QUES 2 : Kubernetes service types (ClusterIP, NodePort, LoadBalancer)

SOLN:

In Kubernetes, a **Service** is an abstraction that exposes a set of Pods as a network service. While Pods in Kubernetes are ephemeral and can change IPs, Services provide a stable endpoint for reliable communication.

Kubernetes supports **multiple types of Services** to handle different networking needs. These types determine **how and where your application can be accessed** — internally within the cluster or externally from the internet.

The 3 commonly used service types are:

ClusterIP *(Default Service Type)*

- **Purpose:** Makes the service accessible only **within the cluster**.
- **Use Case:** Ideal for internal communication (e.g., frontend ↔ backend).
- **Example:** Backend database or internal microservices.

NodePort

- **Purpose:** Exposes the service on a **static port** on each node's IP.
- **Use Case:** Allows external access by hitting `<NodeIP>:<NodePort>`.
- **Limitations:** Port range is limited (30000–32767), less secure for production.

LoadBalancer

- **Purpose:** Provisions a **cloud load balancer** and assigns a public IP.
- **Use Case:** Best for production when deploying on cloud platforms like AKS, EKS, or GKE.
- **Benefit:** External users can access your service through a single public IP.

My Deployment Using LoadBalancer (As Part of This Task) :

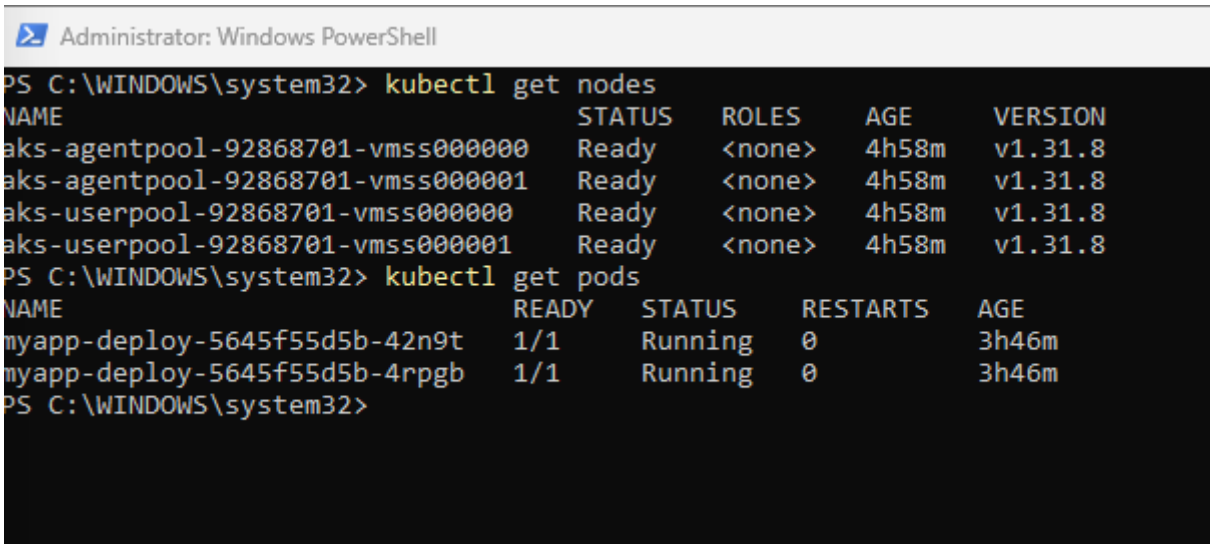
As part of exploring Kubernetes service types, I have already deployed a sample application using the **`LoadBalancer` service type** on my Azure Kubernetes Service (AKS) cluster.

REFERENCE TO QUES 1

This setup allowed me to expose the application externally through a **public IP address** assigned by the cloud provider (Azure). This proves the practical use of the LoadBalancer type in real-world, cloud-based environments.

My Cluster and Pod Setup:

COMMAND USED : kubectl get nodes
kubectl get pods



YAML File Used for LoadBalancer Service:

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
```

```
  app: myapp
ports:
  - protocol: TCP
    port: 80
    targetPort: 80
type: LoadBalancer
```

Result:

After applying this YAML, Kubernetes automatically provisioned a public IP through Azure, enabling users to access the application from outside the cluster.

COMMAND USED : kubectl get svc

This demonstrates how **LoadBalancer services simplify external access** in cloud environments, without needing manual NodePort configuration.

```
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
kubernetes      ClusterIP      10.0.0.1       <none>         443/TCP        5h8m
myapp-service   LoadBalancer   10.0.121.175   4.213.203.76   80:32143/TCP   4h32m
PS C:\WINDOWS\system32>
```

What the Entries Mean:

`kubernetes` (ClusterIP):

- This is **automatically created by Kubernetes**.
- It exposes the **Kubernetes API server** internally to the cluster.
- It's **not related to the app**.
- Yes, it is a ClusterIP type service, but **not your own custom one**.

`myapp-service` (LoadBalancer):

- This is the one **created** for app.
- It has both:
  - A ClusterIP (`10.0.121.175`) → for **internal access**
  - An External IP (`4.213.203.76`) → for **public access via LoadBalancer**
- It also auto-assigned a NodePort: `32143` (which is how LoadBalancer works internally).

What is it?

The `LoadBalancer` type in Kubernetes is used to **expose your service to the internet** by automatically provisioning an **external load balancer** (via the cloud provider like Azure, AWS, or GCP).

How It Works (Internally):

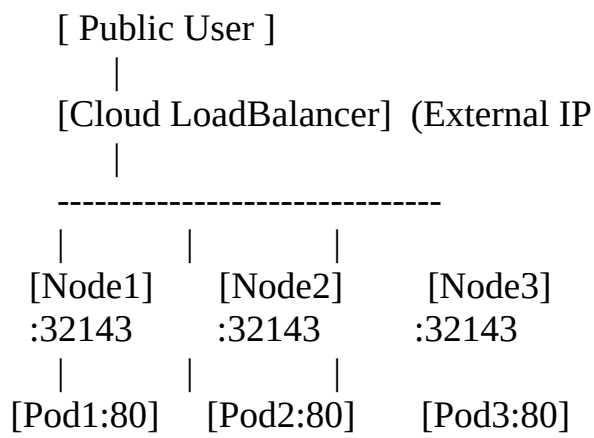
When you create a service of type `LoadBalancer`:

1. Kubernetes **creates a NodePort** service in the background.
2. The cloud provider provisions a **load balancer with a public IP**.
3. The **load balancer forwards traffic** from its public IP → to the **NodePort on cluster nodes** → which routes to your **Pod(s)**.

Flow of Traffic:

1. User accesses the **public IP** of the LoadBalancer.
2. LoadBalancer sends the traffic to one of the **NodePorts**.
3. NodePort forwards the request to the **target port** on the Pod.

Diagram: LoadBalancer Service Flow



`32143` → Auto-assigned NodePort

- **`80` → Target port inside each pod**
- **Traffic from internet → LoadBalancer → NodePort → Pod**

Advantages of LoadBalancer:

- **Simplifies exposure to the internet.**
- **Automatic provisioning of public IP and traffic routing.**
- **Ideal for production deployments on cloud platforms like AKS, EKS, GKE**

SO LETS CONTINUE ON THIS AND CREATE A CLUSTER IP SERVICE AND NODEPORT SERVICE FOR THE SAME IMAGE DEPLOYMENT

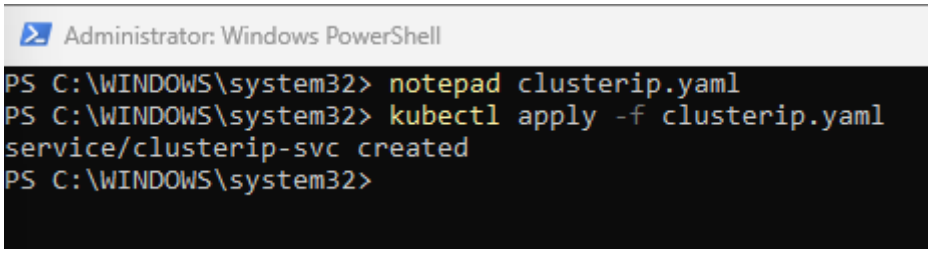
Step 1: Create and Apply ClusterIP and NodePort Services

A. Create `clusterip.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-svc
  labels:
    env: dev
    type: internal
spec:
  selector:
    app: myapp
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Apply it:

```
kubectl apply -f clusterip.yaml
```

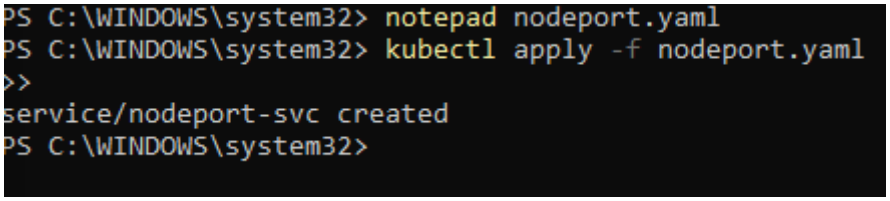


3. Create `nodeport.yaml`

notepad nodeport.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-svc
spec:
  selector:
    app: myapp
  ports:
    - port: 80
      targetPort: 80
      nodePort: 32000
  type: NodePort
```

command used :
kubectl apply -f nodeport.yaml



4. Verify Services

kubectl get svc

You should now see 3 services:

- `myapp-service` (LoadBalancer)
- `clusterip-svc` (ClusterIP)
- `nodeport-svc` (NodePort)

```
Administrator: Windows PowerShell
PS C:\WINDOWS\system32> kubectl get svc
>>
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)        AGE
clusterip-svc   ClusterIP      10.0.51.13     <none>         80/TCP         3m4s
kubernetes      ClusterIP      10.0.0.1       <none>         443/TCP        5h22m
myapp-service   LoadBalancer   10.0.121.175   4.213.203.76   80:32143/TCP   4h46m
nodeport-svc    NodePort       10.0.100.208   <none>         80:32000/TCP   69s
PS C:\WINDOWS\system32>
```

5. Optional: Describe to Check if Pod Is Reached

```
PS C:\WINDOWS\system32> kubectl describe svc clusterip-svc
Name:                     clusterip-svc
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=myapp
Type:                     ClusterIP
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.0.51.13
IPs:                      10.0.51.13
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
Endpoints:                10.244.1.241:80,10.244.2.142:80
Session Affinity:         None
Internal Traffic Policy:  Cluster
Events:                   <none>
PS C:\WINDOWS\system32>
```

```
PS C:\WINDOWS\system32> kubectl describe svc nodeport-svc
Name:                     nodeport-svc
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=myapp
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.0.100.208
IPs:                      10.0.100.208
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32000/TCP
Endpoints:                10.244.1.241:80,10.244.2.142:80
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
PS C:\WINDOWS\system32>
```

so untill this :

Service	Type	Internal Access	External Access	Status
`clusterip-svc`	ClusterIP	✅ Yes (within cluster)	❌ No	Working
`nodeport-svc`	NodePort	✅ Yes (if port-forwarded)	❌ No (no external node IP)	Working
`myapp-service`	LoadBalancer	✅ Yes	✅ Yes (`4.213.203.76`)	Working

WE CANNOT ACCESS PUBLICALLY CAUSE :
`NodePort` mapped the app to a static port (32000) on each node, but due to lack of external node IPs in AKS, it cannot be accessed directly from the internet.

WHAT WE CAN DO TO ACCESS IT PUBLICALLY:

Option 1: Manually Expose a VM Node to the Internet (NOT recommended)

Steps (Advanced & Not Secure for Production):

1. Create a **public IP address** in Azure.
2. Assign that IP to a **network interface** on a VM Scale Set (one of the AKS nodes).
3. Open firewall port (e.g., 32000).
4. Then you can access:

```
http://<Public-Node-IP>:<NodePort>
```

Option 2: Use a LoadBalancer to Simulate NodePort (Cleanest + Practical)

Since Azure doesn't expose nodes directly, you can:

1. Keep your service as NodePort
2. **Manually create a LoadBalancer** resource in Azure Portal

3.Point it to backend pool → Node IPs → NodePort

4.Result: Public IP hits NodePort behind the scenes

But this defeats the purpose because...

Kubernetes LoadBalancer service already does this automatically!

SO ITS DONE FILNALLY:

In this task, I explored all three main types of Kubernetes Services: ClusterIP, NodePort, and LoadBalancer.

I deployed a sample app (myapp-deploy) and connected it to all 3 services:

- ClusterIP was used to enable internal communication inside the cluster.
- NodePort mapped the app to a static port (32000) on each node, but due to lack of external node IPs in AKS, it cannot be accessed directly from the internet.
- LoadBalancer was used to expose the app to the public internet via Azure’s managed load balancer, which provided a working external IP (4.213.203.76).

SO WE CAN UNDER STAND THE USE CASES AS :

Real-World Use Case Comparison of Kubernetes Service Types

◈ 1. ClusterIP – Internal Service Communication Only

Theory:
ClusterIP exposes the service **only inside the Kubernetes cluster**, typically used for **backend-to-backend** communication between microservices.

Real-World Example:
In a real e-commerce site, the **frontend service** (React or Angular) calls the **backend service** (Node.js or Django) via ClusterIP. The backend in turn calls **database** or **authentication services**, all of which are invisible outside the cluster.

Why?
These components don’t need to be public. Using ClusterIP keeps them secure and isolated.

2. NodePort – Debug or Dev-Level External Access

Theory:
NodePort exposes the service on a **fixed port (30000–32767)** on every node's IP. It’s often used for **quick testing or temporary external access**, especially in **bare metal or local setups**.

Real-World Example:
In a private company network, a dev team may use NodePort to expose a service to their **office network** so team members can test the staging build by visiting http://<node-ip>:32000.

Why?
No need for a full cloud LoadBalancer in non-production or internal settings. But not ideal for public or large-scale access.

3. LoadBalancer – Production-Grade Public Access

Theory:
LoadBalancer provisions a **cloud-managed external IP** and routes public traffic to the app. It's best for production websites and APIs exposed to the internet.

Real-World Example:
In real production apps like **Tabcura, Amazon, or Netflix**, frontend services are exposed via LoadBalancer.
For instance:
https://www.netflix.com → hits a **cloud LoadBalancer** → routes to Kubernetes pods hosting their frontend.

Why?
It provides scalability, public reach, and is managed by the cloud (e.g., Azure Load Balancer in AKS).