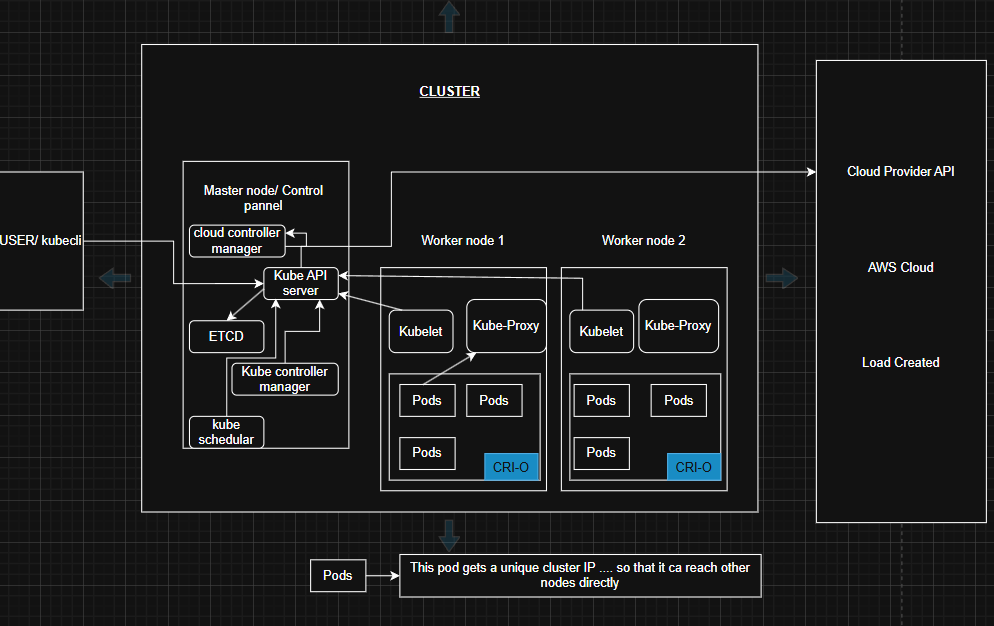
**Task – 1**

1. draw k8s architecture

**🔹 Workflow of a Kubernetes Cluster**

1. **User / kubectl**

\* The user interacts with the cluster using kubectl (command-line tool).

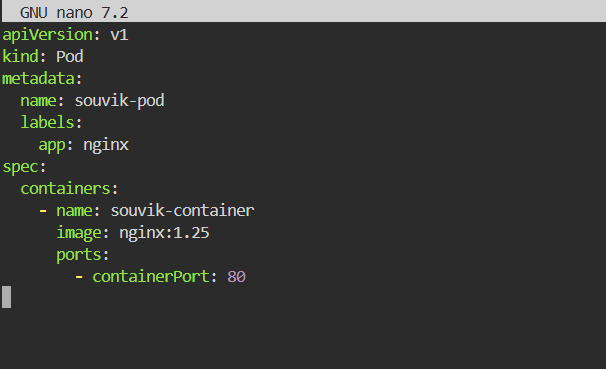
\* Commands like kubectl apply -f deployment.yaml are sent to the **Kubernetes API Server**.

1. **Control Plane (Master Node)**  
   This is the **brain** of the cluster, making global decisions.  
   Components inside it:
   * **Kube-API Server**
     + Entry point for all commands (from user or other components).
     + Validates requests and updates cluster state in etcd.
     + Exposes the Kubernetes API (REST-based).
   * **etcd**
     + A distributed **key-value store**.
     + Stores all cluster data (pods, services, secrets, configmaps)
   * **kube-scheduler**
     + Decides **which worker node** a pod should run on.
     + Considers resource availability (CPU, memory), taints, tolerations, node affinity
   * **kube-controller-manager**
     + Runs multiple controllers to ensure desired state = actual state. Examples:
       - **Node controller**: Manages node status (healthy/unhealthy).
       - **Replication controller**: Ensures the right number of pod replicas are running.
       - **Endpoint controller**: Updates service endpoints.
       - **Job controller**: Manages job execution.
   * **Cloud-controller-manager**
     + Integrates Kubernetes with cloud providers (AWS, GCP, Azure).
     + Manages load balancers, volumes, and node lifecycle when running in cloud.
2. **Worker Nodes (Node 1, Node 2 and so on)**  
   These are the **machines where applications (pods) run**.  
   Each worker node has:
   * **Kubelet**
     + Agent that talks to the API server.
     + Ensures containers described in PodSpecs are actually running.
     + Reports node and pod health back to the control plane.
   * **Kube-Proxy**
     + Handles networking and load balancing inside the cluster.
     + Ensures each pod can talk to another pod across nodes using **cluster IPs**.
     + Manages iptables or IPVS rules for service routing.
   * **CRI-O (or other Container Runtime like containerd/Docker)**
     + The actual container runtime that pulls images and runs containers.
     + Kubernetes doesn’t run containers directly rather it delegates to CRI (Container Runtime Interface).
   * **Pods**
     + The **smallest deployable unit** in Kubernetes.
     + A pod can run one or more tightly coupled containers.
     + Each pod gets a **unique cluster IP** so that it can communicate with other pods across nodes.
3. **Networking**
   * Each pod gets a unique IP in the cluster.
   * **Kube-Proxy** makes sure pods can communicate across worker nodes.
   * Services (ClusterIP, NodePort, LoadBalancer) expose pods reliably.
4. **Cloud Provider API (e.g., AWS, GCP, Azure)**
   * If you’re running on the cloud, Kubernetes uses the **Cloud Controller Manager** to interact with the cloud.
   * Example:
     + Creating a LoadBalancer on AWS when you expose a Service of type LoadBalancer.
     + Attaching storage volumes dynamically.

**🔹 Putting the Workflow Together**

1. **User runs a command** with kubectl (e.g., deploy an app).
2. **API Server** receives the request and stores the desired state in etcd.
3. **Scheduler** decides which node will run the pod.
4. **Controller Manager** ensures replicas, endpoints, and other states are maintained.
5. **Kubelet on the worker node** receives instructions from the API Server, then uses CRI-O (container runtime) to start the containers.
6. **Kube-Proxy** sets up networking so that pods can talk to each other.
7. **Cloud Controller Manager** provisions cloud resources (like load balancers or storage) if needed.
8. **Pods start running** with unique cluster IPs and are accessible according to the service definitions.

2. write pod yaml file, create pod and define each n every instruction of pod yaml



**1. apiVersion**

apiVersion: v1

* Tells Kubernetes which API version to use.
* Pods use v1.

**2. kind**

kind: Pod

* Defines the type of resource we are creating.
* Other values can be Deployment, Service, etc.

**3. metadata**

metadata:

name: souvik-pod

labels:

app: nginx

* Metadata about the Pod.
* **name**: Unique identifier for the Pod inside the namespace. It should be unique for every pod
* **labels**: Key-value pairs to identify and group resources (used by selectors, services, deployments).

**4. spec**

spec:

containers:

- name: souvik-container

image: nginx:1.25

ports:

- containerPort: 80

* **spec** = Desired state of the Pod.
* Defines the containers inside the pod.

Inside the containers:

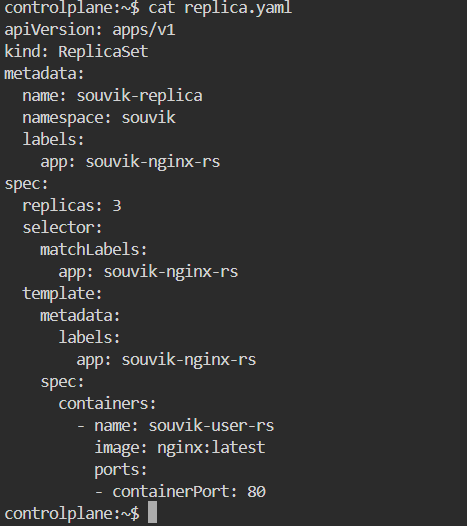
* **name** → A unique name for the container inside the pod.
* **image** → Which container image to run (from DockerHub, private registry, etc.).
* **ports** → Exposes container’s port (here, 80 for Nginx).

3. prepare all commands of pod in k8s

Commands like

1. kubectl apply -f pod.yml = This command creats and runs the (POD) yml file that we have just created inside the shell.
2. kubectl get pods = this command will give us all the pods that are present in the cluster. And check the status
3. kubectl describe pod nginx-pod = Pulls detailed info from the API Server about the pod, like its node, events, and container status. Here nginx-pod is the pod name that we have given inside the yml file and we can also see that using the above command kubectl get pods.
4. kubectl logs nginx-pod = Asks the Kubelet via the API Server to fetch logs from the Nginx container.
5. kubectl exec -it nginx-pod -n souvik/bin/bash = Opens an interactive shell inside the Nginx container via the Kubelet.
6. kubectl delete -f pod.yml = deletes the pod.yml file.

4. write replicaset yaml file, create rs and define each n every instruction of rs yaml



**1. apiVersion: apps/v1**

* Tells Kubernetes which version of the API we’ll be using.
* For ReplicaSets, the version is apps/v1.

**2. kind: ReplicaSet**

* Defines the Kubernetes resource type → Here, it’s a **ReplicaSet**.
* A ReplicaSet ensures that a **specified number of pod replicas** are always running.

**3. metadata**

metadata:

name: souvik-replica

namespace: souvik

labels:

app: souvik-nginx-rs

* **name: souvik-replica** → The name of this ReplicaSet.
* **namespace: souvik** → Runs inside the souvik namespace
* **labels: app: souvik-nginx-rs** → Metadata label to identify this ReplicaSet.

**4. spec**

spec:

replicas: 3

* Defines the **desired state**.
* **replicas: 3** → Ensures there are always **3 Pods** running.

**5. selector**

selector:

matchLabels:

app: souvik-nginx-rs

* Tells the ReplicaSet which pods to manage.
* It looks for pods with the label app: souvik-nginx-rs.
* Must match the labels in the pod template that we’ll be defining next

**6. template (Pod Template)**

template:

metadata:

labels:

app: souvik-nginx-rs

spec:

containers:

- name: souvik-user-rs

image: nginx:latest

ports:

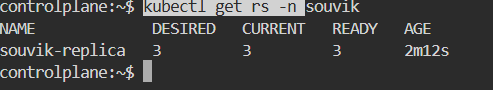
- containerPort: 80

* This defines the blueprint for the pods that ReplicaSet will create.
* **metadata → labels**: app: souvik-nginx-rs → This matches the selector above.
* **spec → containers**:
  + Runs a container named **souvik-user-rs**.
  + Uses the **nginx:latest** image.
  + Exposes port **80** (Nginx default HTTP port).

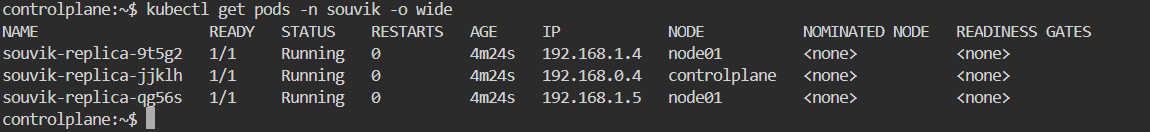
5. prepare all commands of rs in k8s



1. A ReplicaSet named **souvik-replica** is created inside the souvik namespace.
2. It will create **3 pods** with the label app=souvik-nginx-rs.
3. Each pod will run one container (nginx:latest) listening on port 80.
4. If one pod crashes or is deleted, the ReplicaSet will immediately create a newone to maintain 3 replicas.

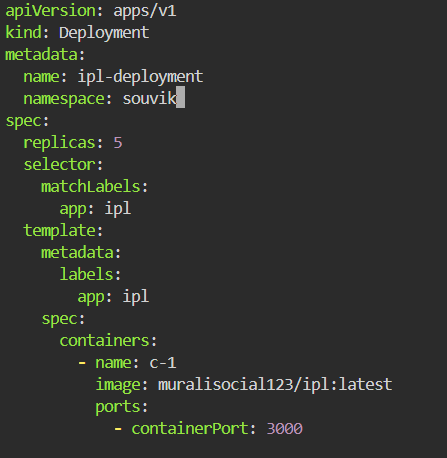


Kubectl get rs -n souvik = this will; show the details of the the replica set with desired state, current set and the status and the age as well in namespace souvik.



Kubectl get pods -n souvik -o wide = this command we use to know the whole details of the pod in namespace souvik.

6. write deployment yaml , create it and 1define each n every instruction of deployment yaml



**1. apiVersion: apps/v1**

* Defines which Kubernetes API version this resource uses.
* apps/v1 is the current stable API for Deployments.

**2. kind: Deployment**

* Specifies the **type of resource** → this is a **Deployment**.
* A Deployment manages ReplicaSets and ensures the desired state (e.g., number of pods, container version) is always maintained.

**3. metadata**

metadata:

name: ipl-deployment

namespace: souvik

* **name: ipl-deployment** = Name of the Deployment.
* **namespace: souvik** = This Deployment is created inside the namespace souvik-ns.
  + If we do not specify a namespace, it assigns it to default.

**4. spec**

spec:

replicas: 5

* Defines the desired state of the Deployment.
* replicas: 5 → The Deployment ensures that **5 pods** are always running.

**5. selector**

selector:

matchLabels:

app: ipl

* The Deployment uses this selector to **identify which Pods to manage**.
* It looks for Pods that have the label app: ipl.

The labels in selector.matchLabels must match the labels defined in the Pod template that we’ll be writing below, otherwise Deployment won’t manage the Pods.

**6. template (Pod Template)**

This defines the **blueprint for Pods** that the Deployment will create.

template:

metadata:

labels:

app: ipl

* Each Pod created will have the label app: ipl.
* This matches the Deployment selector → ensuring that the Deployment manages these Pods.

**7. Pod Spec**

spec:

containers:

- name: c-1

image: muralisocial123/ipl:latest

ports:

- containerPort: 3000

* Defines **what containers run inside the Pod**.
* **containers** → A list of containers .

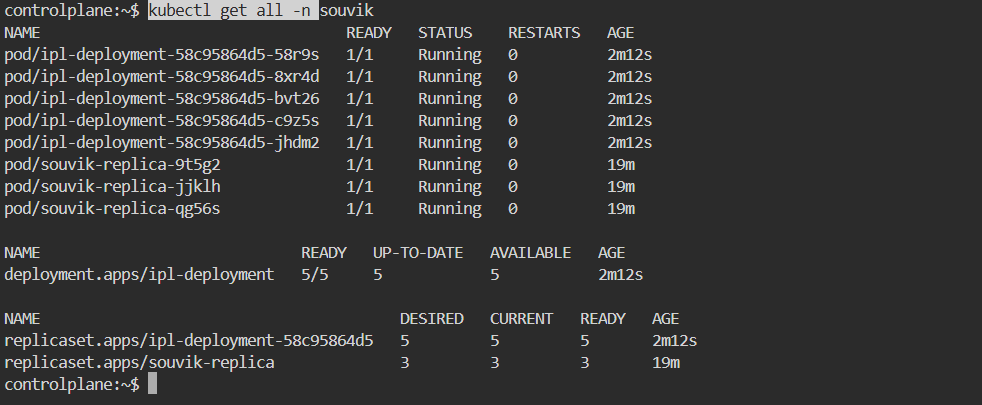
**Inside container spec:**

* **name: c-1** → The name of the container (can be anything, just an identifier).
* **image: muralisocial123/ipl:latest** → Docker image to run.
  + Here it’s pulled from Docker Hub user muralisocial123, repository ipl, tag latest.
* **ports:**
  + **containerPort: 3000** → Exposes port 3000 inside the container.
  + This is where your app (likely a Node.js app) is listening.

7. prepare all commands of deployment in k8s



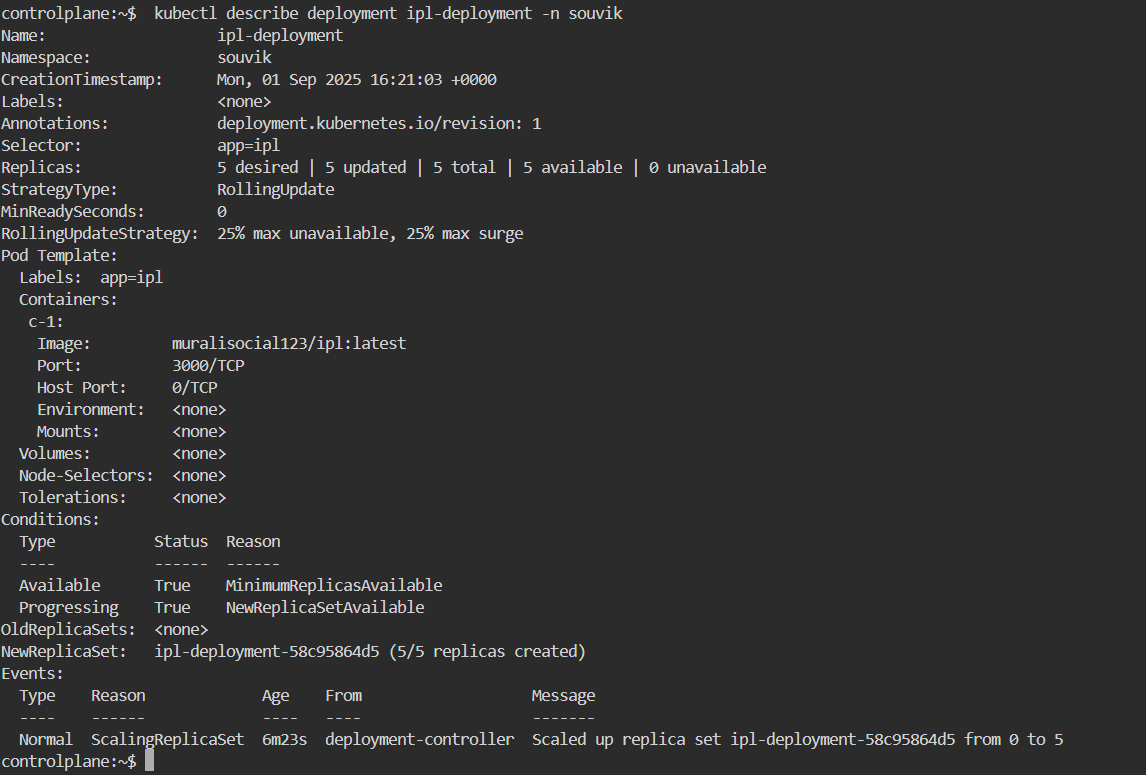
Kubectl apply -f deployment.yaml = A **Deployment** object named ipl-deployment is created in namespace souvik-ns.



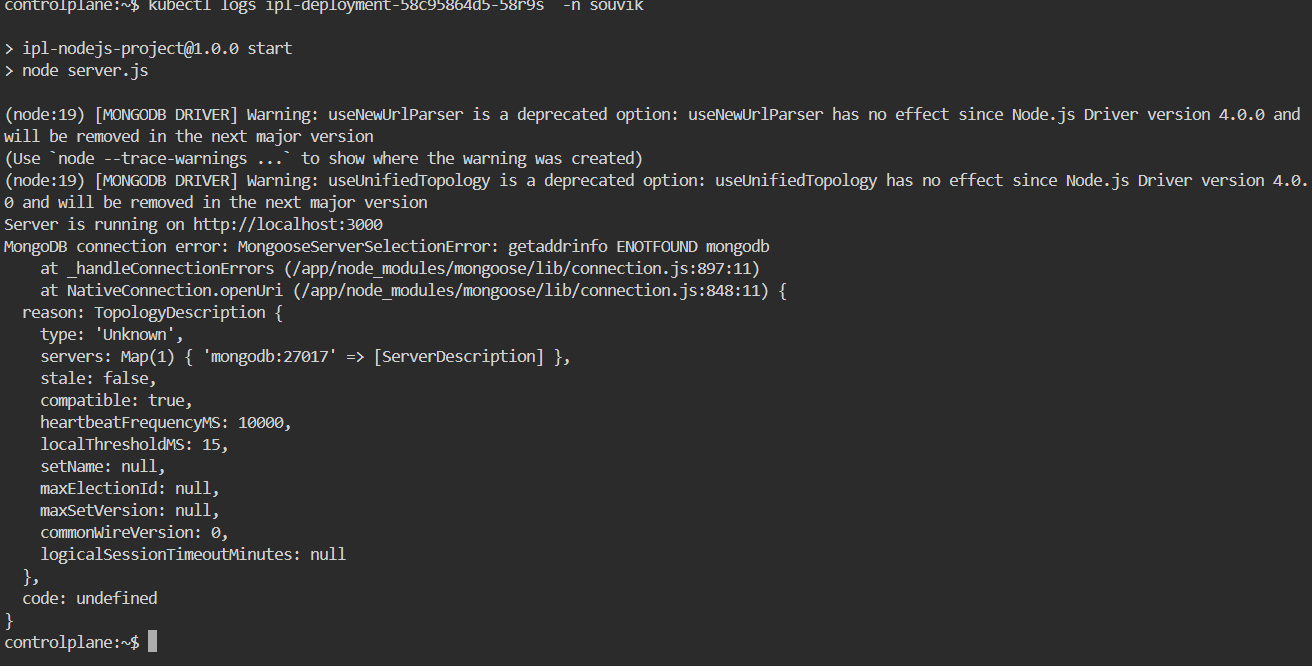
Kubectl get all -n souvik = All running instances of your containers in the namespace souvik.



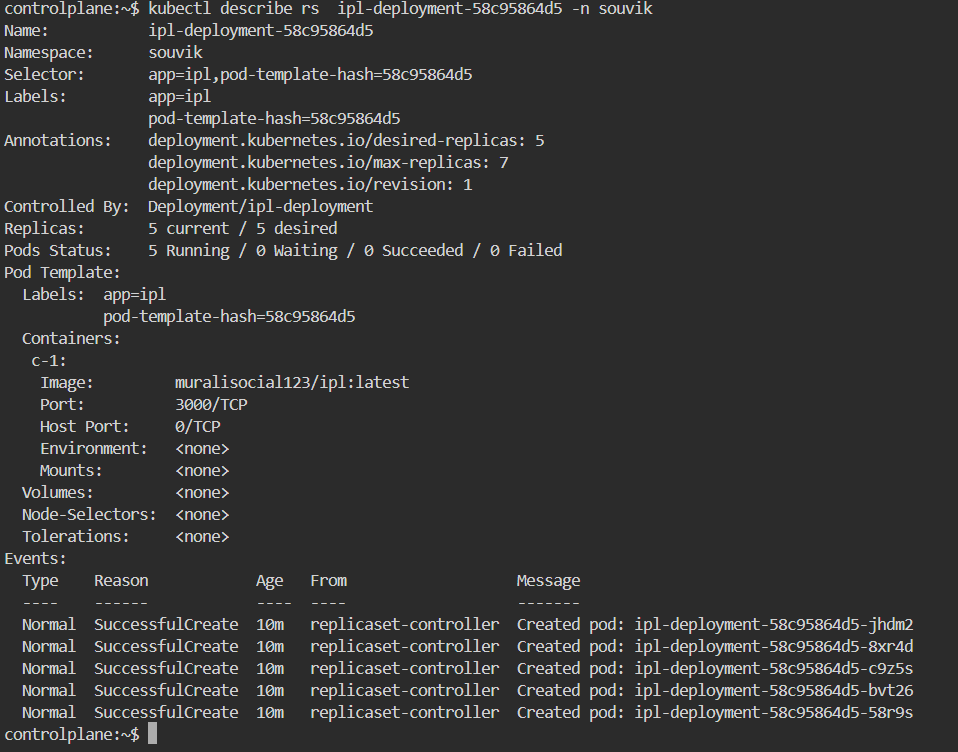
kubectl rollout undo deployment ipl-deployment -n souvik = It **rolls back** the Deployment named **ipl-deployment** in the **souvik** namespace. Since theres nothing to rollback so it shows no rollout history found.



kubectl describe deployment ipl-deployment -n souvik = It shows **detailed information** about the ipl-deployment in the namespace souvik.  
This is much more detailed than kubectl get deployments.



kubectl logs ipl-deployment-58c95864d5-58r9s -n souvik = gives the logs or the actions performed in that pod inside the namespace souvik



kubectl describe rs ipl-deployment-58c95864d5 -n souvik = It describes a **ReplicaSet** created by my Deployment (ipl-deployment) in the namespace souvik.



kubectl describe rs ipl-deployment-58c95864d5-58r9s -n souvik = to get inside the particular pod inside namespace souvik.

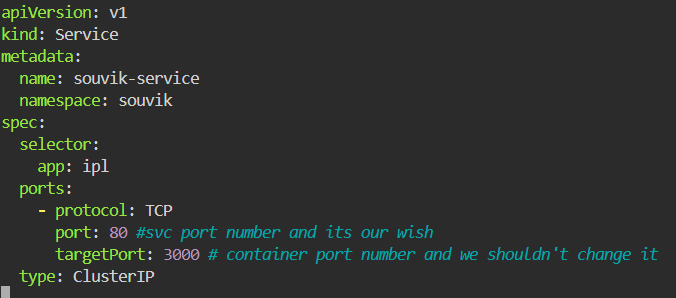


kubectl delete pod ipl-deployment-58c95864d5-58r9s -n souvik = deletes the pod name ipl-deployment-58c95864d5-58r9s inside namespace souvik.



kubectl delete rs ipl-deployment-58c95864d5 -n souvik = deletes the replicaset ipl-deployment-58c95864d5 inside the namespace souvik.

8. write service yaml file for all 4 types and define each n every instruction of svc yaml



**1. apiVersion: v1**

* Specifies the Kubernetes API version.
* Since a Service is a **core resource**, its version is v1.

**2. kind: Service**

* Defines the type of Kubernetes object.
* Here, it is a **Service**, which exposes Pods to other Pods (or externally, depending on type).

**3. metadata**

Holds identifying information about the Service.

* name: souvik-service → The unique name of the Service inside the namespace.
* namespace: souvik → Places this Service inside the souvik namespace.

**4. spec**

This section defines the **desired state** of the Service.

**4.1. selector**

selector:

app: ipl

* This links the Service to Pods that have the label app=ipl.
* Any Pod created by your Deployment/ReplicaSet with this label will be targeted.
* Without this, the Service won’t know which Pods to route traffic to.

**4.2. ports**

ports:

- protocol: TCP

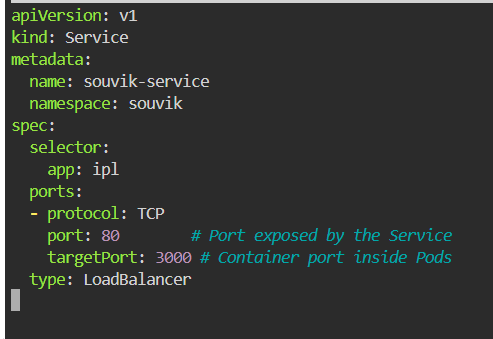
port: 80

targetPort: 3000

* **protocol: TCP** → Communication protocol. Default is TCP, but you can use UDP if needed.
* **port: 80** → The Service’s port (clients inside the cluster connect to this).
  + Example: Another Pod can access this Service at souvik-service:80.
* **targetPort: 3000** → The container port where traffic should be forwarded inside Pods.
  + This must match the container’s containerPort defined in our Pod/Deployment YAML.
* Essentially → Service (80) ➝ forwards traffic ➝ Pod (3000).

**4.3. type: ClusterIP**

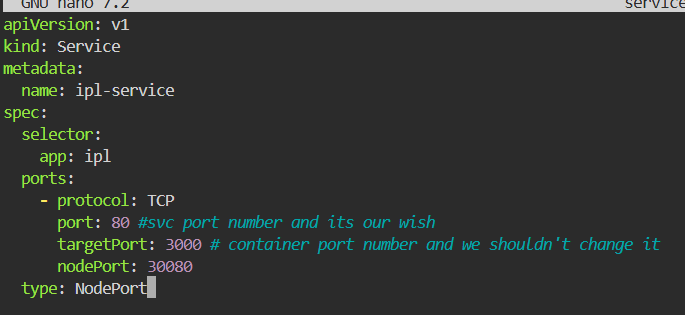
**ClusterIP (default)** → Accessible only within the cluster (for Pod-to-Pod communication).  
  
**Service type load balancer ->**



* 1. Kubernetes will **provision an external IP** (via cloud provider like AWS, GCP, Azure).
  2. You can access your app directly from outside the cluster using:

http://<EXTERNAL-IP>:80

**service type NodePort**



1. Exposes the Service **on each Node’s IP at a static port** (default range: 30000–32767).

2. External traffic → <NodeIP>:<NodePort> → forwarded to Pods.

3. Still uses **ClusterIP** internally, but adds an external port.

Service type = Externalname



1. **type: ExternalName** → Makes it an ExternalName service.

2. **externalName** → DNS name of the external service (resolves to a CNAME).

3. **ports** → Not strictly required, but you can define them to make it easier for apps to connect. Here we use 80

**9. prepare all commands of svc in k8s**

nano servicecluster.yaml = Opens (or creates) a YAML file named servicecluster.yaml in the nano editor

kubectl apply -f servicecluster.yaml = Applies the manifest in servicecluster.yaml to our cluster

ls

nano servicenodeport.yaml

kubectl apply -f servicenodeport.yaml

ls

nano serviceloadbalancer.yaml

ls

kubectl apply -f serviceloadbalancer.yaml

ls

kubectl apply -f service-clusterip.yaml = Creates or updates the ClusterIP service

kubectl apply -f service-nodeport.yaml = Creates or updates the NodePort service

kubectl apply -f service-loadbalancer.yaml = Creates or updates the LoadBalancer service

kubectl apply -f service-externalname.yaml = Creates or updates the ExternalName service

kubectl apply -f serviceloadbalancer.yaml

kubectl apply -f servicecluster.yaml

history

kubectl get svc -n souvik = Lists all services in the namespace souvik.

kubectl describe svc ipl-service-clusterip -n souvik = Detailed description of the service named ipl-service-clusterip in namespace souvik

* Labels, selectors
* Endpoints (which pods it targets)
* Events

kubectl describe svc ipl-service-loadbalancer -n souvik = Detailed description of the service named ipl-service-loadbalancer in namespace souvik

kubectl describe svc ipl-service-nodeport -n souvik = Detailed description of the service named ipl-service-nodeport in namespace souvik

kubectl get svc ipl-service-loadbalancer -n souvik = Shows only that particular service.

history

vi servicecluster.yaml

vi servicenodeport

vi servicenodeport.yaml

vi serviceloadbalancer.yaml

nano serviceexternalname

vi serviceexternalname.yaml

vi serviceexternal.yaml

history