1. Orchestration tools, such as Kubernetes, play a key role in the server infrastructure for the modern applications.
   a) Explain how these tools help manage and scale application servers.
      Orchestration tools like Kubernetes act as a control system for managing large numbers of application containers (e.g., Docker containers) across clusters of servers. Instead of manually starting, stopping, or monitoring containers, Kubernetes automates these tasks.
      - Applications run reliably by monitoring container health and automatically restarting failed ones.
      - Workloads are evenly distributed across available servers (load balancing).
      - Applications scale up or down seamlessly based on demand (e.g., automatically adding more container instances during high traffic).Resources such as CPU, memory, and storage are allocated efficiently across the cluster.

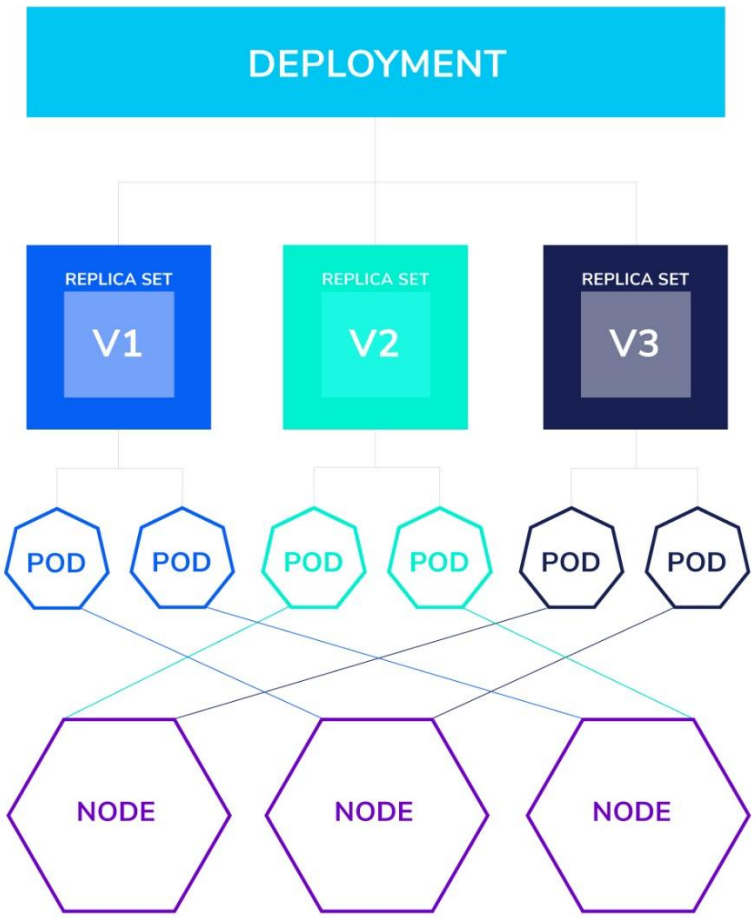
      In short, orchestration tools transform a pool of servers into a single, manageable resource that can adjust dynamically to application needs.
   b) Describe how orchestration tools facilitate automated deployment, scaling, and management of application servers.
      Orchestration tools provide declarative management. For example, developers define the desired system state and the tool continuously works to maintain that state.
      - **Automated Deployment**: Orchestration handles rolling updates and rollbacks. For example, if a new version of an app is deployed, Kubernetes gradually replaces old containers with new ones while keeping the system online. If something goes wrong, it can roll back automatically.
      - **Automated Scaling**: Kubernetes can monitor metrics like CPU usage or incoming requests and automatically add or remove replicas of containers. This means applications handle traffic spikes without human intervention.
      - **Automated Management**: Orchestration takes care of networking, persistent storage management+ and self-healing. For example, if a node crashes, Kubernetes reschedules the affected containers on healthy nodes.

2. Explain the difference between a Pod, Deployment, and Service.

   Pod:
   - Pods can be defined as the smallest computing unit that is assigned an individual IP address and can be deployed and managed.
   - A Pod is essentially a wrapper around one or more containers that are tightly coupled, which often sharing storage, networking, and resources.
   - Typically, a Pod runs a single application container, but it can include sidecar containers like a logging or proxy container.
   - Pods are short-lived. If a Pod dies, it is not restarted directly. Instead, higher-level controllers like Deployments handle replacement.
   - For example, a Pod can run one instance of a Python Flask web server.

   Deployment:
   - A Kubernetes deployment provides a means of changing or modifying the state of a pod, which may be one or more containers that are running, or a group of duplicate pods, known as ReplicaSets. Using a deployment allows us to easily keep a group of identical

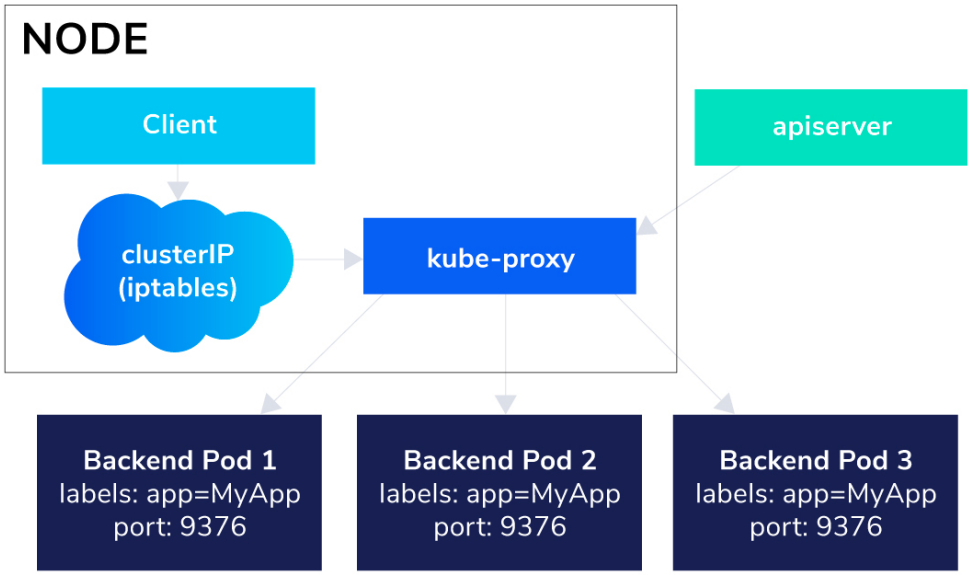
pods running with a common configuration. Once we have defined and deployed our deployment Kubernetes will then work to make sure all pods managed by the deployment meet whatever requirements we have set.



- It ensures that the desired number of Pods are running, handles scaling, rolling updates and rollbacks.
- If we say "I want 5 replicas of my web server," the Deployment ensures 5 identical Pods are created and maintained.
- It replaces Pods automatically if they fail or if we roll out a new version.
- For example, deployment use cases include running multiple instances of an application, scaling the number of instances of an application up or down, updating every running instance of an application and rolling back all running instances of an application to another version.
- While deployments do define how our applications will run they do not guarantee where our applications will live within our cluster. For example, if our application requires an instance of a pod on every node we will want to use a DaemonSet. For stateful applications, a StatefulSet will provide unique network identifiers, persistent storage, and ordered deployment/scaling.
- Always understand our application's desired behavior before determining how to put it in a cluster. While deployments define how our application runs, it doesn't tell anything else in the cluster how to find or communicate with the resources it manages. This is where kubernetes services come in.

Services:

- A Service provides a stable network endpoint (IP and DNS) to access a set of Pods. When using a Kubernetes service, each pod is assigned an IP address. As this address may not be directly knowable, the service provides accessibility, then automatically connects the correct pod, as the example below shows.



- When a service is created it publishes its own virtual address as either an environment variable to every pod or, if your cluster is using coredns, as a dns entry any pod can attempt to reach. In the event of any changes to the number of available pods the service will be updated and begin directing traffic accordingly with no manual action required.
- Services are not just for pods. Services can abstract access to DBs, external hosts, or even other services. In some of these cases you may need an Endpoint object, but for internal communication this is not required.
- For example, a Service might expose the 5 Flask Pods as a single endpoint http://my-flask-service and load-balance requests among them.

3. What is a Namespace in Kubernetes? Please list one example.

   Definition: A Namespace in Kubernetes is a way to logically divide a cluster into separate, isolated environments. It helps organize and manage resources like Pods, Deployments and Services so that multiple teams, projects or environments can share the same cluster without interfering with each other. Namespaces provide isolation, to avoid the clash among resources in different namespaces, resource management and access control.

   Example: Suppose a company uses one Kubernetes cluster for both development and production. They could create two namespaces: dev for development workloads like frequent updates and testing versions, and prod for production workloads like stable, high-availability services. In the dev namespace, the team deploy a frontend Service for testing. At the same time, in the prod namespace, the team can also have a frontend Service and they won't conflict because they're isolated by namespaces.

4. Explain the role of the Kubelet. How do you check the nodes in a Kubernetes cluster? (kubectl command expected)

   The Kubelet is an essential Kubernetes agent that runs on every node in the cluster. Its main

role is to ensure that containers are running in Pods as specified by the Kubernetes control plane.

Role of the Kubelet

- Communicates with the API server to receive Pod specifications (PodSpecs).
- Starts and monitors containers through the container runtime, e.g., Docker, containerd.
- Reports node and Pod status back to the control plane.
- Performs health checks and restarts containers if they fail.
- Ensures the node stays in the desired state defined by Kubernetes.

In short, the Kubelet is the "node agent" that makes sure Pods run as intended on each node.

Command:

- ***kubectl get nodes***: list all nodes in the cluster along with their status, roles, age, and version.
- ***kubectl describe node <node-name>***: shows detailed information about a specific node, such as resource capacity, conditions, labels, and running Pods.

5. What is the difference between ClusterIP, NodePort, and LoadBalancer services?

ClusterIP (default)

- **What it does:** Exposes the Service on an internal, cluster-only IP address.
- **Use case:** Other Pods within the same cluster can access it, but it's **not accessible from outside** the cluster.
- **Example scenario:** A backend database service (e.g., MySQL) that should only be accessed by frontend Pods inside the cluster.

NodePort

- **What it does:** Exposes the Service on a static port (between 30000–32767) on **each node's IP address**.
- **How it works:** External clients can send requests to <NodeIP>:<NodePort> and Kubernetes forwards them to the right Pod.
- **Use case:** When you want to make your Service accessible externally without a cloud load balancer.
- **Example scenario:** Exposing a test web application so developers can access it using the node's IP.

LoadBalancer

- **What it does:** Creates an **external load balancer** (usually provided by the cloud provider, like AWS ELB or GCP Load Balancer) and routes traffic to the Service.
- **How it works:** Assigns a **public IP address** that balances incoming traffic across all Pods behind the Service.
- **Use case:** When you need production-ready external access with built-in load balancing.
- **Example scenario:** A production frontend web application that must be accessible on the internet with high availability.

6. How do you scale a Deployment to 5 replicas using kubectl?

kubectl scale deployment <deployment-name> --replicas=5:

- kubectl: Kubernetes command-line tool for interacting with the cluster.
- scale: Indicates scaling operations.
- deployment: Specifies the resource to scale; in this case, a Deployment (deployment

controller).

- <deployment-name>: The name of your Deployment, for example, nginx-deployment.
- --replicas=5: Specifies the number of replicas; in this case, 5 Pod replicas.

To verify:

kubectl get deployment nginx-deployment

- get: View resources.
- deployment: Specifies the resource type.
- nginx-deployment: Specifies the name of the Deployment to view.

7. How would you update the image of a Deployment without downtime?

A. Update Image

kubectl set image deployment <deployment-name> <container-name>=<new-image>:<tag>

- kubectl set image: Modifies the container image in a resource (here, a Deployment).
- deployment <deployment-name>: Specifies the name of the Deployment to update.
- <container-name>=<new-image>:<tag>: Specifies the container and image to update.

B. Edit Deployment

kubectl edit deployment <deployment-name>

- Open the Deployment configuration file (YAML format) and manually modify it.
- Find the spec.template.spec.containers.image field and update the image to the new version.
- Saving the file will trigger a rolling update.

C. Apply a New Configuration File

kubectl apply -f deployment.yaml

- -f deployment.yaml : Specifies the file containing the Deployment definition.
- If the image in this file is modified, Kubernetes automatically updates the Deployment and performs a rolling upgrade.

D. Check Rollout Status

kubectl rollout status deployment <deployment-name>

- Check the progress of the Deployment update. If the update is not yet complete, it will display "Waiting"; when it is complete, it will display "successfully rolled out."

E. Rollback an Update

kubectl rollout undo deployment <deployment-name>

- If there are issues with the new version, you can undo the update and revert to the previous version.
- Kubernetes will then perform another rolling update, reverting the pods to the previous image.

8. How do you expose a Deployment to external traffic?

A. Using a Service of type NodePort

kubectl expose deployment <deployment-name> --type=NodePort --port=<port>

- **--type=NodePort**: Exposes the Deployment on each Node's IP at a static port.
- **--port**: The port your application is listening on inside the Pod.
- Then you can access the app at <NodeIP>:<NodePort>.

B. Using a Service of type LoadBalancer (cloud environments)

kubectl expose deployment <deployment-name> --type=LoadBalancer --port=<port>

- --type=LoadBalancer: Requests an external load balancer from your cloud provider (AWS ELB, GCP LB, Azure LB, etc.).
- It assigns a public IP or hostname that routes traffic to your Pods.

C. Using an Ingress (with Ingress Controller)

First, expose the Deployment internally with a ClusterIP Service:

kubectl expose deployment my-app --type=ClusterIP --port=80

Then create an Ingress resource to define external access rules (e.g., domain name, TLS, paths). Ingress controllers (NGINX, Traefik, HAProxy, etc.) handle routing traffic from the outside world to your Service.

Example Ingress YAML:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app
            port:
              number: 80
```

9. How does Kubernetes scheduling decide which node a Pod runs on?

A. Filter

The scheduler first eliminates nodes that **cannot** run the Pod. There are some cases:

- Insufficient CPU / memory / storage
- Node taints without matching Pod tolerations
- Node labels don't match Pod's nodeSelector / nodeAffinity
- Pod affinity/anti-affinity rules not satisfied

kubectl describe pod <POD_NAME>: to see why a Pod couldn't be scheduled.

B. Score

Among the feasible nodes, the scheduler **scores** them to find the best fit. Scoring rules include:

- Prefer nodes with more free resources
- Spread Pods across zones or nodes
- Respect "preferred" affinity rules

kubectl -n kube-system get configmap kube-scheduler -o yaml: to see what scoring plugins are active (scheduler config)

C. Bind

The scheduler updates the Pod's spec.nodeName with the chosen node. The kubelet on that node then pulls images and starts containers.

kubectl get pod <POD_NAME> -o wide: to check which node a Pod is bound to. The NODE column shows the assigned node.

10. What is the role of Ingress and how does it differ from a Service?

Ingress is a higher-level method that provides an API for HTTP/HTTPS access to externally exposed services. Rather than assigning external IPs to each Service (via LoadBalancer or NodePort), Ingress consolidates access through a single entry point and routes traffic to the appropriate Service based on defined rules. Ingress understands web concepts such as hostnames, URIs, and paths. It allows you to map traffic to different backends using rules defined through the Kubernetes API.

Difference:

	Kubernetes Service	**Kubernetes Ingress**
Purpose	Internal service discovery and traffic routing	External traffic routing to services
Scope	Manages Traffic inside the cluster	Manages traffic from outside the cluster
Types	ClusterIP, NodePort, LoadBalancer	Uses Ingress Controllers (e.g. Envoy Gateway)
Load Balancing	Internal load balancing across Pods	HTTPS based traffic routing
TLS Support	Not built in – requires manual addition	Supports TLS termination for secure traffic

How service and Ingress work together:

Kubernetes deployments typically use both Services and Ingress to manage traffic flow. Use a Service to enable stable internal access to a set of Pods. Use Ingress as a gateway to direct external traffic to the appropriate application or microservice based on hostname, path, or other routing rules. Modern deployments often combine Ingress with a service mesh like Istio to add advanced traffic control, security, and observability.