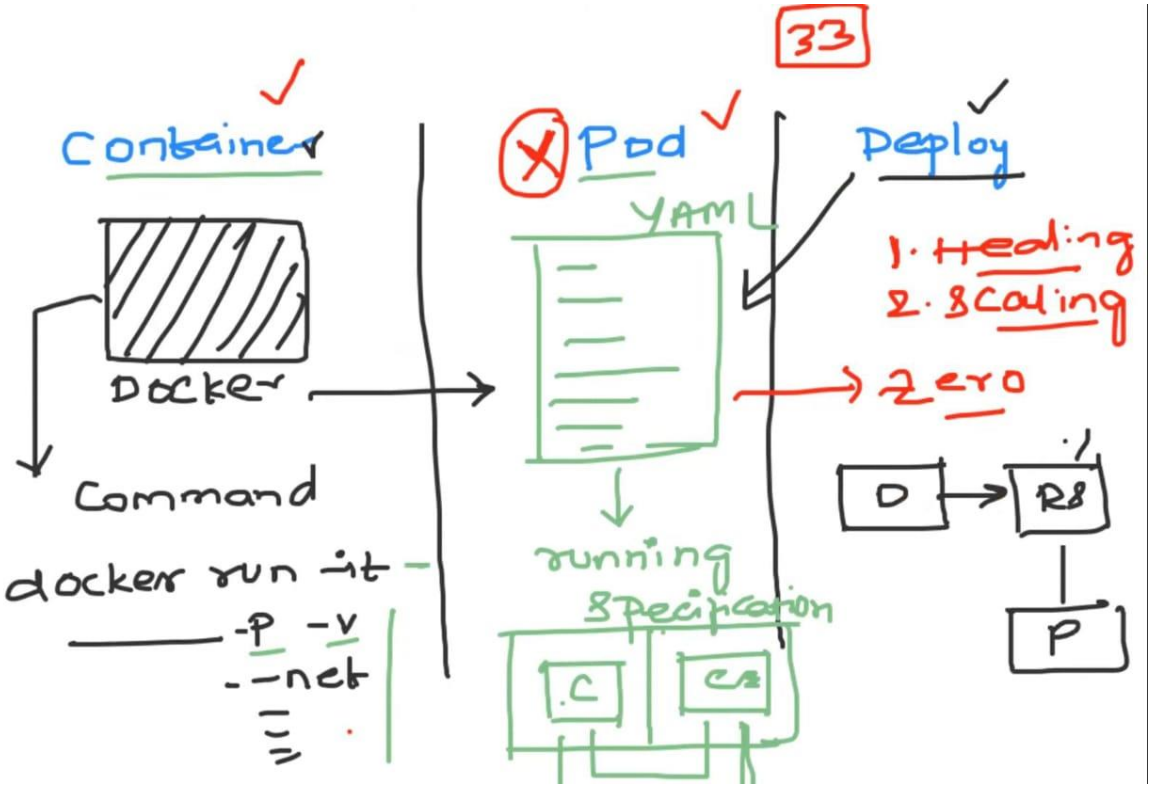


Container : used to run containers

Pod : It is also run containers but in a pod we can run multiple containers and pod is created using yaml file

Deploy : Deploy is used to create and manage number of replica sets which is used for auto scaling and auto healing

---

kubectl commands

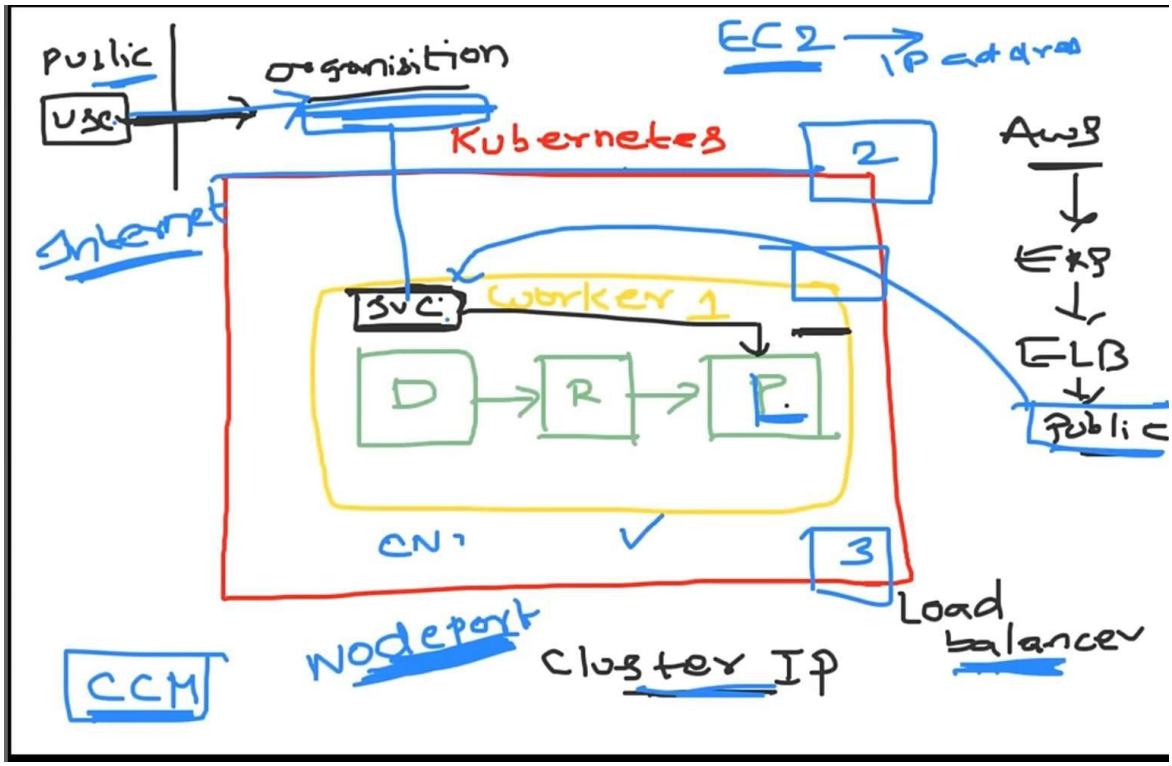
vim config.yaml & kind create cluster --config confid.yaml --name my-cluster --> it is used to create cluster using yaml file

kubectl get nodes --> It is used to see number of nodes in that cluster

kubectl config get-contexts --> to see all clusters on machine

kubectl config use-context <cluster name> --> to switch from one cluster to another cluster

---



Kubernetes service network types

* Cluster ip --> It can be act like conditional access, it can be accessed by only set of users which have permission to that pod

* Nodepod --> nodepod can be accessible by organization. anybody can access this pod in same organization if they have permission to EC2 or VNet like that

* Load balancer --> load balancer is nothing but a public ip and can be accessed by anyone from the world.

How load balancer will works?

In cloud, when pod say I want to deploy on load balancer, the azure will assign public ip to that pod through AKS and this all work on under CCM (cloud control management)

Cluster ip --> It can be act like conditional access, it can be accessed by only the group of node users who are in same cluster

Difference between contexts and clusters

Cluster = The actual Kubernetes environment.

Context = A shortcut to easily switch between different clusters, users, and namespaces.

Example:

kubectl config get-clusters

output:

dev-cluster

prod-cluster

kubectl config get-contexts

output:

CURRENT	NAME	CLUSTER	AUTHINFO	NAMESPACE
	dev-context	dev-cluster	dev-user	dev
	prod-context	prod-cluster	admin-user	default

K8 have L4 level load balancing which is round robin algorithm defaultly. For advance load balancing we have to go for cloud.

In Kind, all networking is managed by Docker's networking system, specifically through the Docker bridge network. Unlike Minikube, which assigns an external IP directly, Kind does not automatically provide an external IP for Ingress resources.

To see network for kind,

docker network inspect kind

```
d
ubuntu@ip-172-31-89-68:~/day33/Flask/k8s$ cat ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-world
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: "example.com"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: hello-world
            port:
              number: 80
ubuntu@ip-172-31-89-68:~/day33/Flask/k8s$ kubectl get pods -A
```

In this ingress file,

annotations is important one because we want to change on annotations of what load balancing algorithm we going to use.

In spec, there is ingressClassname..

Why ingressclassname?

When we use multiple load balancers on one machine, without ingressclassname it will collapese with each load balancers. So when we use ingressclassname it will load balance with that particular load balancer.

Host --> is a domain name which want load balancer

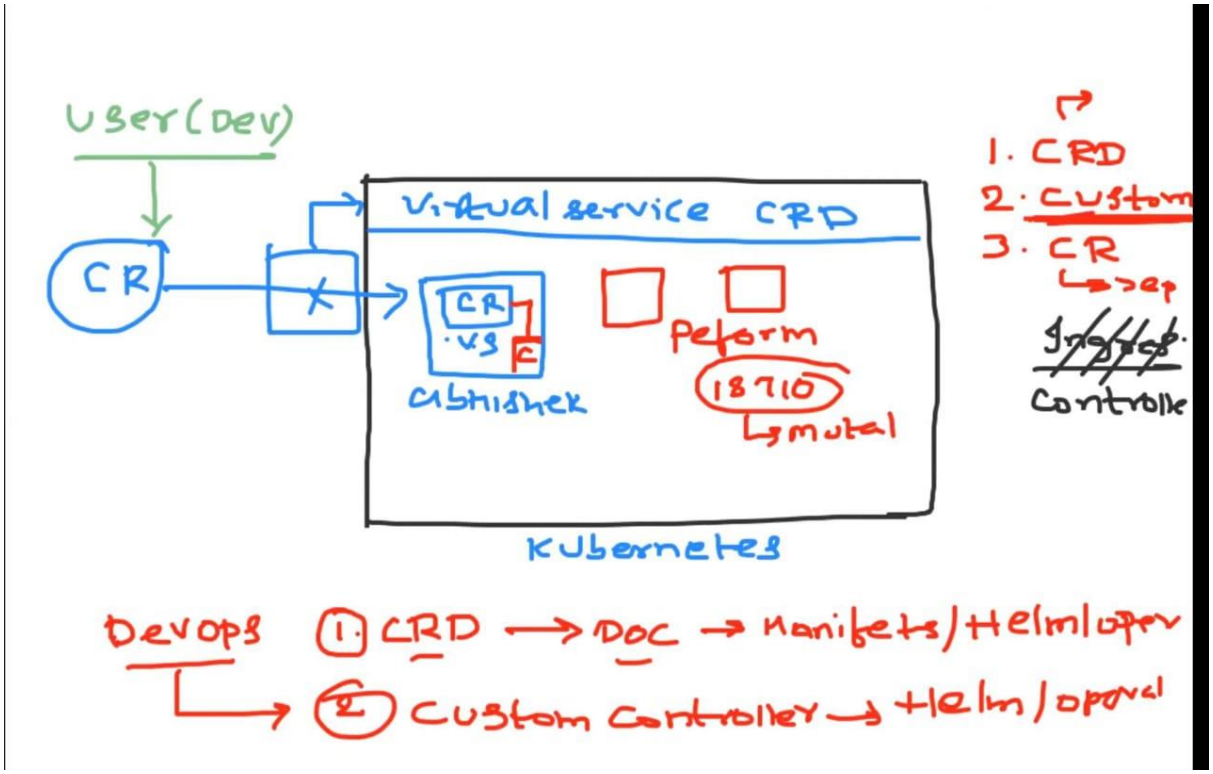
Path / → Any request to / on example.com will be forwarded.

Load Balancing Method	Annotation
Round Robin (default)	*(No change needed)*
Least Connections	`nginx.ingress.kubernetes.io/load-balance: "least_conn"`
IP Hash (Sticky Sessions)	`nginx.ingress.kubernetes.io/upstream-hash-by: "$remote_addr"`
Random	`nginx.ingress.kubernetes.io/load-balance: "random"`
Weighted Load Balancing	`nginx.ingress.kubernetes.io/rvice-weight: ervice1=80, service2=20"`

Nginx defaultly used round robin algorithm only...

If we want change the load balance algorithm the commands or shown here

---



How to create CR and CRD on local machine?

First, we have to create CRD before create CR... and then we have to create Custom controller and finally we have to create CR.

CDR --> created by Devops engineer on local machine

CR --> created by user or Devops engineer

Custom controller --> created by devops engineer in the language which is famous called golang

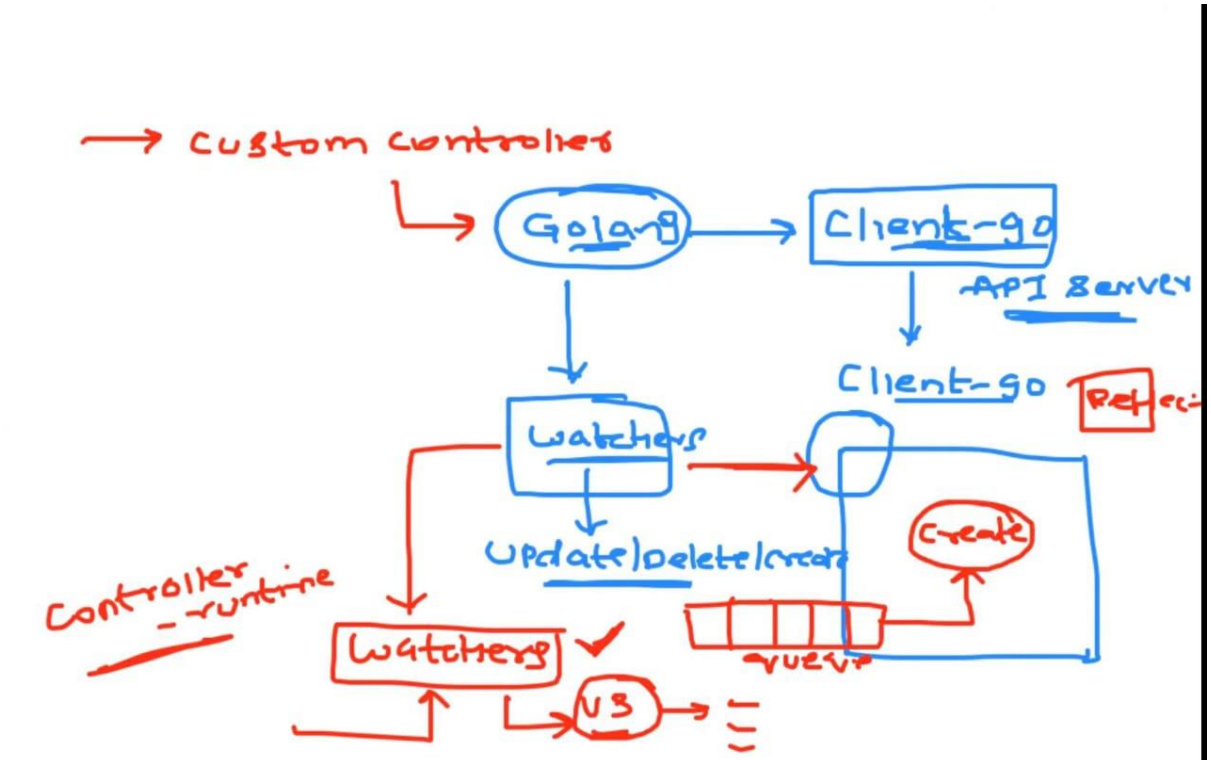
---

When you use kubectl port-forward svc/httpd-service 9010:80, it forwards traffic from your local machine to one of the pods backing the httpd-service. However, port-forwarding in Kubernetes works in a single-pod binding manner and does not automatically balance traffic across multiple pods.

ReplicationController is an older version and is now deprecated in favor of ReplicaSet.

ReplicaSet is used to ensure that a specific number of pod replicas are running.

Deployment is the recommended method for managing applications in Kubernetes, as it provides features like rolling updates, rollbacks, and declarative management.

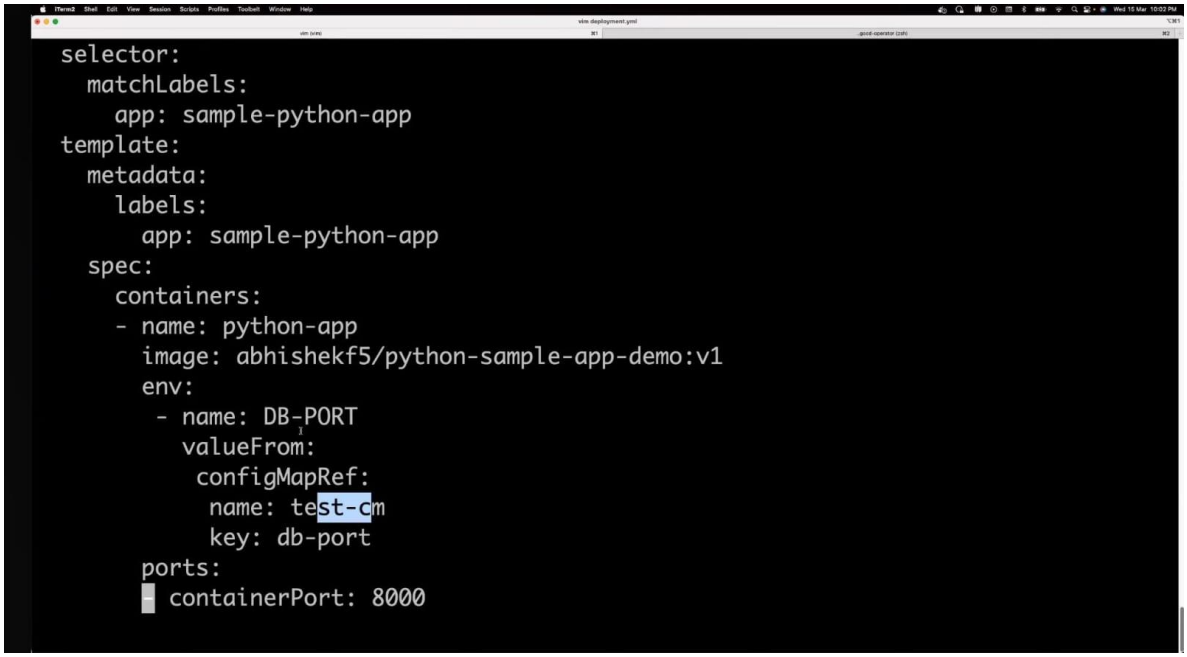


How controller will works?

In controllers, there is a watcher which will watch the update, Create, delete resources. Naturally there is a watchers for default resources like services, ingress in k8 but when we create a custom resource we have to create new watcher and there is CNCF community which will provide watcher framework for use

After watcher detected it will notify to the reflector which is used to add to the queue... After reflector put it in the queue it will start processing like create a resource or update the resource

---



```
  selector:
    matchLabels:
      app: sample-python-app
  template:
    metadata:
      labels:
        app: sample-python-app
    spec:
      containers:
      - name: python-app
        image: abhishekf5/python-sample-app-demo:v1
        env:
         - name: DB-PORT
           valueFrom:
            configMapRef:
             name: test-cm
             key: db-port
        ports:
        - containerPort: 8000
```

In env, This like we have to give configmap variables to the pod.

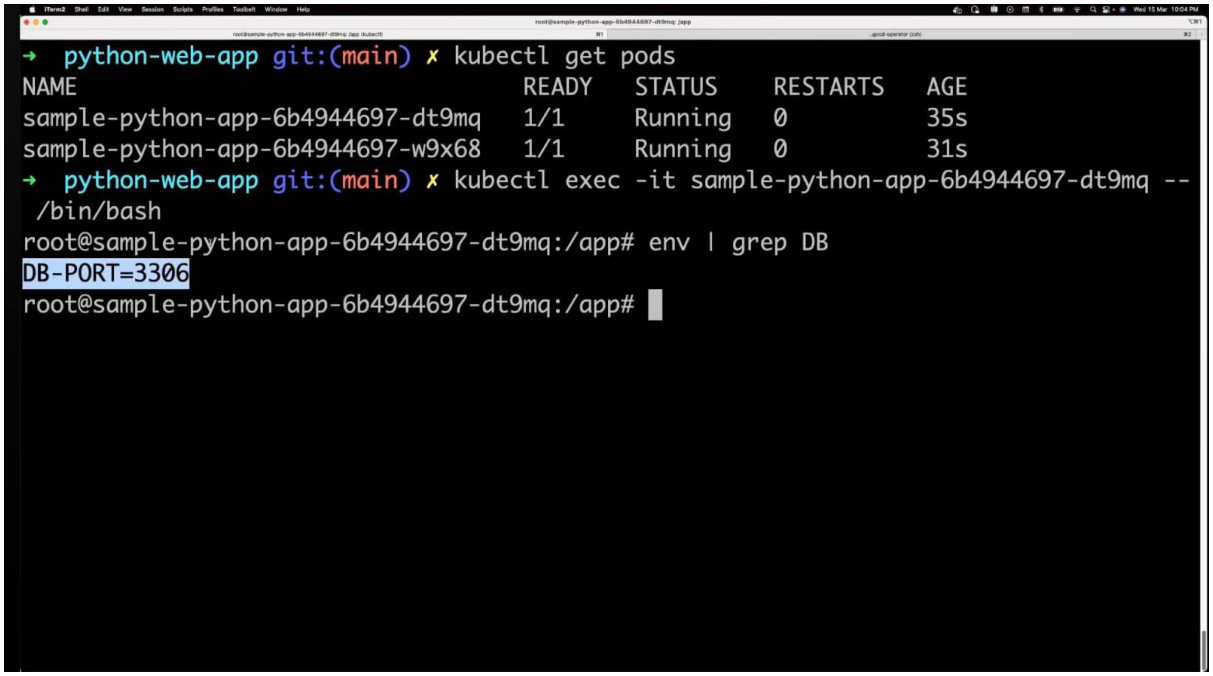
name --> given by us

valueFrom --> Refered as where the values we will get from

configMapKeyRef --> is refered as get the keys from configmap

name --> is which name we will give on the configmap

key --> is refered as which variable we going to use

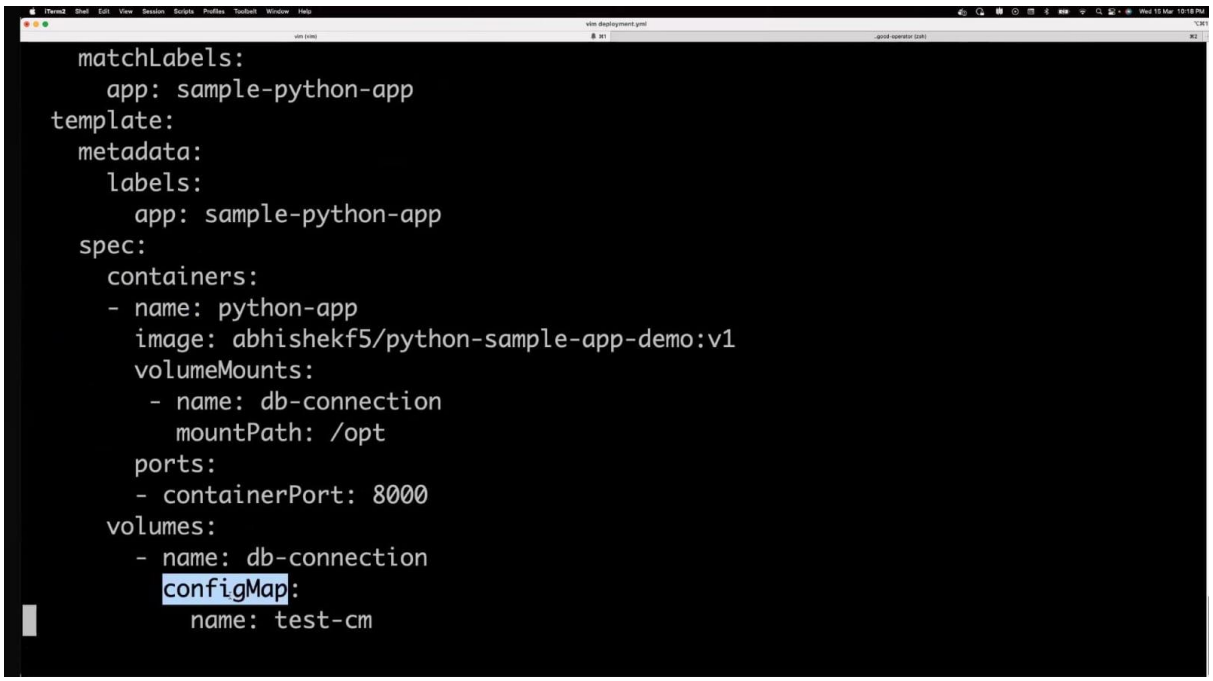


```
→  python-web-app git:(main) ✗ kubectl get pods
NAME                                 READY   STATUS    RESTARTS   AGE
sample-python-app-6b4944697-dt9mq    1/1     Running   0          35s
sample-python-app-6b4944697-w9x68    1/1     Running   0          31s
→  python-web-app git:(main) ✗ kubectl exec -it sample-python-app-6b4944697-dt9mq --
 /bin/bash
root@sample-python-app-6b4944697-dt9mq:/app# env | grep DB
DB-PORT=3306
root@sample-python-app-6b4944697-dt9mq:/app#
```

This is how we check is this variable is granded or not

Once we assigned variables to the pod we can't able to edit or update the variable, we have to terminate the variable and we have create new pod because k8 not allow to overwrite the variables in pod

---



```
    matchLabels:
      app: sample-python-app
  template:
    metadata:
      labels:
        app: sample-python-app
    spec:
      containers:
      - name: python-app
        image: abhishekf5/python-sample-app-demo:v1
        volumeMounts:
         - name: db-connection
           mountPath: /opt
        ports:
        - containerPort: 8000
      volumes:
        - name: db-connection
          configMap:
            name: test-cm
```

How to change the variable without terminate the pod?

There is a concept called volume and volume mount which is used to change the variable without terminate the pod

How to use it?

Volume

name --> Given by us

configMap --> which is refered for variables

name --> name of the variable which we have given on the configmap

Volume mounts

name --> Given by us

MountPath --> is refered as where we have to mount(save) the variable on kubernetes pod

---

To create secrets

kubectl create secret generic test-secret --from-literal=db-port= "3306"

generic --> third party encryption resource which is used for encryption

test-secret --> file name of the secret

db-port= "3306" --> which is used to encryption in a key value pair model

How to see is encrypted or not?

Kubectl describe secret test-secret

It shows the test-secret file

How to see the encrypted data in local machine?

echo [(MzMwNg==)encrypted data] | base64 –decode

How to check api version of our pod in local machine?

* kubectl explain pod

Kubectl edit pod nginx-pod --> is act like a Vim editor which is used to edit a pod in k8

kubectl run nginx --image=nginx --dry-run=client

What is --dry-run=client?

--dry-run=client means the command simulates the creation of a resource without actually applying it to the cluster.

Does not create the Pod—only shows what would happen.

Runs on the client side (kubectl), without sending the request to the Kubernetes API server.

Used for validation and previewing the YAML output before applying.

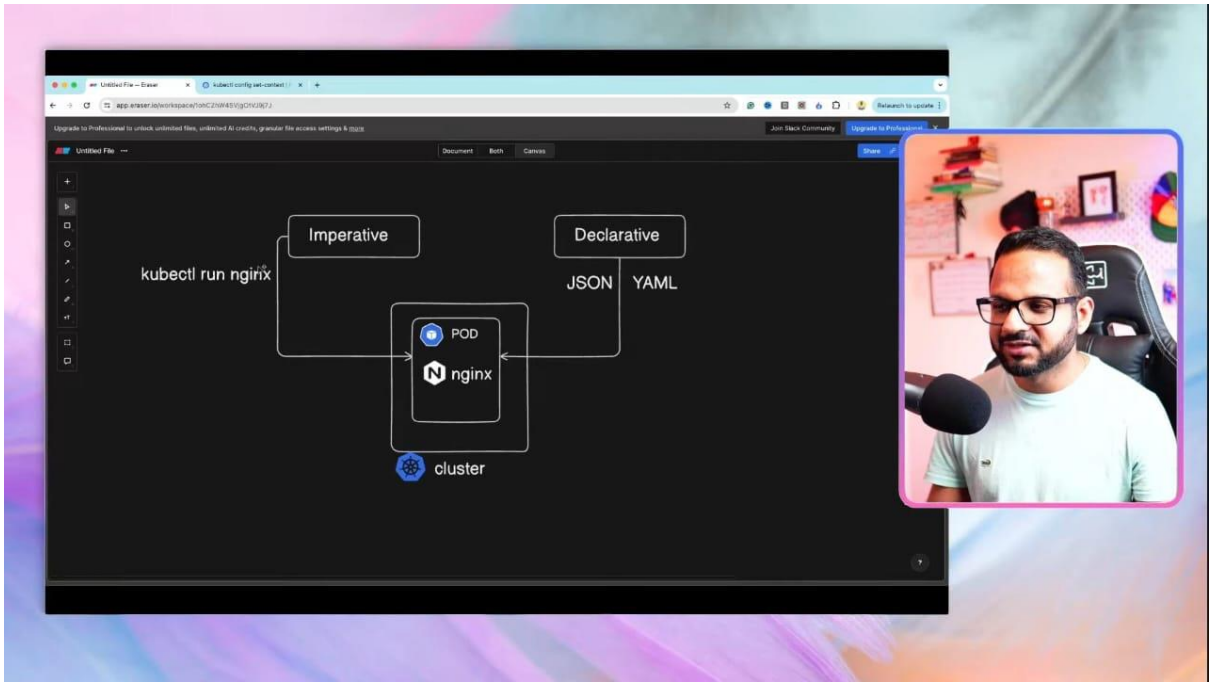
---

kubectl run nginx --image=nginx --dry-run=client -o yaml --> which is used to show how imperactive command can create pod in the format of yaml

kubectl run nginx --image=nginx --dry-run=client -o yaml > nginx-new. yaml --> it is used to store the format of yaml that have created by imperactive, that yaml file stored in nginx-new.yaml file

kubectl get pods nginx-pod --show-labels --> It shows the labels for each pod

---

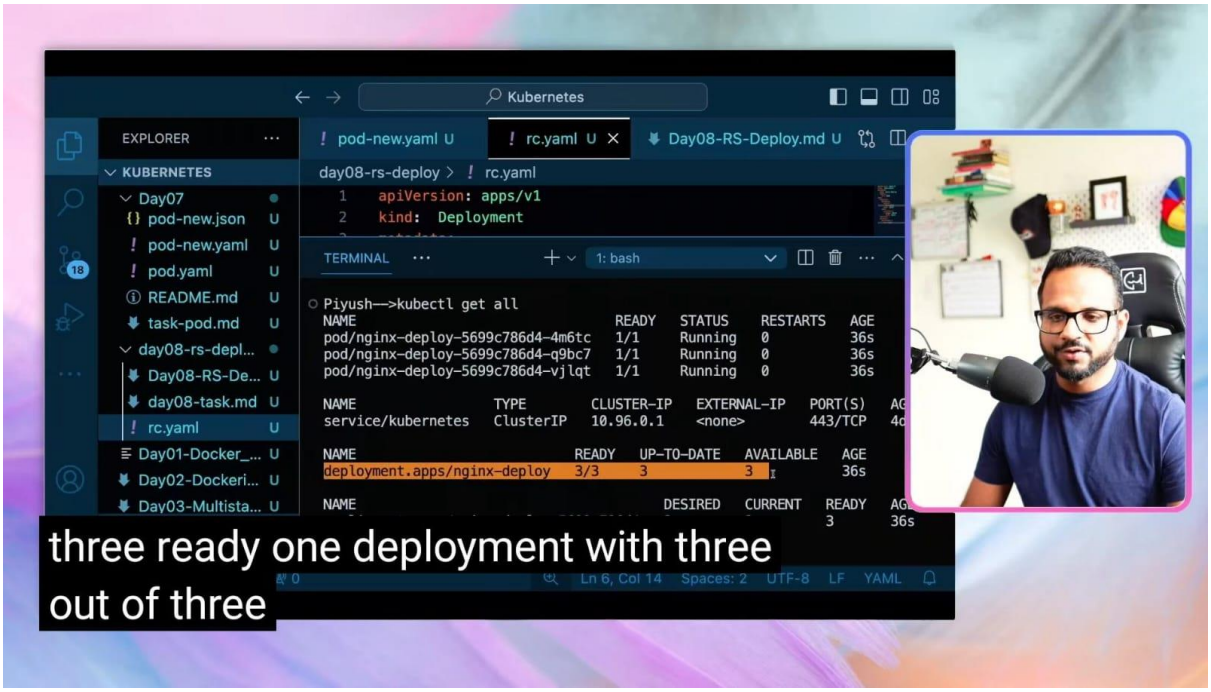


Two types to create pod in k8. They are,

* Imperative --> which is used as kubectl in terminal just like adhoc commands

* Declarative --> It is used to create a pod through yaml file just like playbooks and k8 supports both JSON and YAML format to create pod

---



kubectl get all --> used to show in one command(deployment, replicaset, pods and etc)

kubectl set image deploy/nginx-deploy nginx=nginx:1.9.1 --> is used to update image

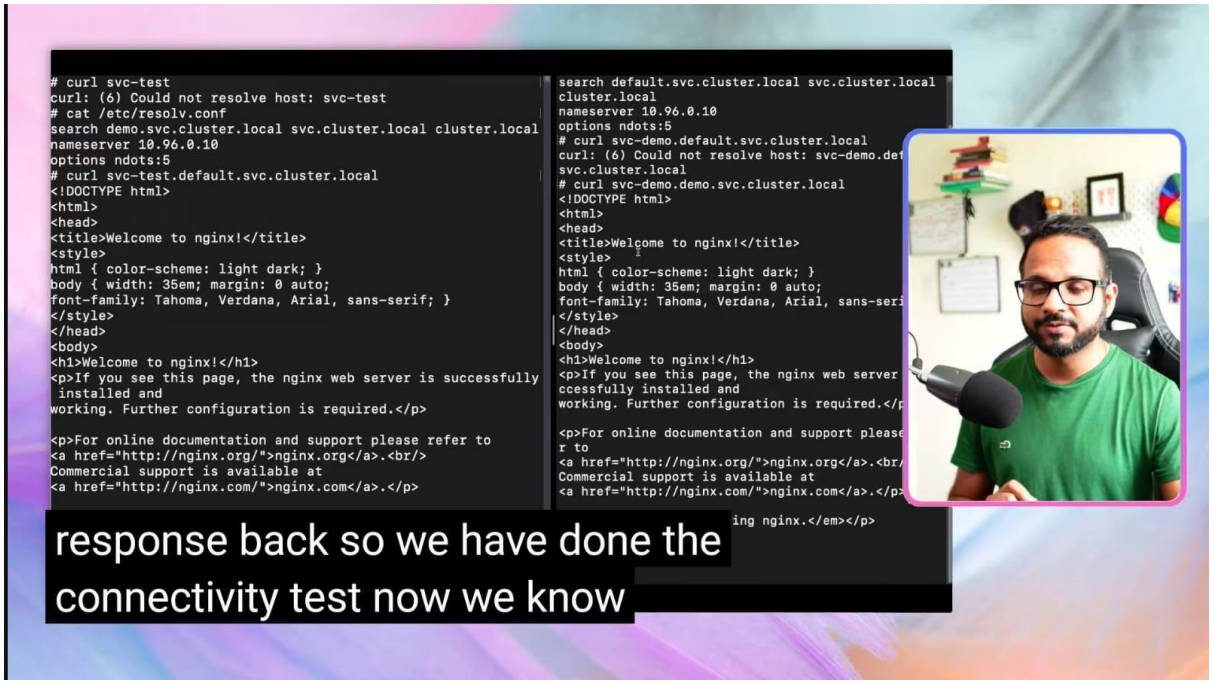
How it will update image when a pod is production?

Ex: when a 3 pods are running on production, this command will update the pods one by one without interference to the user for use the app and if one pod will on update this will redirect the requests to another pods

kubectl rollout undo deploy/nginx-deploy --> it will come to before version of nginx are if changes any on that it will come back to that

How to check it will come back?

kubectl describe pod <podname>



When we create service, we cant able to access another pod things on our pod in different namespace...

* we can't able to access another pod on our pod in different namespace using host name rather than we use IP but ip is changed dynamically if we once restart

How to access another namespace pod service on our pod?

* we can use FSDN(full stack domain name)

How to do it?

* we have to get into the pod and write command (cat /etc/resolv.conf) it shows the ip and FSDN, we take that domain name and use it on different pod

---

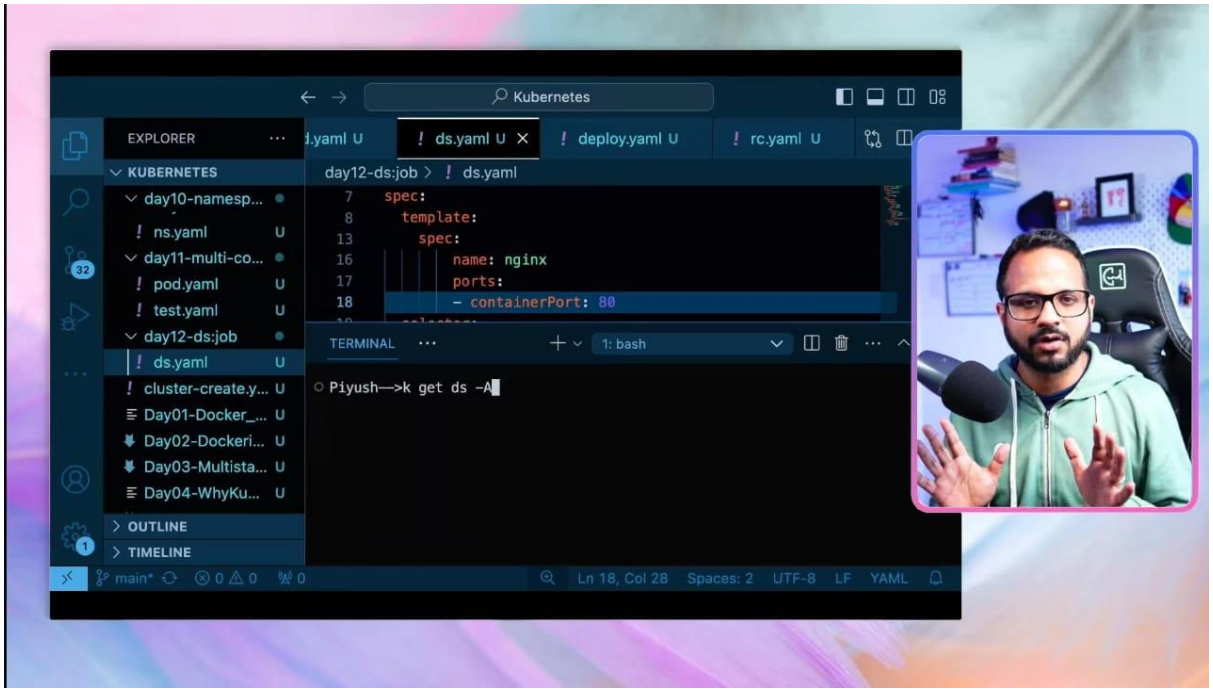
https://youtu.be/yRiFq1ykBxc?feature=shared

K8 multi container

Kubectl get pod --selector <labelname> --> used to select particular labels in group of labels(it is only used metadata labels) are shown below

Label Type	Defined In	Purpose
Selector Labels	`.spec.selector.matchLabels`	Helps controllers (Deployment, Service) select Pods
Metadata Labels	`.metadata.labels`	General labels for identification, filtering, or grouping resources
Template Labels	`.spec.template.metadata.labels`	Specifies labels assigned to new Pods; must match selector labels

---



Kubernetes Daemen set

Kubernetes scheduler actually performed as it assigns a pod based on the cpu, memory and space of the node in cluster but there is a thing called daemon set and it performed as when there is an 3 node, it create a 3 pod and if there is a two node, it create a 2 pods

If there 3 node(2 worker Node and one control node), it actually creates a pod only on worker node not an worker node

How to create it?

* It more than same like deployment but in kind we have to set daemon set and we dont have to mention replicas

kubectl get pods -A --> It shows all namespace components

Taints and Tolerations

Taints (Applied on Nodes): Prevent pods from being scheduled on a node unless they have a matching toleration.

Tolerations (Applied on Pods): Allow a pod to be scheduled on a tainted node if it has a matching toleration.

Example Usage:

Keep workloads away from specific nodes (e.g., GPU nodes for ML workloads).

Dedicated nodes for high-priority applications.

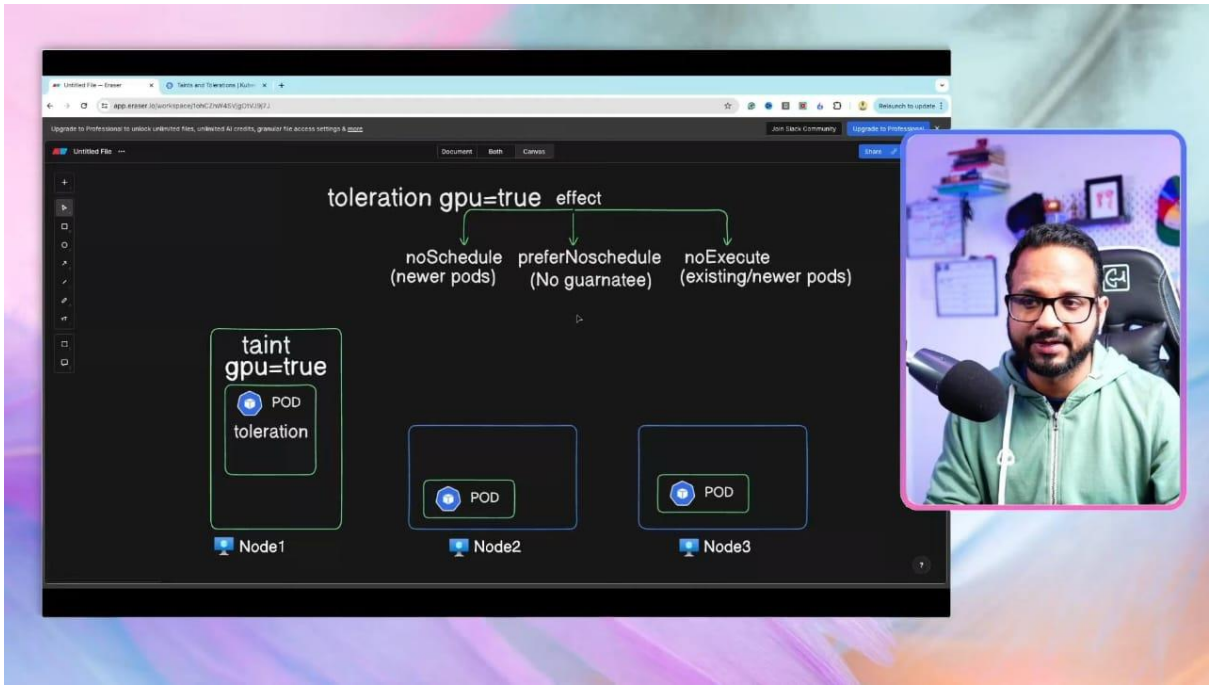
Node selector

Node Selector: Assign pods to specific nodes using labels.

Example Usage:

Schedule pods only on nodes with specific hardware or OS.

Avoid running multiple replicas on the same node.



How to assign taint to the node?

kubectl taint node <node-name> gpu=true:NoSchedule

gpu=true --> which is key value for assign taint & tolerant in both node and pod

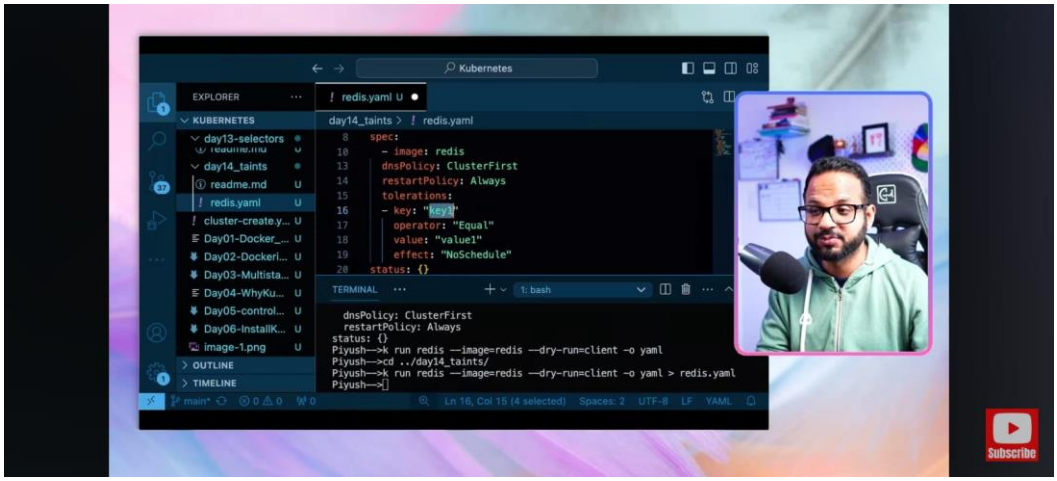
NoSchedule --> It is used to which pod which we have assign

How to see taint is assigned to node?

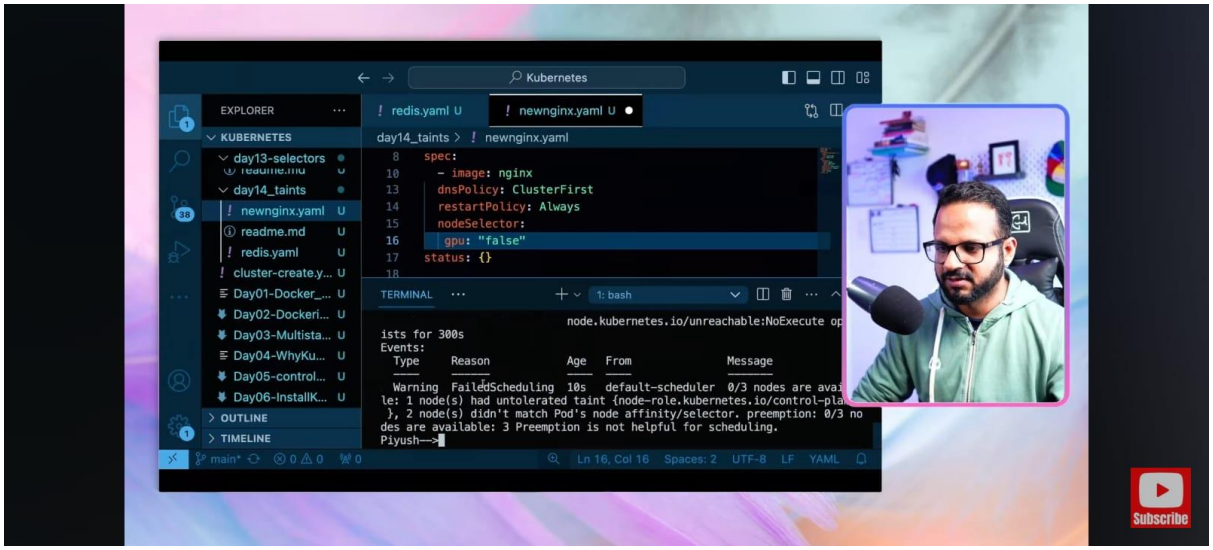
kubectl describe node <node-name> |grep -i taint

How to remove taint on node?

kubectl taint node <node-name> gpu=true:NoSchedule-



This is how we will assign Toleration on yaml file for pod.



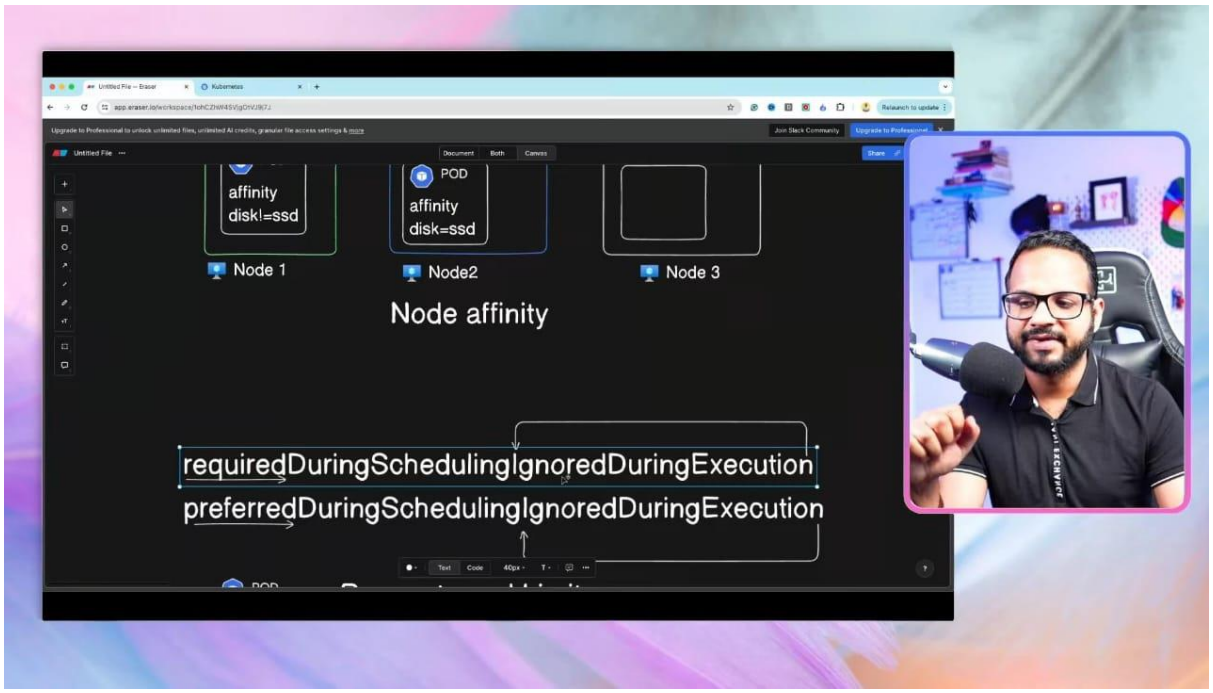
This is how select node selector on pod

What is label?

* Label is used to find nodeselector on pod

Label-for node

Nodeselector - for pod

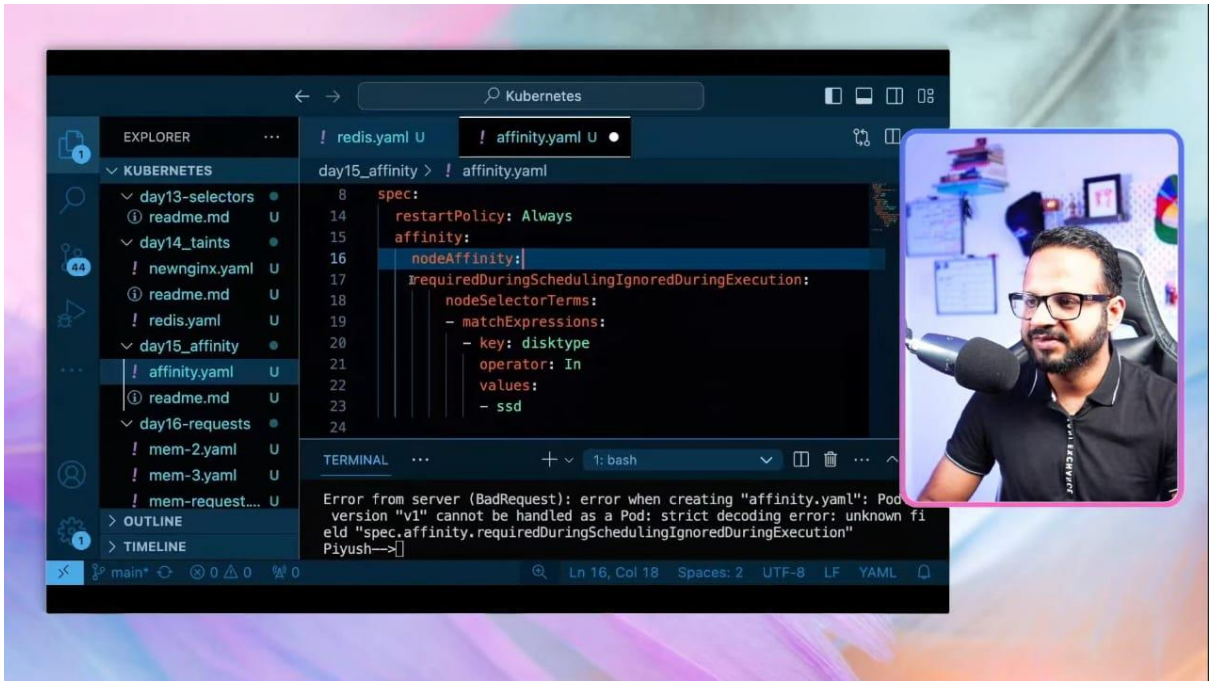


Node affinity have two types

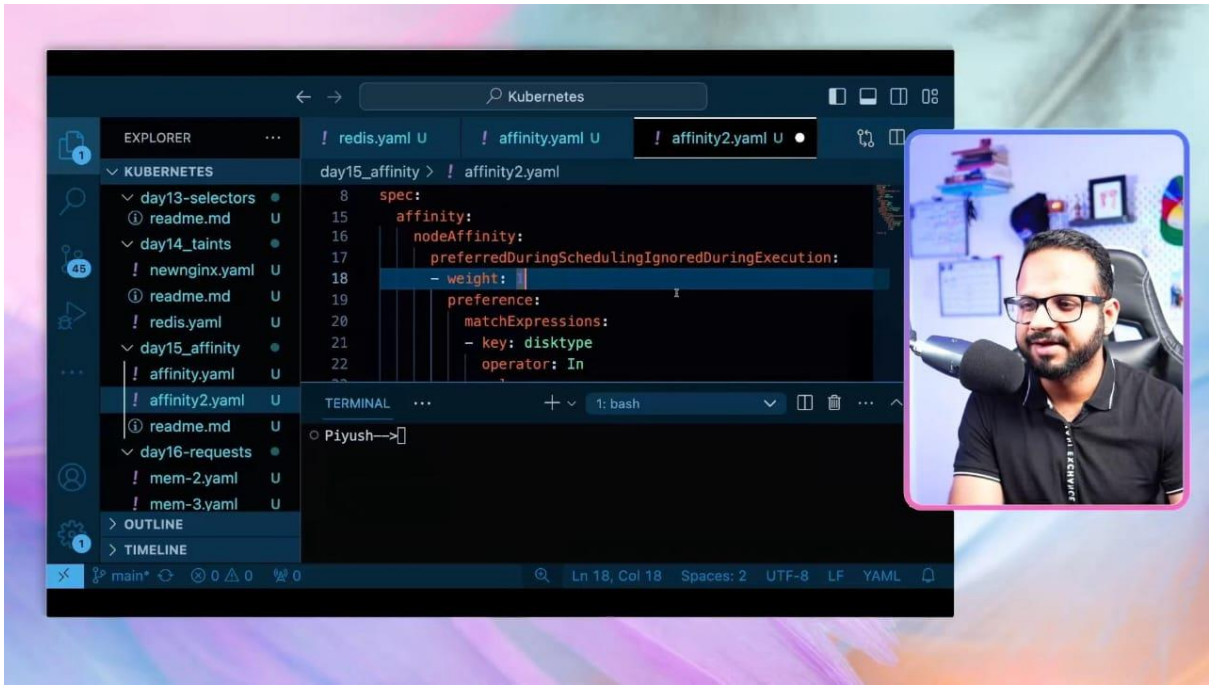
* requiredDuringSchedulingIgnoredDuringExecution --> It will assign the pod exactly on the same label(example: If the label is true and the nodeselector also true, it will assign only on that partucular node)

* preferedDuringSchedulingIgnoredDuringExecution --> It will also assign the pod on the same label but if that label is not have in the node it will assign on node(randomly)(example: If the nodeselector also true but label is not there in node, it will assign a pod on any nodes randomly but before assign to the random node it will check if the label is avilable). This is how preferredDuringSchedulingIgnoredDuringExecution works

---



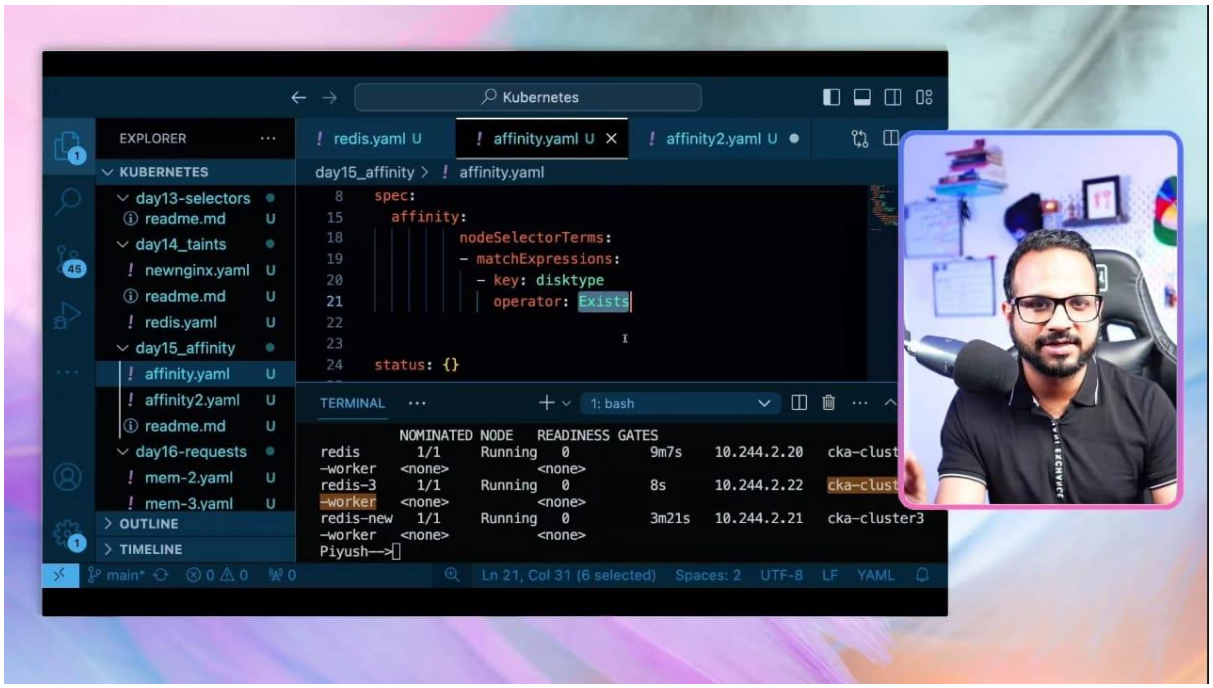
Nodeaffinity for requiredDuringSchedulingIgnoredDuringExecution.



Nodeaffinity for preferredDuringSchedulingIgnoredDuringExecution

Preference --> It is used for checks the nodes if the required label is avilable before assigned to the random nodes

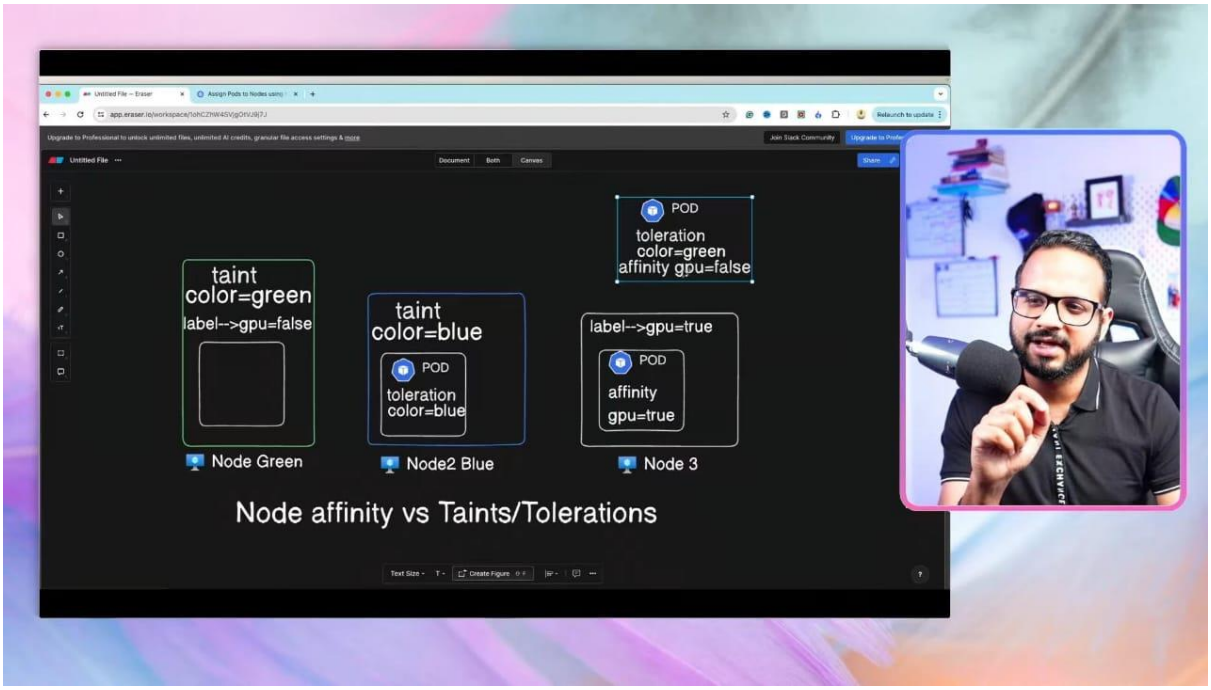
---



We can able to write the Nodeaffinity even without value and it works

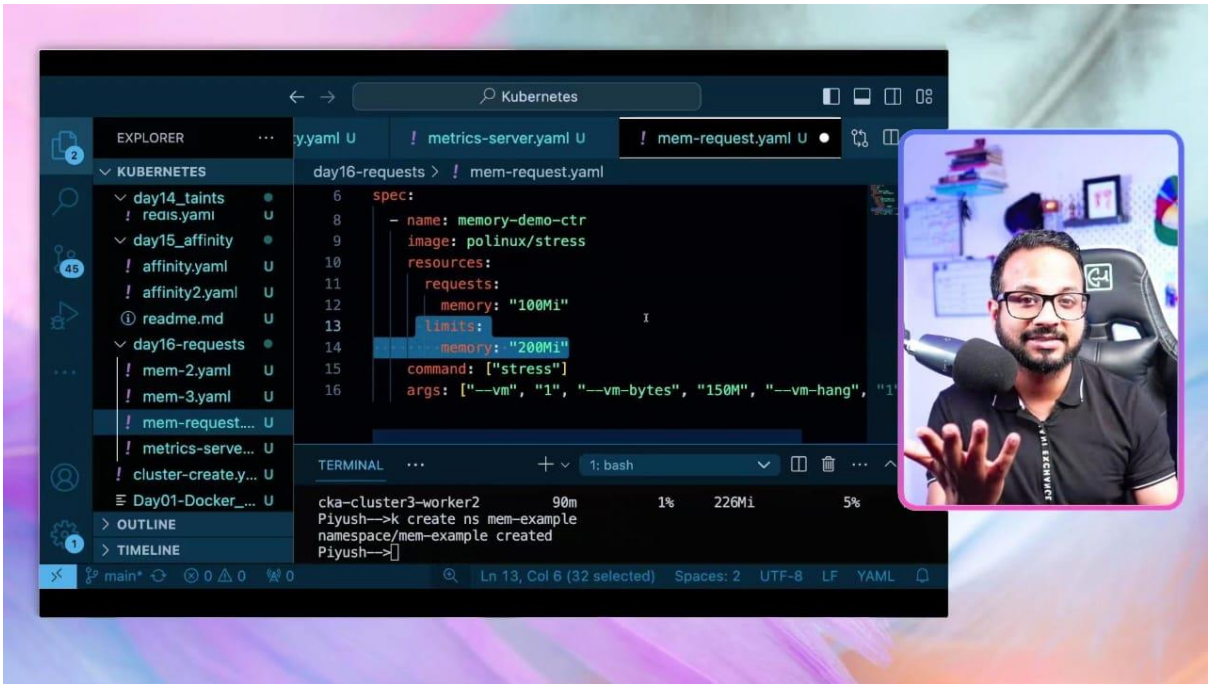
How to create label without value(only key)?

kubectl label node <node-name> disktype=



We can able to create both taint & toleration and Nodeselectors & labels on same yaml file for better performance

---



Before creating pod with limitable metrics, we have to set up the node with metric-server.yaml.

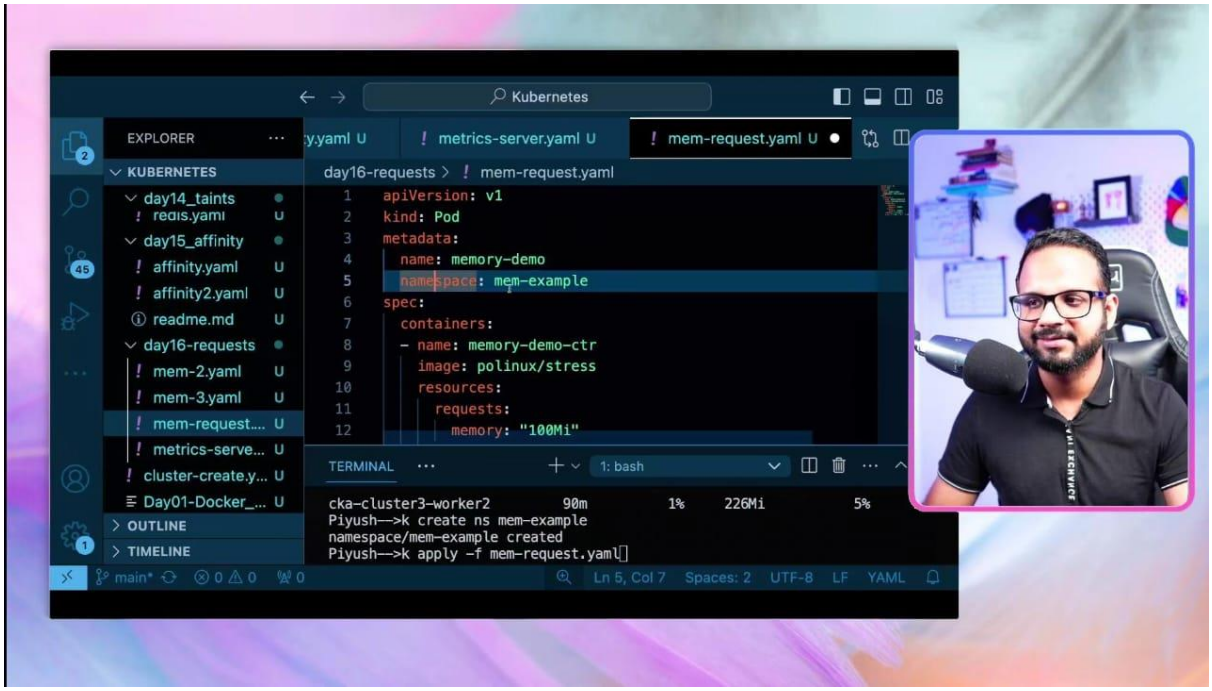
What is Metric-server?

* Metric server is used to show the metrics of the node(cpu utilization, memory utilization)

To create a pod with limitable metrics example have been shown above,

For better understanding refer k8 documentation

--> OOMKilled is Insufficient memory space error

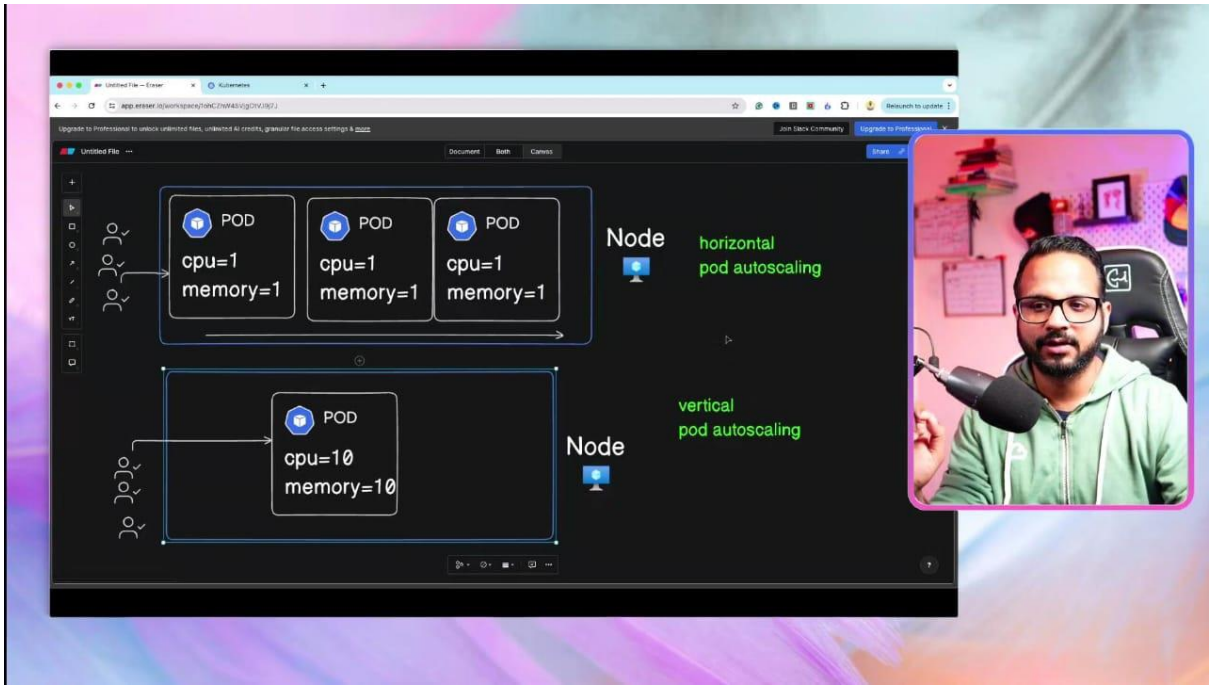


We can able to write namespace in two methods,

One is in Imperactive mode(kubectl apply -f <podyaml> -n <namespace>

Another one is we can write in yaml file itself under metadata

---

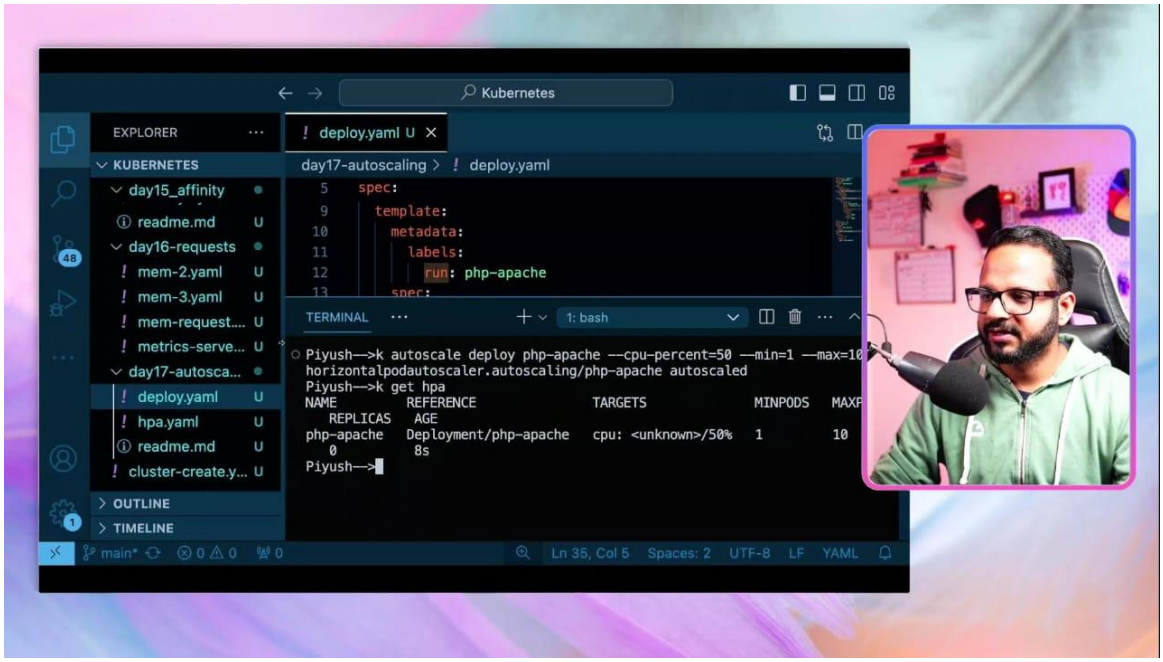


Horizontal Autoscaling(HPA)

* HPA is defaultly have on the k8 and it use metric-server for check the CPU utilization and increase the pod based on the demand. Even HPA is default we have set metric-server manually

Vertical Autoscaling(VPA)

* VPA is not defaultly on the k8, we have to setup manually and VPA increase the size of the node rather than increase pod. VPA requires some downtime to increase the size of the node(YONO Sbi app which takes downtime)



How to autoscale a deployment a deployment pod?

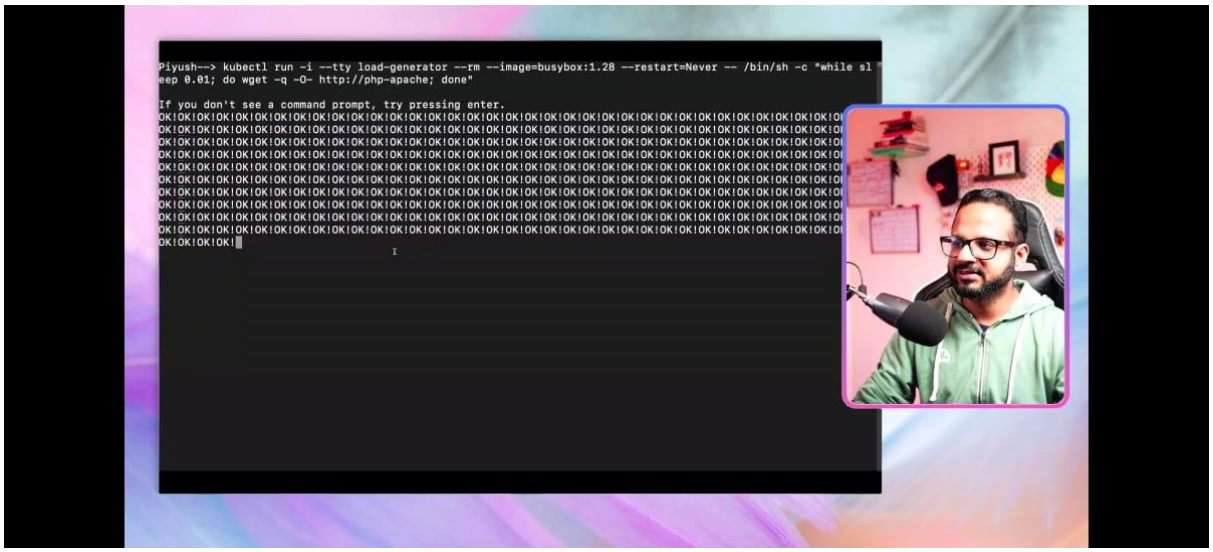
kubectl Autoscale deploy <deploymentfilename> --cpu-percent=50 --min=1 --max=10

* It works by when a pod reach 50 percent it autoscale the pod

* It checks the metrics 15seconds once for autoscale

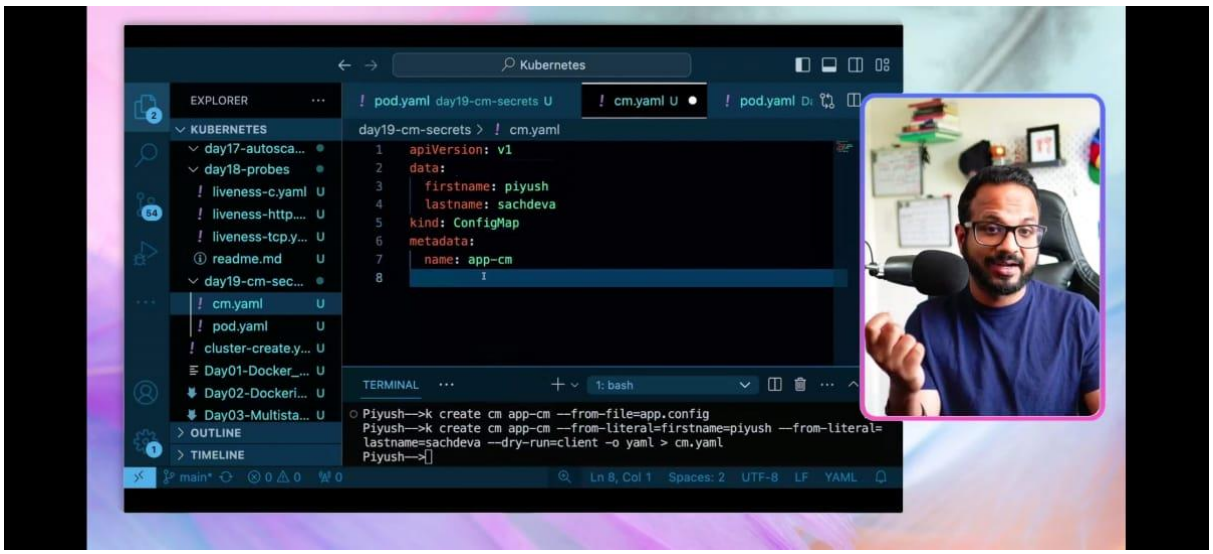
How to see HPA?

kubectl get hpa



Busybox image --> It is used to run some certain commands and doesn't have any os and it is lightweight

---



Rather than create configmap variable one by one, we can able to create a file and we save all the variables on that file and use

How to use a file on pod.yaml?

envFrom:

-- configMapRef:

name: <file-name>

```
apiVersion: v1
kind: Pod
metadata:
  name: startup-probe-example
spec:
  containers:
  - name: my-app
    image: my-app-image
    startupProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 10  # Wait 10s before first check
      periodSeconds: 5         # Check every 5s
      failureThreshold: 6      # Fail after 6 failed attempts (total 30s)
```

How It Works:

1. Kubernetes waits 10 seconds before the first probe.

2. It then checks the /healthz endpoint on port 8080 every 5 seconds.

3. If the probe fails 6 times in a row (total 30 seconds), Kubernetes restarts the container.

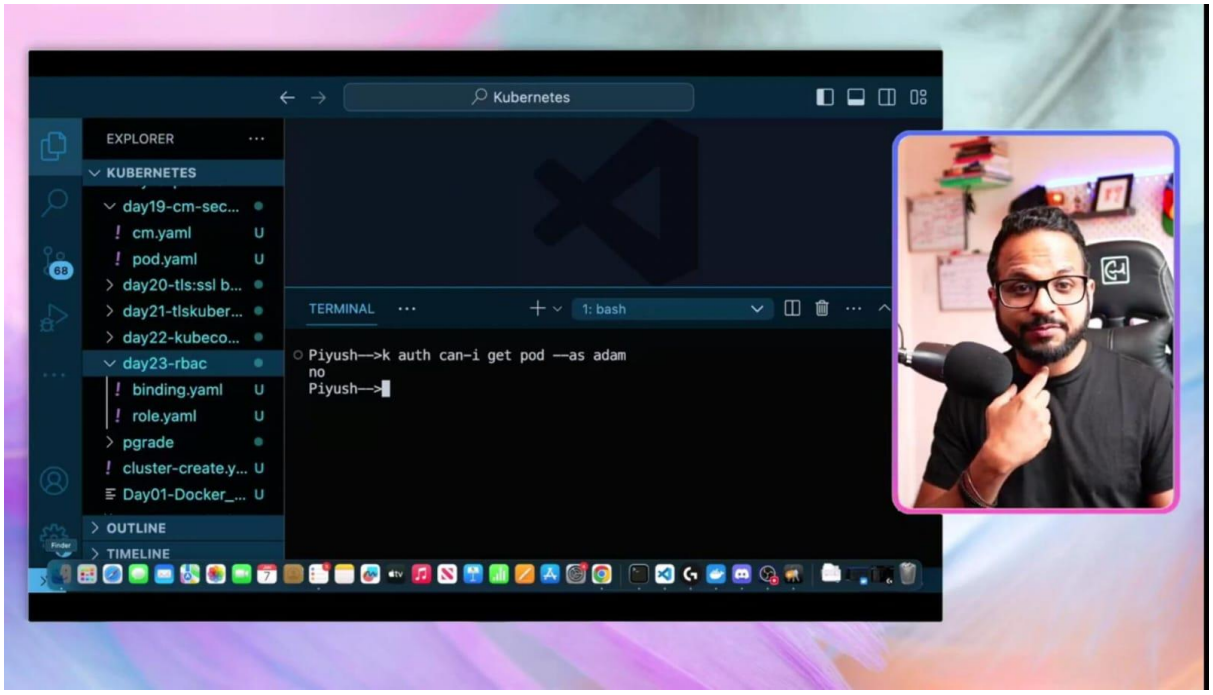
4. Once the Startup Probe succeeds, it stops running, and Liveness & Readiness Probes take over (if defined).

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-readiness-example
spec:
  containers:
  - name: my-app
    image: my-app-image
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 5  # Wait 5s before first check
      periodSeconds: 10       # Check every 10s
      failureThreshold: 3     # Restart if fails 3 times

    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 2  # Wait 2s before first check
      periodSeconds: 5        # Check every 5s
      failureThreshold: 3     # Remove from service if fails 3 times
```

Liveness Probe: Checks /healthz every 10s after an initial 5s delay. If it fails 3 times, Kubernetes restarts the container.

Readiness Probe: Checks /ready every 5s after an initial 2s delay. If it fails 3 times, the Pod is removed from the Service endpoints, but the container is not restarted.

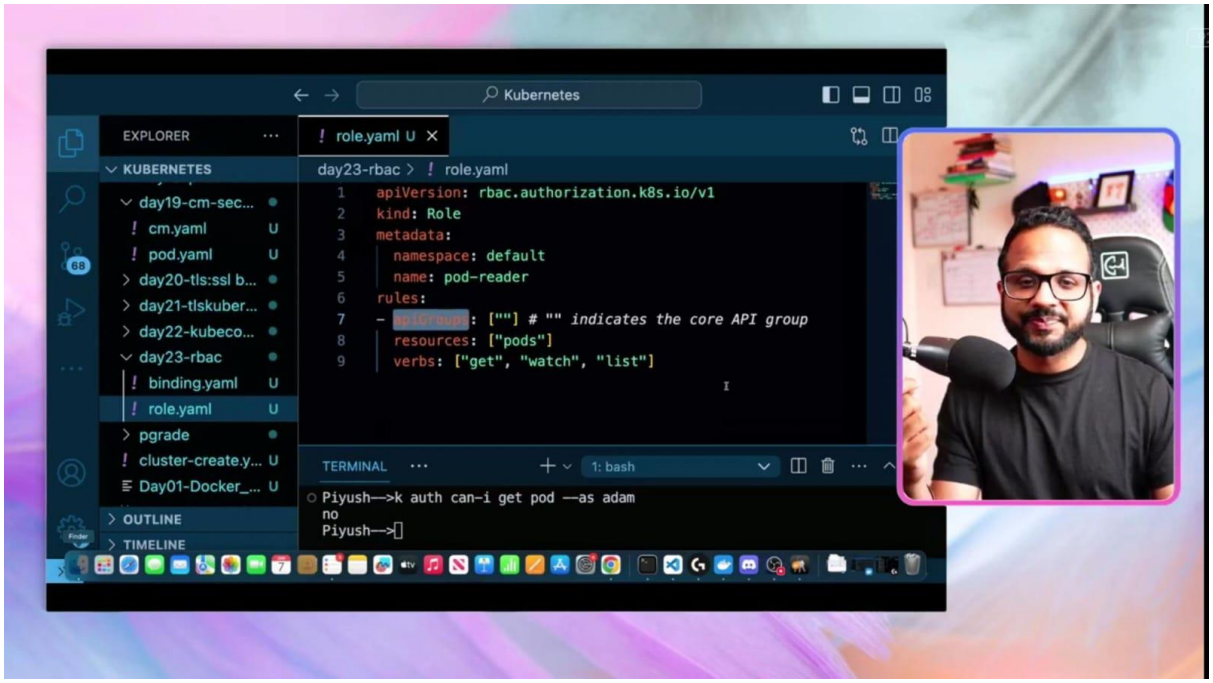


k auth whoami --> It shows what is my username

k auth can-i get pod --> It shows can i have access(for my user) to the pod

k auth can-i get pod --as <nithi> --> as a admin when i run this command, it shows the pod can be access by user(nithi) or not

---



in apiVersion : if there is only v1 that is core API or if there is an more than v1 that is name API.

apigroup:when we use (" "), it indicates that core API group

Verbs: it it used to give permission to that user are group

resources : which resource we have to give permission

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: Adam
  apiGroup: rbac.authorization.k8s.io
roleRef:
  ClusterRole
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

---

* It is the role binding to assign the role to the user adam

---

k get role --> to show the roles are in the k8

k describe role <role-name> --> it shows the detail about the role

k get roles -A --no-headers | wc -l --> it shows the total count of the roles are in the local machine in all name space

k get rolebinding --> it shows the role bindings

k describe rolebinding <rb-name> --> it shows the details about the rb

---

openssl req -new -key nithi.key -out nithi.csr -subj "/CN=nithi"

openssl genrsa -out nithi.key 2048

Scenario: A CI/CD Pipeline Deploying an Application in Kubernetes

Imagine you have a Jenkins pipeline running in a pod inside your Kubernetes cluster. This pipeline needs to:

1. Pull the latest application image from a Docker registry.

2. Deploy the application by creating/updating Kubernetes resources like Deployments, Services, and ConfigMaps.

How a Service Account Helps:

1. You create a Service Account (e.g., jenkins-sa).

2. Assign it the necessary permissions using RBAC (e.g., allowing it to create/update deployments).

3. Kubernetes automatically mounts an API token into the Jenkins pod.

4. Jenkins uses this API token to authenticate with the Kubernetes API and deploy the application.

Why the API Token is Important?

It ensures secure communication between Jenkins and the Kubernetes API.

Without the token, Jenkins wouldn't be able to create or update resources in the cluster.

Using a dedicated service account instead of the default one follows the principle of least privilege, reducing security risks.

This approach is widely used in DevOps workflows for automated deployments, monitoring, and managing Kubernetes resources securely.

---

kubectl auth can-i get pods --as=system:serviceaccount:<namespace>:<serviceaccountname> --> for check permission have assigned for sa

---

To assign a RBAC for group, before when we create a user there itself we have to define group to add the user into that group

How to do it?

openssl genrsa -out user1.key 2048

openssl req -new -key user1.key -out user1.csr -subj "/CN=user1/O=devops-team"

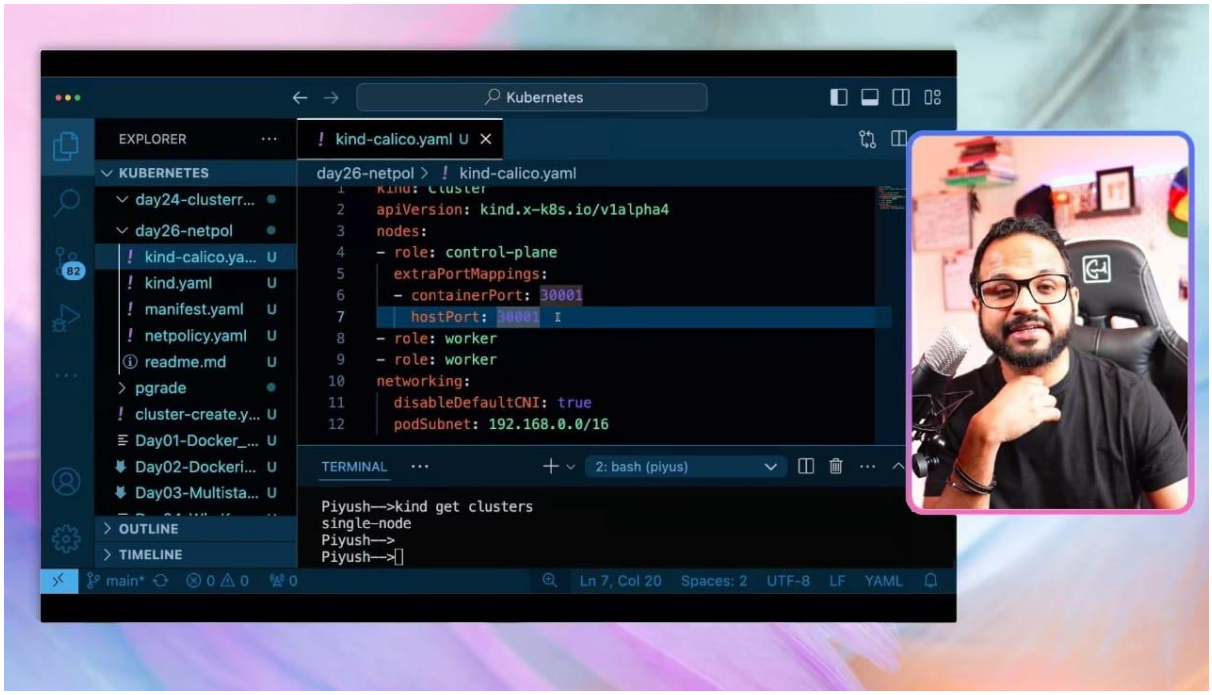
How to role binding for the group?

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-role-binding
  namespace: dev
subjects:
- kind: Group
  name: devops-team  # This must match the group name in the certificate (O=devops-team)
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-role
  apiGroup: rbac.authorization.k8s.io
```

Advantages for using group:

* We can give permission to the group instead of give permission to the users one by one if there is 15 users

---



This is how we can create cluster on calico network policies

---

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

  name: db-test

spec:

  podSelector:

    matchLabels:

      name: mysql

  policyTypes:

  - Ingress

  ingress:

  - from:

    - podSelector:

        matchLabels:

          role: backend

    ports:

- port: 3306

In MatchLabels(role: backend in ingress) --> we have to give, what pod will have to communicate to this pod(example: If we have backend pod that have to communicate with db pod, we have to give backend label on MatchLabel

MatchLabels(name:mysql) --> used to say what pod the backend want to communicate to

access: 'true' --> It allows to access the db only if the labels is matched

We can use db through using telnet

---

kubectl get pods -n kube-system -o wide -> to check CNI on local nodes

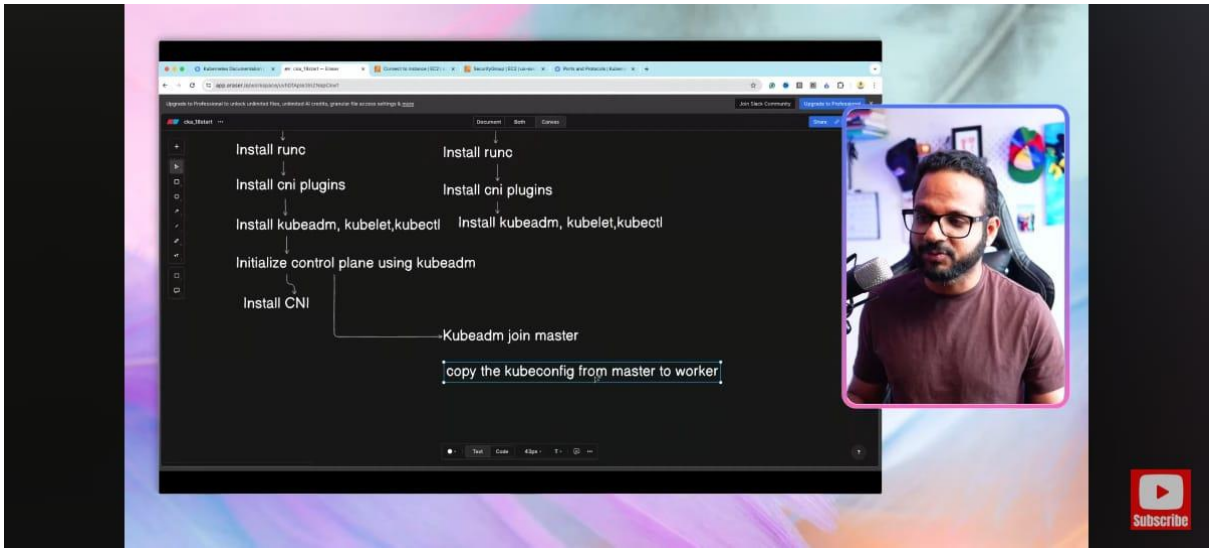
Purpose of kube-system

* Manages Kubernetes core components (API server, scheduler, etc.)

* Hosts networking plugins (CNI: Calico, Flannel, Weave, etc.)

* Runs monitoring and logging services (Metrics Server, CoreDNS)

* Contains system-level controllers (kube-proxy, etcd, controller-manager)



After setup the master and worker node we have to cope the kubeconfig file from master node and copy it on worker node on (cd $HOME/.kube/

- --kubelet-insecure-tls

Why is it needed?

When using kubectl top nodes or kubectl top pods, the metrics-server collects data from the Kubelet API running on each node. If the Kubelet API is using self-signed certificates or has a misconfigured TLS setup, metric queries can fail due to TLS verification errors.

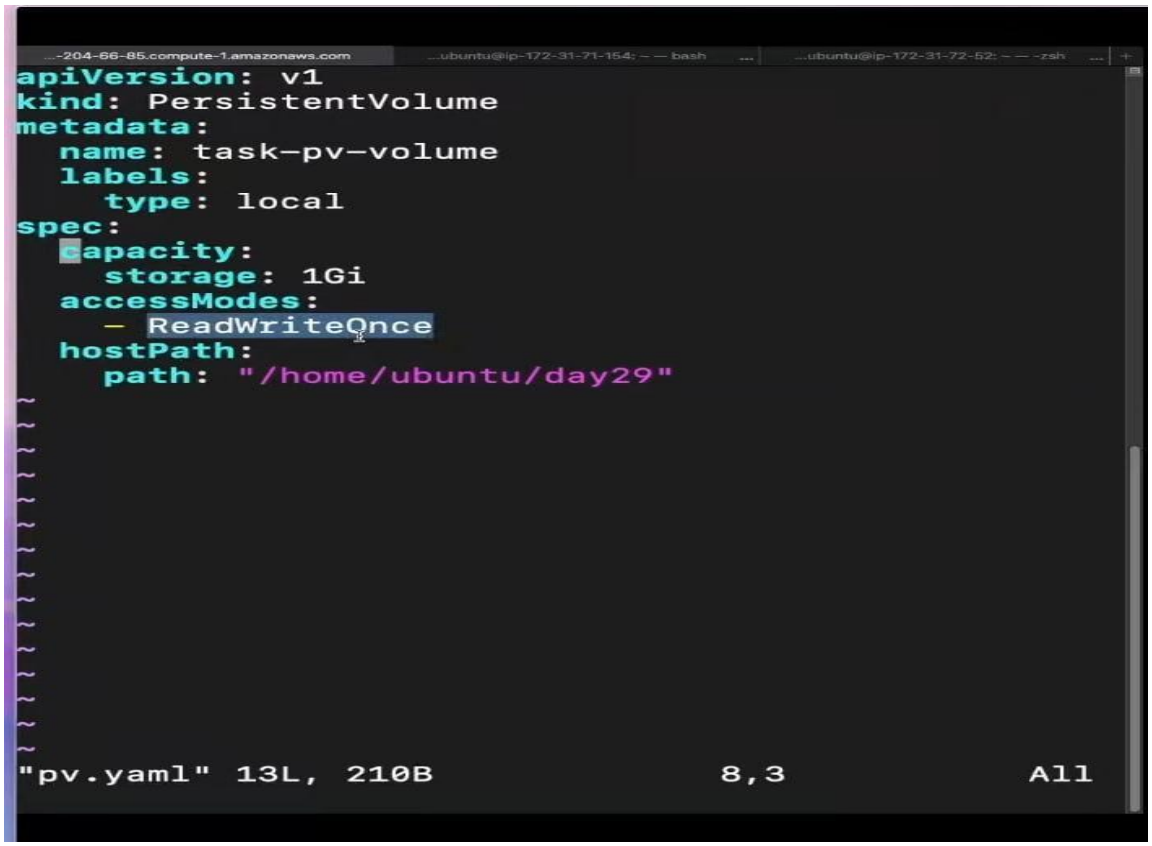
Adding --kubelet-insecure-tls forces the metrics-server to ignore certificate verification, allowing it to fetch metrics even when there are TLS issues.

kubectl top nodes --kubelet-insecure-tls

kubectl top pods --kubelet-insecure-tls

PV (Persistent Volume): A storage space in Kubernetes that keeps data safe even if a pod is deleted or restarted.

PVC (Persistent Volume Claim): A request made by a pod to use a specific amount of storage from the available PVs.



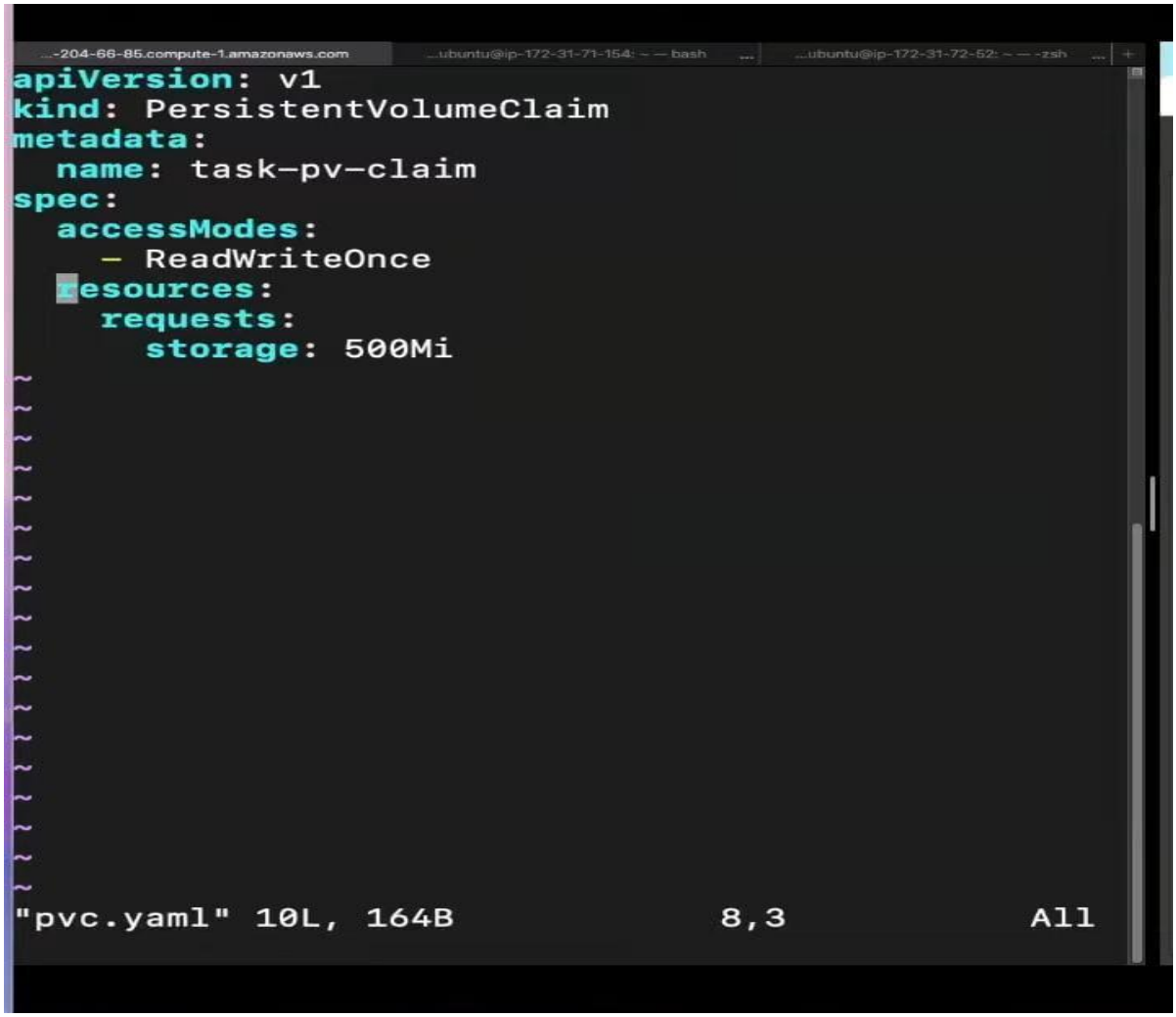
Storage --> It is assigned for the PV

accessmode --> assigned to the PV

hostpath --> It said where the volume will mount the data's in the PV

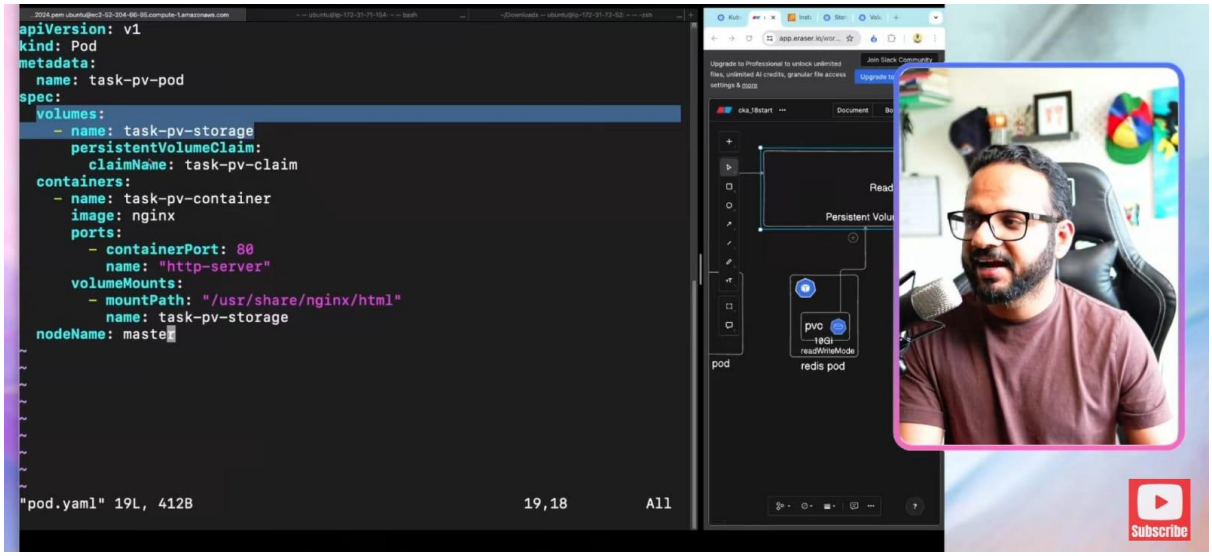
How to get PV?

--> kubectl get pv



It says how to create PVC and takes storage from PV for the pod without mention it

---



It is creating a pod with volume

Volumes --> It says which volume we going to use on this pod

Volumemount --> It says where this volume will mount on the container(pod) and the name is volume name claimname is PVC claim name

Retain: Keeps the storage and data even after the PVC is deleted. You have to delete or reuse it manually. (Best for important data like databases.)

Delete: Removes both the storage and data when the PVC is deleted. (Best for temporary data like logs.)

Recycle (Deprecated): Clears the data but keeps the storage for reuse. (Old method, now replaced by better options.)

---

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: azure-disk

provisioner: disk.csi.azure.com

parameters:

  storageaccounttype: Standard_LRS # Options: Standard_LRS, Premium_LRS, StandardSSD_LRS

  kind: Managed

reclaimPolicy: Delete

volumeBindingMode: WaitForFirstConsumer

allowVolumeExpansion: true

Used to create storage class in azure

provisioner: disk.csi.azure.com → Uses Azure Disk CSI driver.

storageaccounttype → Defines disk performance (Standard_LRS, Premium_LRS).

reclaimPolicy: Delete → Deletes storage when PVC is removed.

allowVolumeExpansion: true → Allows resizing the volume later.

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

---

used to assign a PVC for a storage class

```
apiVersion: v1
kind: Pod
metadata:
  name: azure-pod
spec:
  containers
      - name: my-container
      image: nginx
```

```
    volumeMounts:
      - mountPath: "/mnt/data"
        name: azure-storage
  volumes:
    - name: azure-storage
      persistentVolumeClaim:
        claimName: azure-pvc
```

---

used to assign a pod for storage class PVC

Why Didn’t It Get Scheduled on kind-worker Without nodeSelector?

Even though your pod had a toleration for gpu=true:NoSchedule, Kubernetes preferred kind-worker2. Here’s why:

1️⃣ The Kubernetes Scheduler Chooses the "Best" Node

By default, the scheduler tries to balance resource usage across all available nodes.

kind-worker2 had no taints, making it a "simpler" choice.

Even though kind-worker was eligible (due to the toleration), the scheduler still preferred kind-worker2.

2️⃣ Tolerations Do Not Force Scheduling

Tolerations only allow a pod to run on a tainted node, but they don’t force the pod onto that node.

The scheduler still prefers non-tainted nodes unless directed otherwise (e.g., via nodeSelector).

3️⃣ No Other Scheduling Constraints Were Present

If multiple nodes are eligible, Kubernetes randomly picks one based on internal scoring.

Since kind-worker2 was fully available, it was chosen over kind-worker.

---

kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"

Explaination:

kubectl run: Runs a Kubernetes pod.

-i --tty: Allocates a terminal so you can interact with the pod.

load-generator: Name of the pod.

--rm: Automatically deletes the pod once it exits.

--image=busybox:1.28: Uses the BusyBox image (lightweight Linux distro).

--restart=Never: Ensures this is a one-off pod, not a deployment.

-- /bin/sh -c "...": Executes a shell command.

Inside the shell command:

while sleep 0.01; do wget -q -O- http://php-apache; done: Continuously makes HTTP requests (every 0.01 seconds) to a service called php-apache.

- path: /

  pathType: Prefix

This will match all paths (like /, /foo, /bar/test) because every path starts with /.

It's commonly used to direct all traffic to a default backend (like a homepage, main app, etc.).

Route Multiple Services on the Same Domain Using Ingress

One of the biggest advantages of Ingress is that you can route multiple services using the same domain name by using path-based routing or subdomain-based routing.

🎯 Example: Path-Based Routing (Multiple Services Under One Domain)

Let’s say you have two services:

frontend-service (UI) → /

backend-service (API) → /api

You can define an Ingress like this:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  ingressClassName: nginx
  rules:
    - host: myapp.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: backend-service
                port:
                  number: 80
```

🔷 How It Works:

URL Routed to

http://myapp.local/ --> frontend-service

http://myapp.local/api --> backend-service

So, users accessing http://myapp.local/ will get the frontend, and API requests to http://myapp.local/api will be sent to the backend.

Errors

1. Error Validating Data

→ The configuration (YAML/JSON) you provided has a mistake. Kubernetes can't understand or accept it.

2. Scheduling Disabled

→ The cluster/node isn’t allowed to schedule new pods, maybe due to maintenance or it's been marked unschedulable.

3. ImagePullBackOff

→ Kubernetes is trying to pull the container image, but it's failing repeatedly. Possible reasons:

Wrong image name

No access (e.g., private repo)

Image doesn't exist

4. Create Container Error

→ Kubernetes couldn’t create the container. Could be due to wrong configs, missing image, or permission issues.

5. Exceeds CPU Limits

→ Your pod is trying to use more CPU than allowed (as per resource limits you set in the config).

6. Kubelet Crash

→ The kubelet (the agent on each node) crashed or failed. This can impact pod scheduling and health checks.

7. Failed Mount

→ Kubernetes couldn't mount a volume (like a PVC or host path) to the pod. Might be due to incorrect path, missing volume, or permissions.

8. Failed Scheduling

→ Kubernetes can't find any suitable node for your pod. Reasons could be:

Not enough CPU/RAM

NodeSelector/Tolerations/affinity rules mismatch

9. Run Container Error

→ Container couldn’t start. Could be due to command errors, bad entrypoint, or missing files.

10. Probe Failing

→ Liveness or readiness checks are failing. Kubernetes thinks the app inside the pod is unhealthy or not ready.

11. CrashLoopBackOff

→ Your container keeps crashing and Kubernetes is waiting before trying to restart it again. Common reasons:

App error

Missing config/env variables

Port conflicts

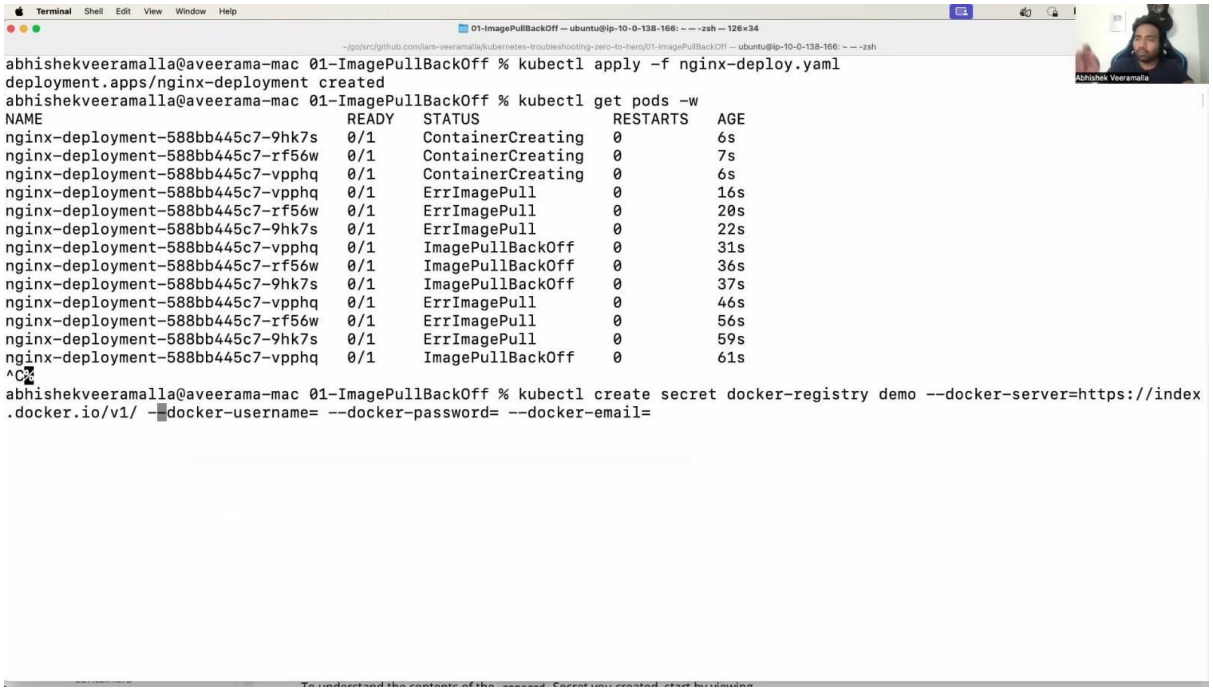
What Does initialDelaySeconds Do in Probes?

The initialDelaySeconds parameter in Kubernetes liveness and readiness probes determines how long Kubernetes waits before starting to check the health of a container after it starts.

📌 Why It Matters?

If your application takes some time to fully start, the probe might fail too early, causing Kubernetes to restart the pod unnecessarily.

Increasing initialDelaySeconds gives your application more time to initialize before Kubernetes starts checking its health.



docker registry --> It is a secret store for docker and it can be changed based on app(docker, ECR)

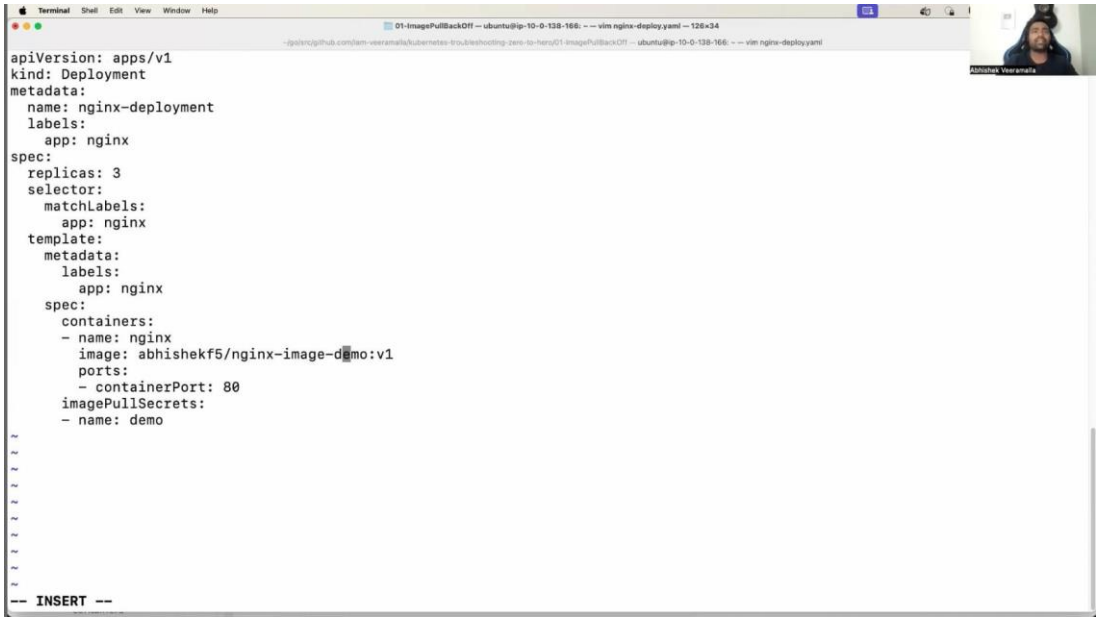
demo --> Secret name

docker server "https://index.docker.io/v1/" --> It is default for every docker server

docker username --> Dockerhub userme

docker password --> Dockerhub password

docker email --> dockerhub email



This is how we can give yaml for create pod which is in Private docker registry

image : <dockerhub username>/<image name>

image pull secrets is used to say the file name of the secrets of docker credientials on local machine

---

When we create a stateful set with volume and delete the stateful set and then create the same stateful set, it will cant create because when we delete stateful set we have to delete volume too, without we delete volume manually it cant create a stateful set when we try to re apply that yaml file... But we can create with another volume name

---

crictl in Kubernetes

crictl is a CLI tool used to interact with the container runtime interface (CRI) on Kubernetes nodes. It’s mainly used for troubleshooting at the node level.

Common Commands:

Check containers:

crictl ps -a

View pods:

crictl pods

Pull an image:

crictl pull nginx

View logs:

crictl logs <container_id>

Why use crictl?

It helps debug when kubectl can’t connect (e.g., kubelet or API server issues).

Interacts directly with the container runtime (containerd, CRI-O).

Is we able to connect to another pod in another namespace if we are in different namespace?

Yes, you can connect to a pod in another namespace in Kubernetes, even if you're in a different namespace, as long as network policies or firewall rules don’t block it.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
```

```
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

This NetworkPolicy:

Applies to pods labeled role: db.

Allows ingress:

From IPs in 172.17.0.0/16 (excluding 172.17.1.0/24),

From pods labeled role: frontend in namespaces labeled project: myproject,

Only on port 6379.

Allows egress:

To IPs in 10.0.0.0/24,

Only on port 5978.

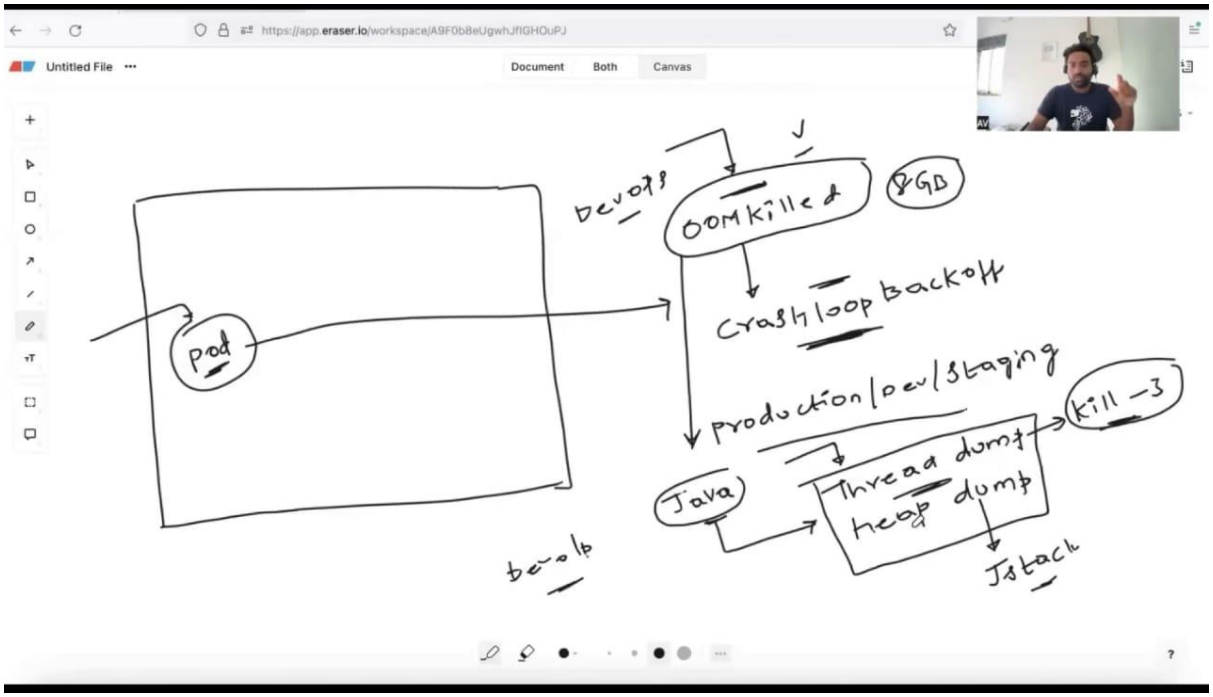
Ingress: Incoming traffic to the selected pods.

Egress: Outgoing traffic from the selected pods

REWATCH THE VIDEO 11, 27, 34, 35, 38

Reference

https://kubernetes.io/docs/tasks/debug/debug-cluster/



There are three real time challenging scenerios in real time production environment

* Resource usage --> There is a single cluster in production and we have to deploy 4-5 applications at a single cluster so we divided by namespace. But when a pod get leaked memory it containes more RAM and memory so other pods will get crashed. To solve this, we set two things

1) Resource Quota --> It will set limit for namespace level

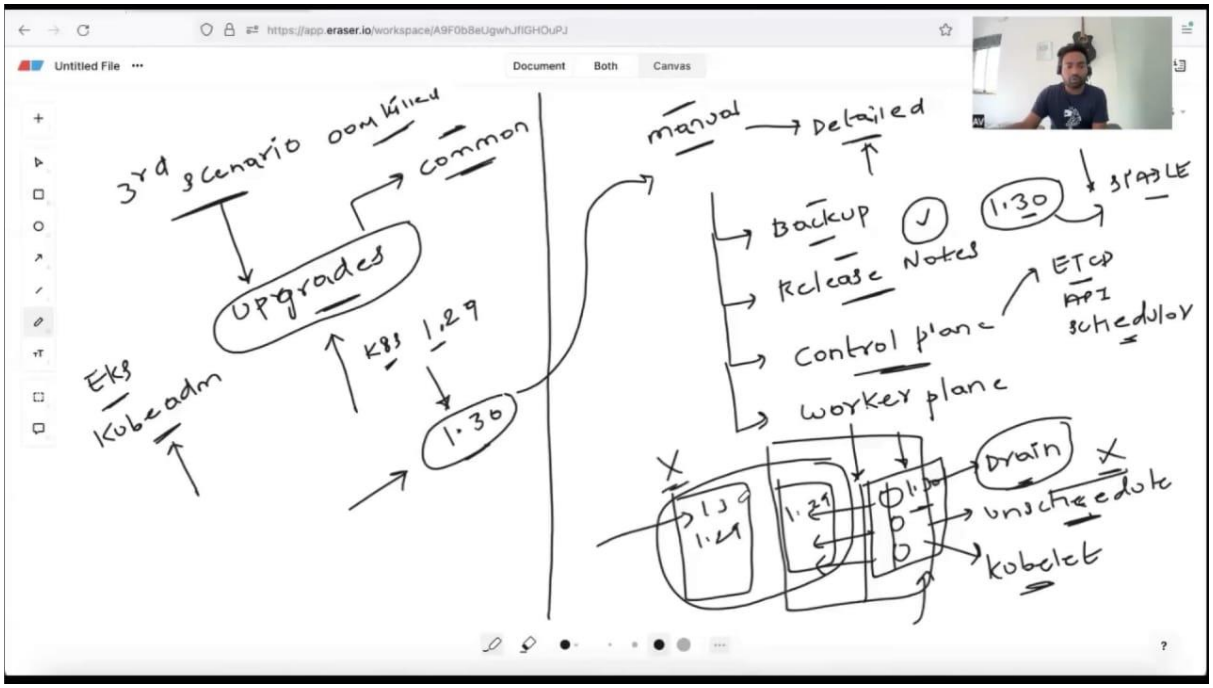
2) Resource limit --> It will set limit for pod level

One solved, But what if pod get leaked memory and It full and get OOMkilled error,

To solve this as a devops engineer,

We will do Thread dump and heap dump to that pod inside that pod and we will save the output on file and send it to the developers to find the errors and resolve it

---



Another challenging is upgrade the nodes

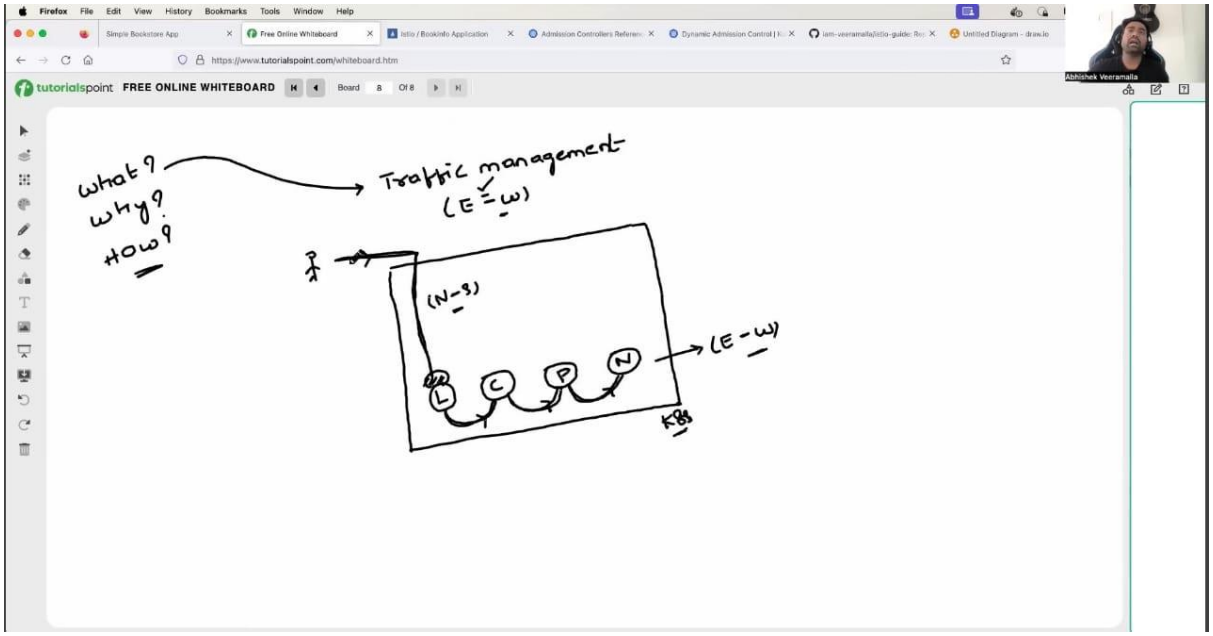
Update the nodes to the newer version is some more complicated in production level. To solve this, we have to put step by step manual to update the nodes

Step 1 - Backup the Datas

Step 2 - Read the Release notes of the newer version to save node from the node down

Step 3 - Update the control plane one by one(api, etcd, scheduler)

Step 4 - Update the worker node one by one



Istio

Istio is used to manage the traffic between the pods(microservices)

Istio manages the East-west traffic

- East west traffic is a traffic between Pods inside the cluster

- North south traffic is a traffic outside from the users to the pod(load balancer)

Advantages of Istio

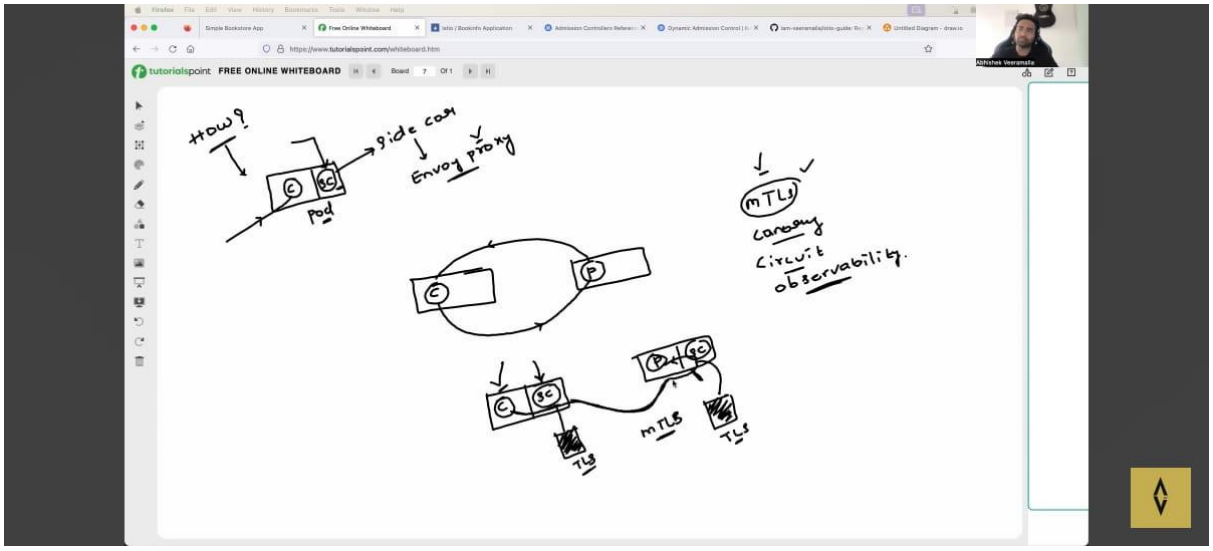
* MTLS (manual TLS) --> makes a secure connection between the pods by ensuring the TLS certificate

* Canary Load balancing

What is canary?

- When a pod is in update state, we will manually set the load balancing by 10% request will send to new pod(updating pod) and 90% will send to old pod(existing pod)

* Kiali --> Kiali is a observability to manage the services between the pods and it supports by Istio and it have UI to easy access.



How Istio works?

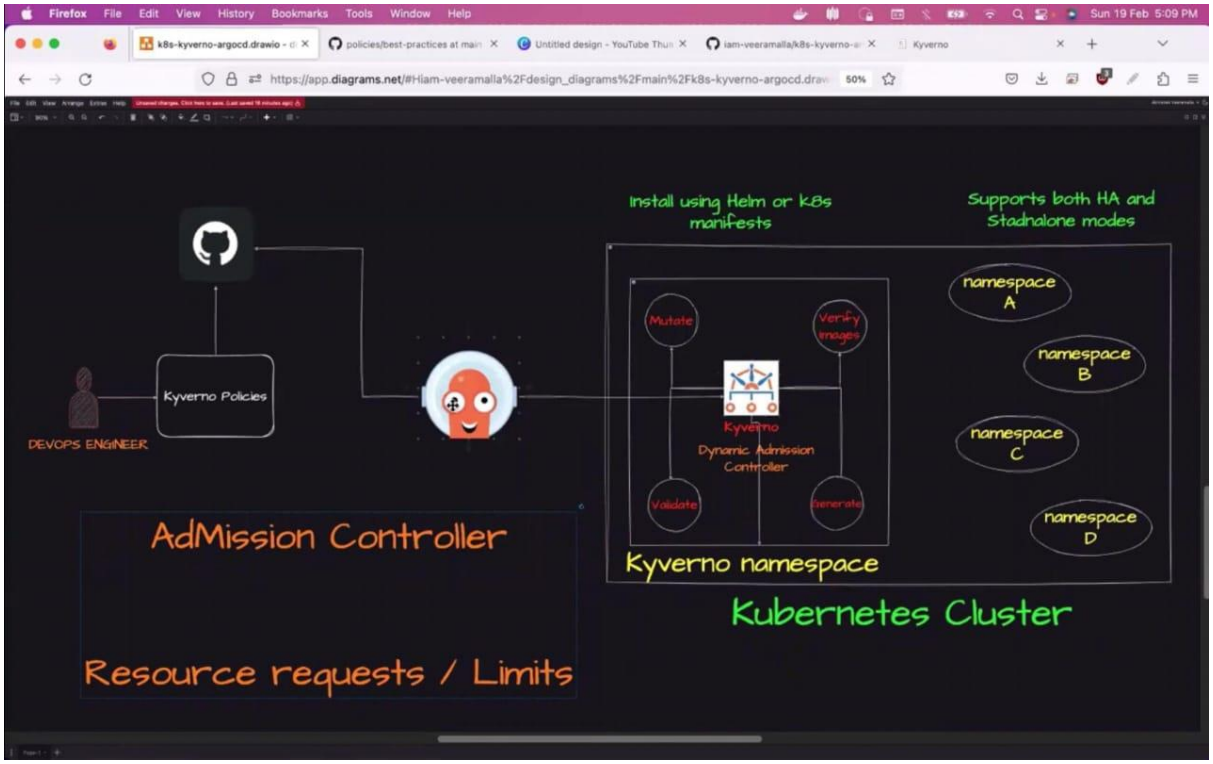
When there is an pod is already running, how to set istio?

* When there is an existing pod, Istio will set the sidecar pod(extra pod or helper pod) to the existing pod.

* When the request will come to the pod, the request will verified by sidecar pod by verifing TLS certificates before the request will reach the pod and it can applicable for all pods inside the cluster

* The requests are verified by sidecar pod before it reach the main pod

---



Kyverno

Kyverno is the dynamic Admission controller tool which helps to write admission controller easy by devops engineers

What is admission controller?

Admission controller is the rules that have given by organization to maintain their organization standardly

(Ex: If a pod want to take upto 2gb of ram, we will write the admission control to prohibit to create the pod above 2gb)

* Writing admission controller will be more complicated before kyverno but kyverno will made it easy by creating admission controller using yaml file

Must⚠(Change on clusterpolicy file)

--Validatefailureaction=enforce

Readiness probe

httpGet → Ping http://<pod-ip>:8080/health to check readiness.

initialDelaySeconds: 0 → Start probing immediately after container starts.

timeoutSeconds: 1 → Fail the probe if no response in 1 second.

periodSeconds: 10 → Probe check every 10 seconds.

successThreshold: 1 → 1 success = Pod is marked "Ready".

failureThreshold: 3 → 3 consecutive failures = Pod marked "Not Ready" (no traffic sent).

Liveness Probe

httpGet → Ping http://<pod-ip>:8080/health to check if the app is alive.

initialDelaySeconds: 60 → Start probing after 60 seconds (give app time to boot)

timeoutSeconds: 5 → Fail the probe if no response in 5 seconds.

periodSeconds: 10 → Probe check every 10 seconds.

successThreshold: 1 → 1 success = considered healthy.

failureThreshold: 5 → 5 consecutive failures = Kubernetes restarts the pod.

---

port: 5000 : Service’s internal port (within cluster)

targetPort: 5000 : Container’s port (where app runs)

nodePort: 30002 : Port on Node/VM to access from outside

---

kubectl rollout restart deployment myapp(image name)

When to Use It:

When your image tag hasn't changed (e.g., still using latest), but you want Kubernetes to pull and restart the pods anyway.

After updating the image in DockerHub or ACR with the same tag.
