How it Works**:
   * User creates an Ingress resource with routing rules.
   * The Ingress Controller processes these rules and configures the load balancer or proxy to manage traffic.

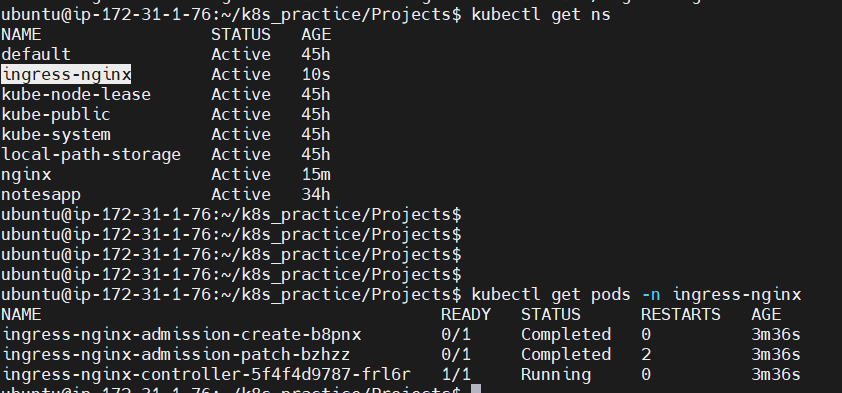
***Annotations:***

Annotations in Kubernetes are key-value pairs used to attach arbitrary, non-identifying metadata to objects like Pods, Services, or Ingress. Unlike labels, annotations are not used for selection or grouping but provide additional configuration or information.

1. **Purpose**:
   * Customize the behaviour of Kubernetes resources.
   * Pass metadata to controllers or external systems.
2. **Common Use Cases**:
   * **Ingress**: Configure specific behaviours like SSL redirection, URL rewrites, or custom timeouts.
   * **Monitoring**: Pass information to monitoring systems like Prometheus.
   * **Metadata**: Attach version information, build data, or debugging hints.
3. **Examples**:
   * **Ingress Annotations**:
4. **Comparison with Labels**:
   * Labels are used for identifying and grouping objects.
   * Annotations are used for attaching extra information or configuration.

# kubectl apply -f <https://kind.sigs.k8s.io/examples/ingress/deploy-ingress-nginx.yaml>

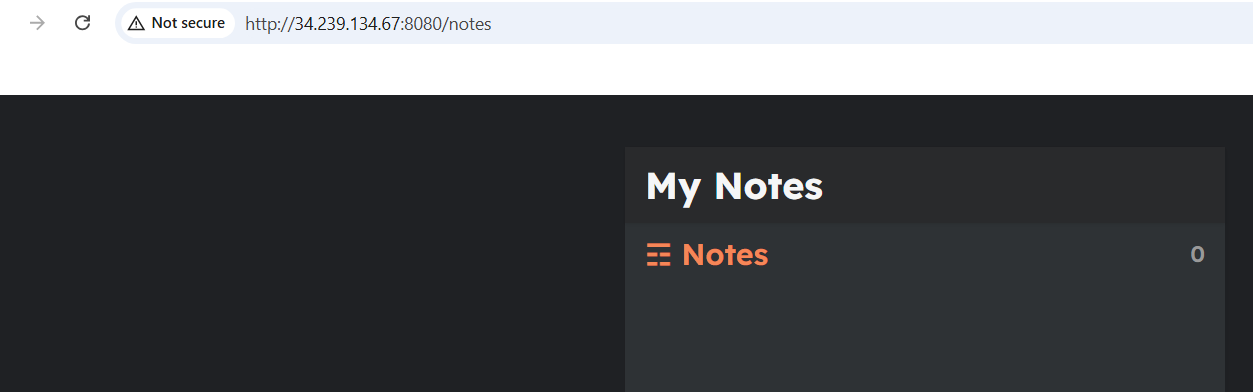


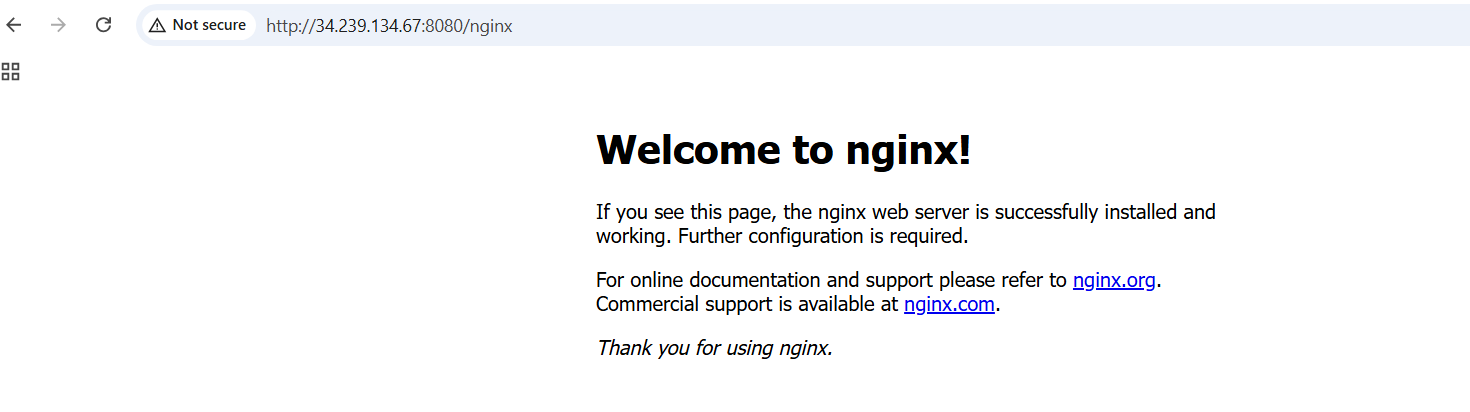
**

# kubectl apply -f ingress.yml

# kubectl get svc -n ingress-nginx

# sudo -E kubectl svc/ingress-nginx-controller -n ingress-nginx 8080:80 --address=0.0.0.0





**StatefulSets:**

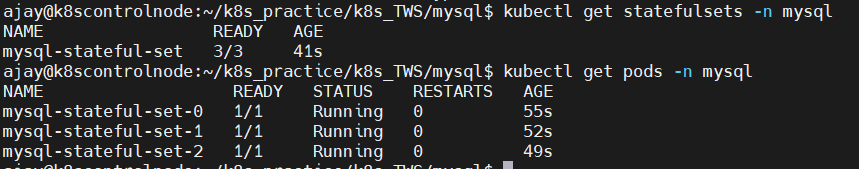
A **StatefulSet** is a Kubernetes controller used to manage and deploy stateful applications. Unlike Deployments, which are designed for stateless applications, StatefulSets maintain a **stable identity** and **consistent storage** for each Pod, even during scaling or rescheduling.

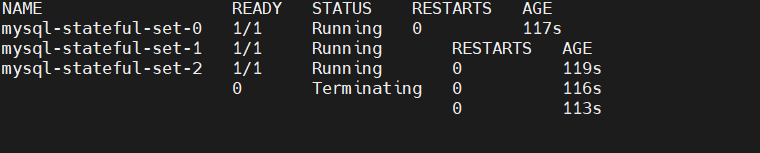
# kubectl apply -f namespace.yml

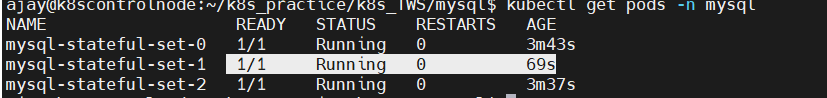
# kubectl apply -f service.yml

# kubectl apply -f st1.yml

# kubectl delete pod mysql-stateful-set-1 -n mysql & watch kubectl get pods -n mysql



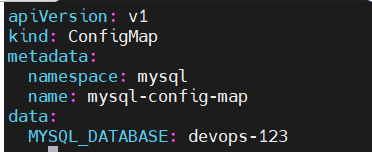


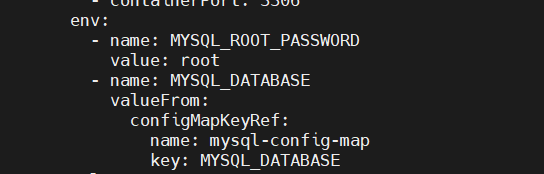


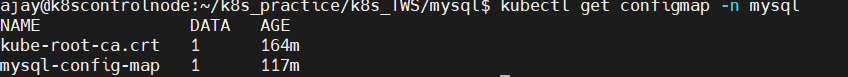


**ConfigMaps:**

A **ConfigMap** in Kubernetes is an API object used to store configuration data as key-value pairs. It is used to decouple configuration details from application code, making applications more portable and easier to manage. ConfigMaps allow you to inject configuration data into your containers at runtime without modifying the container images.



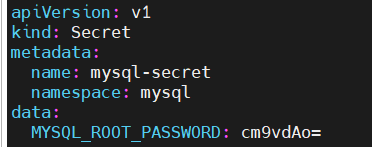


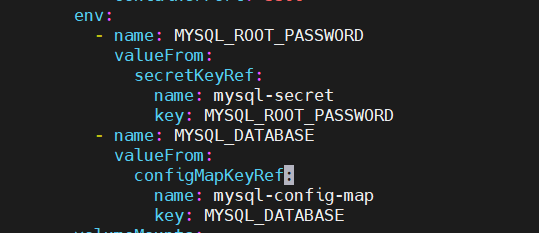


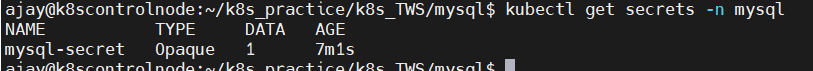
**Secrets:**

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods. Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing sensitive data to nonvolatile storage.







# kubectl get secrets -n mysql

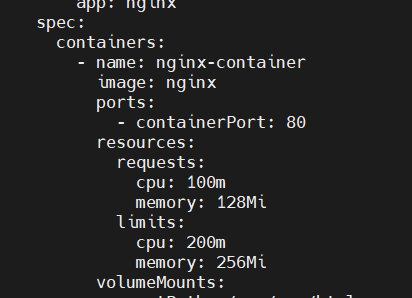
**Resource Quotas and Limits**:

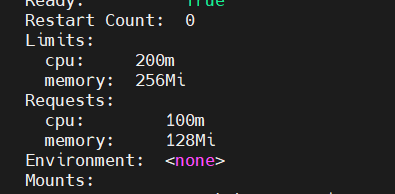
**Resource Quotas**: A Kubernetes mechanism to limit the total amount of compute resources (like CPU, memory, or storage) that a namespace can use. Ensures fair resource allocation among namespaces in a cluster.

**Resource Limits**: Defines the maximum amount of resources (CPU and memory) a container or Pod can consume. Helps prevent resource hogging by individual workloads.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Resource Quotas** | **Resource Limits** |
| **Scope** | Applies to a namespace. | Applies to individual containers/Pods. |
| **Purpose** | Limits resource usage at namespace level. | Prevents containers from over-consuming resources. |
| **Configuration** | Defined in a Resource Quota object. | Set in container specs (limits and requests). |





**Probes:**

Probes are mechanisms used by Kubernetes to check the health of containers within a Pod. There are three types of probes.

**1. Liveness Probe**

* **Purpose**: Determines if a container is still running.
* **Action**: If the probe fails, Kubernetes kills the container and restarts it according to the Pod's restartPolicy.

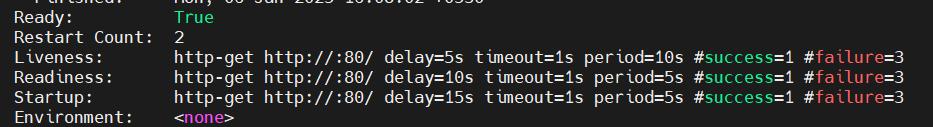
**2. Readiness Probe**

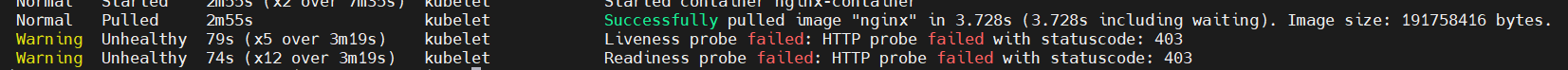
* **Purpose**: Determines if a container is ready to serve requests.
* **Action**: If the probe fails, the container is removed from the Service's endpoint list, stopping traffic from being sent to it.

**3. Startup Probe**

* **Purpose**: Used to check if a container application has started.
* **Action**: Useful for applications with long initialization times. If the probe fails, Kubernetes will restart the container.
* **Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Probe** | **Purpose** | **Action on Failure** |
| **Liveness** | Checks if container is alive. | Restarts the container. |
| **Readiness** | Checks if container is ready. | Removes container from Service. |
| **Startup** | Checks if container has started. | Restarts container if not started. |





**Taints:**

* Taints are applied to **nodes** to indicate that certain pods should **not** be scheduled on them unless the pod explicitly tolerates the taint.
* This ensures that nodes can dedicate resources to specific workloads.

**Taint Effects**:

1. **NoSchedule**: Pods that don’t tolerate the taint won’t be scheduled on the node.
2. **PreferNoSchedule**: Kubernetes will try to avoid scheduling pods on the node, but it’s not guaranteed.
3. **NoExecute**: Pods that don’t tolerate the taint will be **evicted** if already running and **not scheduled** if new.

# kubectl get nodes

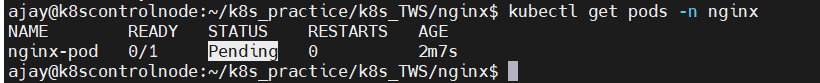
# kubectl get pods -n nginx -o wide

# kubectl taint node my-kind-cluster-worker prod=true:NoSchedule

# kubectl taint node my-kind-cluster-worker2 prod=true:NoSchedule

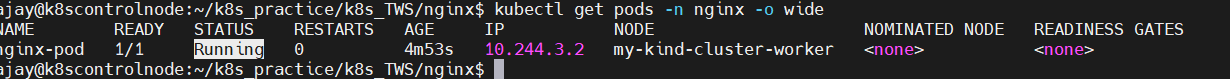
# kubectl taint node my-kind-cluster-worker3 prod=true:NoSchedule

Here I have Tainted all the nodes.



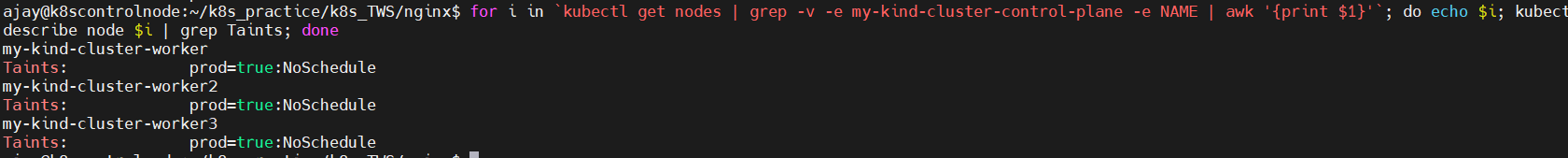
Here I have untainted one of the nodes, we can observe that pod is created in untainted node.

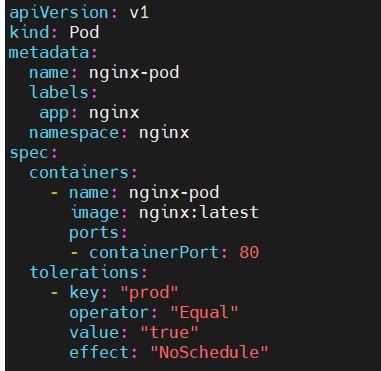
# kubectl taint node my-kind-cluster-worker prod=true:NoSchedule-



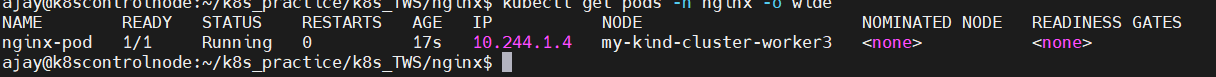
**Tolerations**

* Tolerations are applied to **pods** to allow them to **tolerate specific taints** and be scheduled on nodes with those taints.





By, tolerations in allow to create the pod on nodes.



**Toleration Fields**:

1. **key**: The key of the taint the pod tolerates.
2. **operator**: Can be "Equal" (default) or "Exists".
3. **value**: The value of the taint (optional for Exists).
4. **effect**: The taint effect (NoSchedule, PreferNoSchedule, or NoExecute).

We can observe that all nodes are Tainted, by providing tolerations in pod.yaml file.

**Use Cases:**

**Dedicated Nodes**: Use taints to reserve specific nodes for critical workloads or tenant isolation.

**Eviction**: Use NoExecute to evict non-critical pods during maintenance.

**Scheduling**: Use PreferNoSchedule for "soft isolation."

By combining taints and tolerations, Kubernetes provides fine-grained control over workload placement.

**Horizontal Pod Auto-scaling (HPA):**

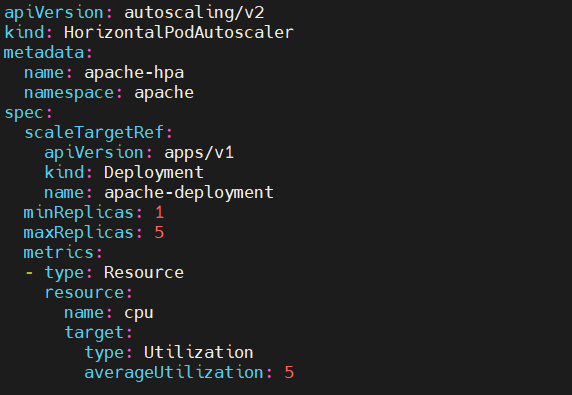
PA automatically adjusts the number of pods in a deployment, replica set, or stateful set based on observed CPU, memory, or custom metrics (e.g., requests per second). It scales out (adds pods) or scales in (removes pods) to maintain application performance under varying workloads.  
Example use case: Scaling web servers during peak traffic hours.

Here we have create apache-deployment with 1 replica.

Added created HPA yaml file for the Deployment.

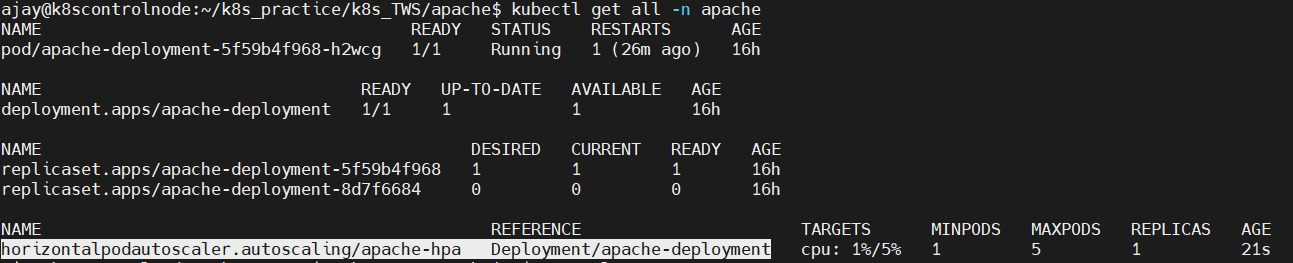
# kubectl get all -n apache

# kubectl apply -f HPA.yml



# kubectl get all -n apache

Below we can observe HPA is created. Now we will create CPU load to check HPA working.



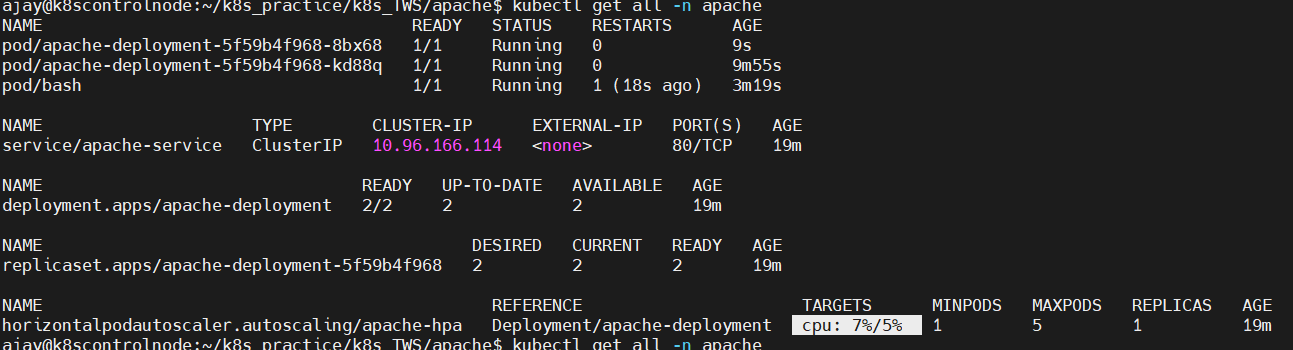
# kubectl get all -n apache

# sudo -E kubectl port-forward service/apache-service -n apache 82:80 --address=0.0.0.0 &

Load Generating container

# kubectl run -i --tty load-gen --image=busybox -n apache /bin/sh

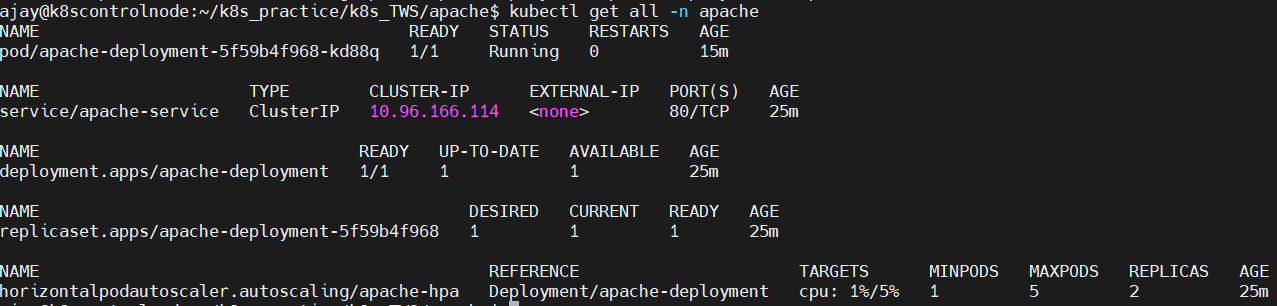
# while true; do wget -q -O- http://apache-service.apache.svc.cluster.local; done



Here we can observe CPU reaches, it started creating pods.

# kubectl delete pod/bash -n apache

I have deleted the load generator pod, it scaled down pods.



**Vertical Pod Auto-scaling** (VPA):

VPA adjusts the resource requests and limits (CPU and memory) of a pod to optimize performance and resource utilization. It recommends or applies changes to running pods, such as increasing memory for a high-consumption app, without altering the pod's replica count.  
Example use case: Optimizing resource allocation for apps with unpredictable memory usage.

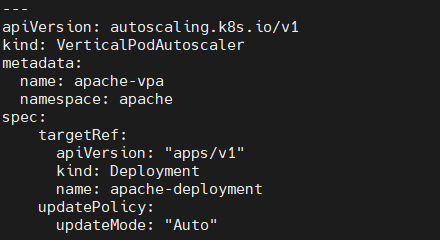
Demo:

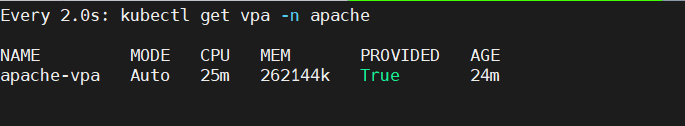
Install all the vertical scaling Dependency.

# git clone <https://github.com/kubernetes/autoscaler.git>

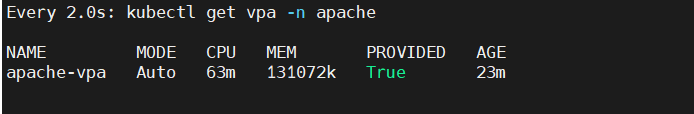
# cd /autoscaler/hack

# . /vpa-up.sh





After applying load we can observe CPU is 63m and MEME 131072k



**Node Affinity**:

**Node affinity** in Kubernetes is a way to control the scheduling of pods onto nodes based on specific rules or conditions. It allows you to specify constraints on which nodes a pod can be scheduled, based on node labels. This feature is part of Kubernetes' **scheduling policies** and helps ensure pods are deployed on suitable nodes.

Node affinity is defined using the affinity field in a pod's specification and is divided into two main types:

**1. Required During Scheduling, Ignored During Execution**

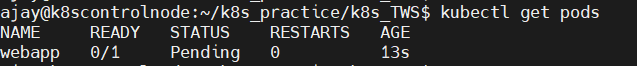
* **Key Behavior**: The scheduler enforces these rules during pod placement. If no node meets the requirements, the pod remains unscheduled.
* **Use Case**: Mandatory rules for pod placement.
* **API Field**: requiredDuringSchedulingIgnoredDuringExecution.

**2. Preferred During Scheduling, Ignored During Execution**

* **Key Behavior**: The scheduler tries to follow these rules but doesn’t enforce them. If no node meets the preference, the pod can still be scheduled elsewhere.
* **Use Case**: Soft preferences for pod placement.
* **API Field**: preferredDuringSchedulingIgnoredDuringExecution.

# kubectl get nodes --show-labels

# kubectl label node <node name> <key>=<value>



# kubectl label node my-kind-cluster-worker app=front-end





**Required**: The pod will only be scheduled on nodes with node-role=frontend.

**Preferred**: Among those nodes, the scheduler prefers nodes in zone=us-east-1.