1. How do Kubernetes and Docker complement each other?

==========

Docker packages apps into containers.
Kubernetes orchestrates and manages container lifecycle, networking, scaling, and updates.
Docker is the runtime used in many Kubernetes clusters (though alternatives exist).

2. Explain Kubernetes Architecture / Explain the Kubernetes architecture and the roles of its components.

==========

Master Node	**Components:**
API Server:	**Frontend to Kubernetes control plane, exposes Kubernetes API.**
etcd:	**Distributed key-value store to hold cluster state data.**
Controller Manager:	**Runs controllers like Node Controller, Replication Controller.**
Scheduler:	**Assigns pods to nodes based on resource availability.**
Worker Node	**Components:**
Kubelet:	**Agent running on nodes, ensures containers are running in Pods.**
Kube-proxy:	**Network proxy for service abstraction, manages network rules.**
Container Runtime:	**Runs containers (Docker, containerd, CRI-O).**

Step-by-step:
API Server processes user commands from CLI or UI. -----> Scheduler places pods on nodes -----> Controllers ensure desired state is met (like replica count). --------> Kubelet communicates with API server to manage pods on nodes.

3. Explain Pod Lifecycle and Management / Describe the Pod lifecycle and how Kubernetes manages Pods.

==========

Pod Lifecycle phases: Pending → Running → Succeeded/Failed → Unknown
--------- Kubernetes manages Pods ----------
Kubelet creates the pod containers using container runtime. -------> If pod dies, Controllers recreate pods (if it is under ReplicaSet/Deployment). ------> Liveness and readiness probes help Kubernetes know if pod is healthy and ready.

4. How do you perform rolling updates and rollbacks in Kubernetes?

==========

When we use kubectl apply -f deployment.yaml to update deployment Kubernetes creates new ReplicaSets with the new version.
The Rolling update replaces pods gradually (default maxUnavailable=25%, maxSurge=25%).
For Rollback: kubectl rollout undo deployment/<deployment-name>

Example Step-by-step:
Modify image/version in deployment manifest. ------> Apply changes using kubectl -------> new pods start before old pods terminate. ------> Monitor rollout status: kubectl rollout status deployment/<name> ------> Undo if needed to previous stable version.

+++++ Explanation ++++++
Rolling Update: Gradually replaces Pods of the old version with Pods of the new version without downtime.
Rollback: Reverts the Deployment to a previous version in case the new version has issues.

Step 1: Create an initial Deployment Let's create a Deployment with an NGINX container version 1.14.2.

```
# nginx-deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
      name: nginx-deployment
spec:
      replicas: 3
      selector:
      matchLabels:
          app: nginx
 template:
     metadata:
          labels:
              app: nginx
      spec:
          containers:
          - name: nginx
            image: nginx:1.14.2
            ports:
            - containerPort: 80
```

Apply this Deployment
kubectl apply –f nginx-deployment
check the pods
kubectl get pod -l app=nginx

Step 2: Let's Perform a rolling update,

For an example Let's update the image version from nginx:1.14.2 to nginx:1.16.1 in the YAML file and apply the deployment or do it directly using the command:

kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1

Kubernetes will start replacing pods gradually with new pods using the new image version without downtime.
Check rollout status: kubectl rollout status deployment/nginx-deployment
Step 3: Verify the update by checking the pods' image versions:
kubectl get pods –l app=nginx –o jsonpath="{.items[*].spec.containers[0].image}"
You should see all pods running nginx:1.16.1.
Step 4: Rollback to previous version if needed
If the new version has problems, rollback to previous version: kubectl rollout undo deployment/nginx-deployment
Check rollout status again: kubectl rollout status deployment/nginx-deployment
check rollout history and status or View rollout history: kubectl rollout history deployment/nginx-deployment
View details about a specific revision: kubectl rollout history deployment/nginx-deployment --revision=1

5. Networking in Kubernetes / Q: Explain how networking works in Kubernetes.

==========

K8 Networking Aspect	What it Does	Example Resource
Pod-to-Pod communication	Direct IP communication between Pods	Pod
Stable network endpoint	Expose group of Pods via DNS/IP	Service (ClusterIP)
External access	Access Service from outside	Service (NodePort/LoadBalancer)
Network control	Restrict traffic flow between Pods	NetworkPolicy

==========----------==========

External access Uses NodePort/LoadBalancer; still may involve kube-proxy but for outside-in traffic.
Stable network endpoint Handled via Services. Kube-proxy manages IP routing to backends
++++++++++
STEP 1 -- Pod-to-Pod Direct Communication
List all pods and get their IPs: kubectl get pods -o wide
Example output:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
pod-a	1/1	Running	0	10m	10.244.1.5	node-1
pod-b	1/1	Running	0	8m	10.244.2.7	node-2

Here, pod-a has IP 10.244.1.5, and pod-b has IP 10.244.2.7.
Pick a pod, exec into it: kubectl exec -it <pod-name> -- sh
kubectl exec -it pod-a -- sh You are now "inside" pod-a's container.

Try ping or curl another pod using its IP: ping <other-pod-ip>

ex → ping 10.244.2.7

You should see responses like:

PING 10.244.2.7 (10.244.2.7): 56 data bytes
64 bytes from 10.244.2.7: icmp_seq=0 ttl=64 time=0.123 ms
64 bytes from 10.244.2.7: icmp_seq=1 ttl=64 time=0.100 ms
Or
if it's an HTTP server running on pod-b, you can:
ex → curl http://10.244.2.7:8080 → you will get a response directly
You'll see that pod-to-pod communication works within the cluster without NAT.

Step 2 → Service Discovery / Services Provide Stable Network Endpoints

---

- **Pods are ephemeral and may be recreated with different IPs.**
- **Kubernetes Services provide a stable IP and DNS name to access a group of Pods.**
- **Services load balance traffic to Pods matching a label selector.**

Create a Pod running nginx:

```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
    name: nginx-pod
    labels:
        app: nginx
spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

STEP - 1

Create a Service to expose nginx Pod inside the cluster:

```
# nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
    name: nginx-service
spec:
    selector:
        app: nginx
 ports:
    - protocol: TCP
      port: 80       # Service port
      targetPort: 80  # Pod container port
```

STEP - 2

Apply both manifest : kubectl apply -f nginx-pod.yaml and kubectl apply -f nginx-service.yaml
Check Service IP: kubectl get svc nginx-service
Output:

NAME	TYPE	CLUSTER-IP	PORT(S)	AGE
nginx-service	ClusterIP	10.96.25.100	80/TCP	1m

You can now access nginx via this stable IP or DNS name nginx-service from other Pods inside the cluster.

Accessing Services from other pod Inside the Cluster

Lets Create a Pod with a curl client to test access:

```
# curl-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: curl-pod
spec:
  containers:
  - name: curl
    image: curlimages/curl
    command: ["sleep", "3600"]
```

Apply: kubectl apply -f curl-pod.yaml

Exec into curl-pod and curl nginx service by DNS name:

kubectl exec -it curl-pod – curl http://nginx-service

You should get the nginx welcome page HTML output.

Step 3 → Exposing Services Outside the Cluster

To allow access from outside, change Service type to NodePort or LoadBalancer.

Example NodePort service:

```
apiVersion: v1
kind: Service
metadata:
    name: nginx-nodeport
spec:
    selector:
       app: nginx
    ports:
       - port: 80
         targetPort: 80
         nodePort: 30080
    type: NodePort
```

```
kubectl apply -f nginx-nodeport.yaml

kubectl get nodes -o wide

Now you can access nginx from outside by hitting:

http://<Node-IP>:30080
```

Step 4 → Controlling Traffic with Network Policies

By default, all Pods can talk to each other. You can create NetworkPolicies to restrict which Pods can talk to which.

Example allowing only Pods labeled role=frontend to access nginx on port 80:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: allow-frontend
spec:
    podSelector:
        matchLabels:
            app: nginx
    ingress:
        - from:
            - podSelector:
                matchLabels:
                    role: frontend
          ports:
            - protocol: TCP
              port: 80
```

6. Explain the difference between Deployment, ReplicaSet, StatefulSet, and DaemonSet Controllers.

==========

Answer:

Deployment: Manages stateless apps, handles rolling updates and rollbacks.

ReplicaSet: Ensures specified number of pod replicas are running, usually managed by Deployments.

StatefulSet: Manages stateful apps, provides stable identities and persistent storage.

DaemonSet: Ensures a copy of a pod runs on all (or some) nodes, e.g., for logging or monitoring agents.

7. Explain how Kubernetes handles storage?

==========

Kubernetes supports Volumes which persist data beyond the lifecycle of a Pod. There is different types of volumes: emptyDir, hostPath, persistentVolumeClaim (PVC), etc.

PersistentVolumes (PV) represent actual storage resources. It is a piece of storage in the cluster provisioned by an admin or dynamically provisioned using a StorageClass (e.g., SSD vs HDD, reclaim policy, etc.).

PersistentVolumeClaim (PVC): A user's request for storage.

1. Define a StorageClass

```
# storage-class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
      name: standard
provisioner: kubernetes.io/aws-ebs      # Use a provisioner
parameters:                                    (e.g., aws-ebs, gce-pd, csi)
      type: gp2
reclaimPolicy: Retain
volumeBindingMode: WaitForFirstConsumer
```

2. Create a PersistentVolumeClaim (PVC)

```
# pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: my-pvc
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
            storage: 1Gi
    storageClassName: standard
```

Apply it: kubectl apply -f storage-class.yaml and kubectl apply -f pvc.yaml

This triggers dynamic provisioning using the StorageClass. Kubernetes creates a PersistentVolume behind the scenes.

Now Create a Pod which will use the PVC

```
# pod_using_pvc.yaml
apiVersion: v1
kind: Pod
metadata:
    name: pvc-demo-pod
spec:
    containers:
        - name: app
          image: busybox
          command: ["sleep", "3600"]
          volumeMounts:
              - mountPath: "/data"
                name: storage
    volumes:
        - name: storage
          persistentVolumeClaim:
                claimName: my-pvc
```

Apply it: kubectl apply -f pod_using_pvc.yaml

Now your container can write to /data, and data will persist as long as the volume exists, even if the Pod is deleted.

```
kubectl get pvc
kubectl get pv
kubectl get pod pvc-demo-pod -o yaml
```

8. What are Controllers in Kubernetes?

==========

Controllers monitor the state of the cluster via the API server and make or request changes to move the current state towards the desired state.

They automate tasks such as: 1) Ensuring a certain number of pods are running , 2) Restarting failed containers , 3) Managing updates , 4) Scaling applications

Types of Built-in Controllers

ReplicaSet Controller –	**Ensures a specific number of Pod replicas are running.**
Job Controller –	**Ensures Pods complete a specific task to completion.**
DaemonSet Controller –	**Ensures a copy of a Pod runs on all or some Nodes.**
StatefulSet Controller –	**Manages stateful applications (with persistent identity).**
CronJob Controller –	**Manages time-based jobs.**

Step 1: Define the ReplicaSet Controller

This YAML defines a Deployment that manages 3 replicas of an NGINX web server.

```yaml
# nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
        name: nginx-deployment
spec:
        replicas: 3
        selector:
            matchLabels:
                app: nginx
 template:
    metadata:
        labels:
                app: nginx
    spec:
        containers:
        - name: my-nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
```

Apply the Deployment YAML : kubectl apply -f nginx-deployment.yaml

Check the Deployment Status kubectl get deployments

You'll see something like:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	10s

------> This means the Deployment controller created 3 Pods via a ReplicaSet.

Verify the Pods

kubectl get pods -l app=nginx

Should return 3 pods like:

nginx-deployment-xxxxx Running
nginx-deployment-yyyyy Running
nginx-deployment-zzzzz Running

Step 2 → Define StatefulSet Controller

StatefulSet Controller manages stateful applications — those needing stable identities, persistent storage, and ordered deployment. Each pod gets a unique name and persistent volume. Useful for databases (like MySQL, Cassandra, etc.)

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
    name: web
spec:
    serviceName: "nginx"
    replicas: 3
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx
                  volumeMounts:
                        - name: www
                          mountPath: /usr/share/nginx/html
    volumeClaimTemplates:
        - metadata:
                name: www
          spec:
                accessModes: [ "ReadWriteOnce" ]
                resources:
                    requests:
                        storage: 1Gi
```

Use-case: Databases, message queues, or anything needing stable storage and identity.

Step 5 → CronJob Controller

Runs Jobs on a schedule, like a Linux cron job.

Can manage retries and failed runs. Syntax follows standard cron format.

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
    name: hello-cron
spec:
    schedule: "*/1 * * * *"              # Every minute
    jobTemplate:
        spec:
            template:
                spec:
                    containers:
                        - name: hello
                          image: busybox
                          command: ["echo", "Hello from CronJob!"]
                    restartPolicy: OnFailure
```

Use-case: Periodic backups, reporting, monitoring jobs.

Step 3 → DaemonSet Controller

Ensures that a copy of a Pod runs on every / selected Node in the cluster. Commonly use for logging, monitoring, or network agents. Use-case: Run agents like Fluentd, Prometheus Node Exporter, etc.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
    name: node-exporter
spec:
    selector:
        matchLabels:
            name: node-exporter
    template:
        metadata:
            labels:
                name: node-exporter
        spec:
            containers:
                - name: node-exporter
                  image: prom/node-exporter
```

Use-case: Run agents like Fluentd, Prometheus Node Exporter, etc.

Step 4 → Job Controller

Creates Pods that run to completion (once, not forever).
Perfect for batch jobs or one-time tasks. Retries if Pods fail.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
    name: hello-job
spec:
    template:
        spec:
            containers:
                - name: hello
                  image: busybox
                  command: ["echo", "Hello from Job!"]
                  restartPolicy: Never
    backoffLimit: 4     ----------->   4 retries if pod fails
```

Use-case: One-off tasks like database migrations, data processing

9. How do you secure a Kubernetes cluster?

==========

Use RBAC (Role-Based Access Control) for permission management.
Enable network policies to control pod-to-pod communication.
Use TLS for communication between components.
Use image scanning and trusted registries.
Enable Pod Security Policies or Pod Security Admission for runtime restrictions.
Encrypt secrets at rest.
Regularly patch and update cluster components.

Step 1. Use Role-Based Access Control (RBAC)

RBAC use to limit who can perform which action in your cluster. For that we need to create RBAC role and bind it to the user.

Let's create a role that allows reading pods:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
    namespace: default
    name: pod-reader
    rules:
        - apiGroups: [""]
          resources: ["pods", "services"]
          verbs: ["get", "watch", "list"]
```

1st step

Now bind role to a user:

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

2nd step

Apply: kubectl apply -f role.yaml and kubectl apply -f rolebinding.yaml

Enable Authentication and Authorization using client certificates, OIDC, or cloud provider IAM.

- **Use OIDC with Google or Azure AD to authenticate users.**
- **Configure --authentication-mode=Webhook and --authorization-mode=RBAC in the API server.**

Step 2 → Use Network Policies

Control traffic between pods and namespaces, by enabling a network plugin that supports policies (e.g., Calico, Cilium)

Kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml → Download and apply Calico manifests

kubectl get pods -n calico-system → Verify Calico pods are running

Block all traffic except from a specific app using NetworkPolicy :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: allow-nginx
    namespace: default
spec:
    podSelector:
        matchLabels:
            app: nginx
    policyTypes:
        - Ingress
    ingress:
        - from:
            - podSelector:
                matchLabels:
                    app: frontend
```

Step 3 → Use TLS for communication between components.

As etcd contains all cluster secrets and configurations.

- Encrypt etcd at rest.
- Use TLS for etcd client and peer connections.
- Limit access to etcd to the API server only.

Vi /etc/kubernetes/encryption-config.yaml

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
        - secrets
    providers:
        - aescbc:
            keys:
                - name: key1
                  secret: <base64-encoded-secret>
        - identity: {}
```

Edit /etc/kubernetes/manifests/kube-apiserver.yaml file and add the --encryption-provider-config /etc/kubernetes/encryption-config.yaml

Restart the kube-apiserver if needed

Step 4 → Scan Container Images

Use tools like Trivy, Clair, or Aqua Security scan the container image

Ex → trivy image nginx:latest

10. What strategies are available for deploying applications in Kubernetes?

==========

Rolling Update: Incrementally updates pods with zero downtime.

Blue/Green Deployment: Runs two environments and switches traffic from old to new.

Canary Deployment: Gradually rolls out new versions to a subset of users.

Recreate: Shuts down old version before starting the new one.

Rolling update → explained at above

Blue/green deployment →

In case of blue/green deployment the downtime and risk are reduces by running two production environments (Blue and Green) in parallel. Once test is success we can delete old version.

Lets think the Current version is running in the Blue environment. Green is a new version of your app, deployed alongside Blue. Then tests/health checks are run against Green. Once health check is good switch traffic from Blue to Green via Ingress or LoadBalancer. Once confirmed, Blue is optionally deleted or kept for rollback.

To perform blue/green deployment there should be two deployment manifest & a selector base service manifest file.

Vi deployment-blue.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
      name: myapp-blue
spec:
      replicas: 3
      selector:
           matchLabels:
                app: myapp
                version: blue
     template:
           metadata:
                labels:
                     app: myapp
                     version: blue
           spec:
                containers:
                     - name: myapp
                       image: myapp:1.0
                       ports:
                            - containerPort: 80
```

Vi deployment-green.yaml

Same as above , but the version: green and image: new_image_with_new_version as below.

```
metadata:
name: myapp-green
...
version: green
image: myapp:2.0
```

vi selector_base_service.yaml

```
apiVersion: v1
kind: Service
metadata:
     name: myapp-service
spec:
     selector:
          app: myapp
          version: blue       # Change to 'green' during rollout
     ports:
          - protocol: TCP
            port: 80
            targetPort: 80
```

vi blue_green_deploy.yml ---------- it is in Ansible

```
- name: Blue-Green Deployment on Kubernetes
  hosts: localhost
  gather_facts: no
  vars:
    new_version: green  # or blue
    old_version: blue   # or green
    image_tag: "2.0"

  tasks:
    - name: Apply new version deployment
      kubernetes.core.k8s:
        state: present
        definition: "{{ lookup('file', 'deployment-' + new_version + '.yaml') }}"

    - name: Wait for rollout to complete
      command: >
              kubectl rollout status deployment/myapp-{{ new_version }}
      register: rollout_status
      until: rollout_status.stdout.find('successfully rolled out') != -1
      retries: 10
      delay: 10
```

```
- name: Run health checks (optional)
  uri:
      url: "http://{{ new_version }}.myapp.internal/health"
      status_code: 200
  register: health_result
  until: health_result.status == 200
  retries: 5
  delay: 5
- name: Switch service to new version
  kubernetes.core.k8s:
    state: present
    definition:
      apiVersion: v1
      kind: Service
      metadata:
        name: myapp-service
      spec:
        selector:
          app: myapp
          version: "{{ new_version }}"
        ports:
          - protocol: TCP
            port: 80
            targetPort: 80

- name: (Optional) Delete old version deployment
  kubernetes.core.k8s:
    state: absent
    kind: Deployment
    name: myapp-{{ old_version }}
```

Canary Deployment → Canary deployment is a process to roll out a new version of an application to a small subset of users before exposing it to the entire user base. This lets you test the new version in production with real traffic, minimizing risk by limiting exposure to potential issues.

For Canary deployment we need to follow below steps

Existing app version (stable).
New app version (canary) deployed alongside stable.
Traffic is split between stable and canary pods, e.g., 90% stable, 10% canary.
You monitor canary behavior.
If successful, shift 100% traffic to new version and remove old version.

```
ansi-canary-folder/
├── canary-deployment.yaml
├── stable-deployment.yaml
├── service.yaml
└── canary-playbook.yaml
```

stable-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-stable
spec:
  replicas: 9
  selector:
    matchLabels:
      app: myapp
      version: stable
  template:
    metadata:
      labels:
        app: myapp
        version: stable
    spec:
      containers:
        - name: myapp
          image: mydockerhub/myapp:v1
          ports:
            - containerPort: 80
```

canary-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-canary
spec:
  replicas: 1
  selector:
  matchLabels:
    app: myapp
    version: canary
  template:
    metadata:
      labels:
        app: myapp
        version: canary
    spec:
      containers:
        - name: myapp
          image: mydockerhub/myapp:v2
          ports:
            - containerPort: 80
```

canary_service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

Ansible Playbook - canary-playbook.yaml

```
- name: Canary Deployment on Kubernetes EKS
  hosts: localhost
  gather_facts: no
  tasks:

    - name: Deploy Stable version
      k8s:
        state: present
        definition: "{{ lookup('file', 'stable-deployment.yaml') }}"

    - name: Deploy Canary version
      k8s:
        state: present
        definition: "{{ lookup('file', 'canary-deployment.yaml') }}"

    - name: Deploy Service exposing both versions
      k8s:
        state: present
        definition: "{{ lookup('file', 'service.yaml') }}"

    - name: Wait for stable deployment rollout to complete
      k8s_info:
        kind: Deployment
        namespace: default
        name: myapp-stable
      register: stable_deploy

    - name: Wait until stable replicas ready
      k8s_info:
        kind: Pod
        namespace: default
        label_selectors:
```

```
      - "app=myapp,version=stable"
    register: stable_pods
    until: stable_pods.resources | selectattr('status.phase','equalto','Running') | list | length == 9
    retries: 10
    delay: 15

  - name: Wait until canary pods ready
    k8s_info:
      kind: Pod
      namespace: default
      label_selectors:
        - "app=myapp,version=canary"
    register: canary_pods
    until: canary_pods.resources | selectattr('status.phase','equalto','Running') | list | length == 1
    retries: 10
    delay: 15
```

Once real time testing complete by users then scaling Canary Up/Stable Down (Promoting Canary) using Ansible playbook

```
- name: Promote canary to stable by scaling
  hosts: localhost
  gather_facts: no

  tasks:
    - name: Scale stable deployment down
      k8s:
        kind: Deployment
        name: myapp-stable
        namespace: default
        replicas: 0

    - name: Scale canary deployment up
      k8s:
        kind: Deployment
        name: myapp-canary
        namespace: default
        replicas: 10

    - name: Update canary label to stable (optional)
      k8s:
        kind: Deployment
        name: myapp-canary
        namespace: default
        definition:
            spec:
                template:
                    metadata:
                        labels:
                            version: stable
```

11. How do you monitor a Kubernetes cluster?

==========

Use tools like Prometheus (metrics collection), Grafana (visualization).
Use ELK/EFK stacks (Elasticsearch, Fluentd/Fluent Bit, Kibana) for logs.
Kubernetes Dashboard or Lens IDE for cluster state.
Use probes (liveness/readiness) for pod health.

12. Explain how you would troubleshoot a failing pod?

==========

Check pod status:
kubectl get pods
Check error message and exit status of pod in pod logs:
kubectl logs <pod-name>
kubectl logs myapp-pod -c <container-name> ----------> If multiple containers
Inspect events for failures:
kubectl describe pod <pod-name>
kubectl get events --sort-by='.lastTimestamp'
Check container runtime status on the node (MemoryPressure and DiskPressure will be True)
kubectl describe node nodeName
Validate resource limits/requests and node capacity.
kubectl get pod <pod-name> -o yaml

ex→

```
resources:
   requests:
        cpu: "4"
        memory: "8Gi"
```

If your node only has 2 CPUs, this pod will remain in Pending state.
Check network connectivity and service endpoints.
Kubectl get svc
Kubectl get endpoints ----------> it will give ip and port use that port in blow command
Kubectl exec -it <pod-name> -- curl <service>:<port>

13. What is a ConfigMap and Secret? How do you use them?

==========

ConfigMap: Used to store non-sensitive configuration data as key-value pairs.
Secret: Used to store sensitive data like passwords, tokens, or keys, base64-encoded.
They can be consumed by Pods as environment variables or mounted as files.

Create configmap from yaml file

Vi testConfigMap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: my-config
data:
    APP_ENV: "production"
    APP_DEBUG: "false"
    DATABASE_URL: "mysql://db:3306"
```

Create configmap
kubectl apply -f testConfigMap.yaml

First create the encrypted data
echo –n 'admin' | base64 ----------> o/p → YWRtaW4
echo –n 'suman@#345' | base64 -----> o/p → MWETY345DAV45AK

Create Secret from yaml file
Vi mySecretFile.yaml

```
apiVersion: v1
kind: Secret
metadata:
    name: my-secret
type: Opaque
data:
    DB_PASSWORD: MWETY345DAV45AK
```

Create secret
kubectl apply -f mySecretFile.yaml

Use this ConfigMap in a Pod as environment variables

```
apiVersion: v1
kind: Pod
metadata:
    name: configmap-demo
spec:
    containers:
        - name: app
          image: myapp:latest
          envFrom:
              - configMapRef:
                    name: my-config
```

Use this Secret in a Pod as environment variables

```
apiVersion: v1
kind: Pod
metadata:
    name: secret-demo
spec:
    containers:
        - name: app
          image: myapp:latest
          env:
              - name: DB_PASSWORD
                valueFrom:
                    secretKeyRef:
                        name: my-secret
                        key: DB_PASSWORD
```

Create a property file

```
cat /etc/config/app.properties
    APP_ENV=production
    APP_DEBUG=false
    DATABASE_URL=mysql://db:3306
```

Create a Configmap yaml Vi my_configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
    name: app-config
data:
    app.properties:
        APP_ENV=production
        APP_DEBUG=false
        DATABASE_URL=mysql://db:3306
```

Step-1

Or Create a Configmap from the command

```
Kubectl create configmap app-config --from-file=/etc/config/app.properties
```

Define the Pod that mounts the ConfigMap as a volume

```
apiVersion: v1
kind: Pod
metadata:
    name: configmap-volume-pod
spec:
    containers:
        - name: app-container
          image: busybox
          command: [ "sleep", "3600" ]
          volumeMounts:
              - name: config-volume
                mountPath: /etc/config  # Files from ConfigMap(app.properties) is available here
    volumes:
        - name: config-volume
          configMap:
              name: app-config          # Refers to the ConfigMap created earlier
```

Step-2

First create the encrypted data

```
echo –n 'admin' | base64 ----------> o/p → YWRtaW4
echo –n 'suman@#345' | base64 -----> o/p → MWETY345DAV45AK
```

Now create the manifest file

Vi mySecrete.yaml

Step 1

```
apiVersion: v1
kind: Secret
metadata:
    name: mysecret
    type: Opaque
data:
    username: YWRtaW4     → Here username is the key and encrypted data is value
    password: MWETY345DAV48AK   →   password is key and encrypted data is value
```

create pod manifest file to use secret

```
apiVersion: v1
kind: Pod
metadata:
    name: secret-volume-pod
spec:
    containers:
        - name: secret-container
          image: busybox
          command: [ "sleep", "3600" ]
          volumeMounts:
              - name: secret-volume
                mountPath: "/etc/secret-data"
                readOnly: true
    volumes:
        - name: secret-volume
          secret:
              secretName: my-secret
```

Step 2

```
Kubectl create -f mySecretPod.yaml
Kubectl exec secret-volume-pod ls /etc/secret-data → two files username , password will be there
```

14. What is Helm and why would you use it?

==========

Helm is a package manager for Kubernetes, simplifying app deployment and management.
It uses Charts to define, install, and upgrade Kubernetes applications.
It helps manage complex apps and their dependencies declaratively.

15. Deploy NGINX App with Helm?

==========

Step 1 → Create the folder structure / Skeleton for Helm Chart

helm create my-nginx-test → it will create below folder structure

```
my-nginx-test/
    Chart.yaml      → Defines metadata about the chart (name, version, hart dependencies)
    values.yaml     → it is use to pass label to yaml
    templates/
        deployment.yaml
        service.yaml
        _helpers.tpl
```

Step 2 → create values.yaml

```
replicaCount: 2
image:
    repository: nginx
    tag: latest
    pullPolicy: IfNotPresent
service:
    type: ClusterIP
    port: 80
resources: {}
nodeSelector: {}
tolerations: []
affinity: {}
```

```
my-nginx/                  # Github repo
├── .github/
├── helm/
│   └── my-nginx-test/          # Helm chart folder
│       ├── Chart.yaml          # Chart metadata
│       ├── values.yaml         # Default configuration values
│       ├── templates/          # Kubernetes YAML templates
│       │   ├── deployment.yaml
│       │   ├── service.yaml
│       │   └── _helpers.tpl
│       └── README.md      # Chart-specific README
├── nginx.conf             # NGINX configuration file
├── Dockerfile             # To build custom NGINX image (optional)
├── README.md              # Project overview and instructions
├── LICENSE                # License file (e.g., MIT)
└── .dockerignore          # Ignore files for Docker build
```

Step 3 → _helpers.tpl is the file use to store reusable template helpers

```
{{/* Generate chart name */}}
{{- define "my-nginx-test.name" -}}
{{- .Chart.Name -}}
{{- end }}

{{/* Generate full resource name */}}
{{- define "my-nginx-test.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" -}}
{{- end }}

{{/* Common labels for resources */}}
{{- define "my-nginx-test.labels" -}}
app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
app.kubernetes.io/version: {{ .Chart.AppVersion }}
helm.sh/chart: {{ .Chart.Name }}-{{ .Chart.Version }}
{{- end }}
```

Step 4 → deployment.yaml which manages pods, specifying things like replicas, container images, ports, and labels

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: {{ include "my-nginx-test.fullname" . }}
    labels:
        {{ include "my-nginx-test.labels" . | indent 4 }}
spec:
    replicas: {{ .Values.replicaCount }}
    selector:
        matchLabels:
            app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
            app.kubernetes.io/instance: {{ .Release.Name }}
    template:
        metadata:
            labels:
                app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
                app.kubernetes.io/instance: {{ .Release.Name }}
        spec:
            containers:
                - name: nginx
                  image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
                  imagePullPolicy: {{ .Values.image.pullPolicy }}
                  ports:
                      - containerPort: {{ .Values.service.port }}
                  livenessProbe:
                      httpGet:
                          path: /
                          port: {{ .Values.service.port }}
                  readinessProbe:
                      httpGet:
                          path: /
                          port: {{ .Values.service.port }}
                  resources:
                      {{- toYaml .Values.resources | nindent 12 }}
```

Step 5 → service.yaml exposes your pods to other services or the outside world.

```
apiVersion: v1
kind: Service
metadata:
    name: {{ include "my-nginx-test.fullname" . }}
    labels:
        app: {{ include "my-nginx-test.name" . }}
spec:
    type: {{ .Values.service.type }}
    ports:
        - port: {{ .Values.service.port }}
          targetPort: {{ .Values.service.port }}
          protocol: TCP
          name: http
    selector:
        app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
```

Step 6 → create Chart.yaml

```
apiVersion: v2          # Helm chart API version (v2 is for Helm 3+)
name: my-nginx-test     # Name of the chart
description: A simple NGINX web server
type: application       # Can be "application" or "library"
version: 1.0.0          # Chart version (used for Helm chart packages)
appVersion: "1.21.6"    # Version of app being deployed (e.g. NGINX version)
keywords:
    - nginx
    - web
    - http
maintainers:
    - name: sumanta
      email: sk@example.com
home: https://nginx.org
sources:
    - https://github.com/example/my-nginx
```

Step 7 → Install Your Chart **helm install nginx-app ./my-nginx-test-proj**
Step 8 → update replicaCount value to 3 in values.yaml we can upgrade using helm
helm upgrade nginx-app ./ my-nginx-test-proj

16. Explain Horizontal Pod Autoscaling (HPA).
==========HPA automatically scales the number of pods based on CPU/memory usage or custom metrics.
It queries the Metrics Server for resource usage.
Configuration involves defining min/max pod count and target utilization.
Example

Step 1: Deploy NGINX using deployment manifest file

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            requests:
              cpu: 100m
            limits:
              cpu: 500m
```

Step 2 : create the HPA (Horizontal Pod Autoscaling) manifest file

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment    <----------it look for deployment manifest file
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

kubectl apply -f nginx-deployment.yaml
kubectl apply -f nginx-hpa.yaml
kubectl get hpa nginx-hpa → Check HPA status

Generate load on NGINX to increase CPU usage (for example using kubectl exec to run a stress command inside a pod or an external load generator). The HPA will increase pod count automatically.

1. Scenario: You want to deploy an application with multiple replicas. How do you scale your Kubernetes deployment?

Answer: You can scale a Kubernetes deployment by using the kubectl scale command or by modifying the deployment's YAML file.

Using kubectl command: kubectl scale deployment <deployment-name> --replicas=<number-of-replicas>

For example: kubectl scale deployment my-app --replicas=5

Alternatively, you can edit the YAML file of the deployment and change the replicas field:

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: my-app
spec:
   replicas: 5
   selector:
       matchLabels:
          app: my-app
```

Apply the changes: kubectl apply –f deployment.yaml

2. Scenario: You have a Kubernetes pod that is stuck in CrashLoopBackOff . How do you diagnose and fix it?

Answer:

1. Check pod logs to understand why it's crashing: kubectl logs <pod-name> --previous

2. Describe the pod to get detailed information about its state and events: kubectl describe pod <pod-name>

3. Look for any error messages or failed container states in the logs. Common issues include misconfigurations, missing files, or incorrect image versions.

4. Fix the issue based on the logs (e.g., fix environment variables, update the image, or modify the configuration) and then restart the pod.

To restart the pod manually: kubectl delete pod <pod-name>

3. Scenario: You need to expose a service in your Kubernetes cluster to the internet. How do you expose the service using a LoadBalancer?

Answer: To expose a service using a LoadBalancer , you need to define a service of type LoadBalancer in your YAML configuration.

Example Service definition:

```
apiVersion: v1
kind: Service
metadata:
    name: my-service
spec:
    selector:
        app: my-app
    ports:
        - protocol: TCP
          port: 80
          targetPort: 8080 ----------->  we can access the application using this port
      type: LoadBalancer
```

Apply the YAML: kubectl apply -f service.yaml

Once the service is created, Kubernetes will provision a cloud load balancer (if supported by the cloud provider) and assign an external IP address to access the service.

4. Scenario: You want to run a job in Kubernetes that executes once and then completes. How do you create a job in Kubernetes?

Answer: To run a job once we need to create a kubernetes job manifest file with restartPolicy: Never

```
apiVersion: batch/v1
kind: Job
metadata:
    name: my-job
spec:
    template:
        spec:
            containers:
            - name: my-container
```

```
      image: busybox
      command: ["echo", "Hello, World!"]
      restartPolicy: Never
```

This job will run the echo command once and then exit.
Apply the job: `kubectl apply -f job.yaml` **and You can check the status of the job using:** `kubectl get jobs`
To view the job's logs: `kubectl logs job/my-job`

5. Scenario: You need to restrict access to a Kubernetes service based on labels. How do you achieve this?
Answer: You can use Network Policy to restrict traffic based on pod labels. A network policy controls the communication between pods based on labels and namespaces.
Example of a NetworkPolicy that allows traffic only to pods with the label role=db in the same namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
   name: allow-db
spec:
  podSelector:
  matchLabels:
     role: db
  ingress:
  - from:
     - podSelector:
        matchLabels:
          role: app
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: myapp
spec:
  podSelector:
    matchLabels:
    role: backend
  ingress:
  - from:
   - podSelector:
      matchLabels:
        role: frontend
```

when we apply this manifest file `kubectl apply -f networkpolicy.yaml` **the NetworkPolicy allow only pods with the role=app label to communicate with pods labeled role=db .**

6. Scenario: You want to upgrade a deployment to a new version. How do you perform a rolling update in Kubernetes? / How you can upgrade a pod without downtime?
Answer: Kubernetes supports rolling updates by default so simply update the deployment's manifest file (with container image / replicas etc) and apply
1. In this example update the image version in the deployment YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
   name: my-app
spec:
  replicas: 3
  selector:
     matchLabels:
       app: my-app
template:
  metadata:
     labels:
       app: my-app
  spec:
     containers:
     - name: my-app
       image: my-app:v2 # New version of the image
       ports:
       - containerPort: 80
```

2. Now apply the deployment manifest file `kubectl apply -f deployment.yaml` **Once applied kubernetes will automatically perform a rolling update by replacing the old pods with the new ones without downtime.**

7. Scenario: You want to increase the memory limit of a pod. How do you modify the pod's resource limits?
Answer: To modify the resource limits (CPU/memory) for a pod, you need to update the resource requests and limits in the pod's YAML definition.

```
apiVersion: v1
kind: Pod
metadata:
    name: my-pod
spec:
   containers:
   - name: my-container
     image: nginx
     resources:
        requests:
           memory: "64Mi"              This is the min memory and cpu
           cpu: "250m"
        limits:
           memory: "128Mi"             This is the max memory and cpu
           cpu: "500m"
```

Now you can apply the manifest file kubectl apply -f pod.yaml or you can delete the pod or trigger a deployment update (kubectl rollout restart deployment your-deployment-name --: rolling update) to ensure the pod is restarted with the new limits.

8. Scenario: You need to monitor the health of a pod. How do you configure a liveness and readiness probe?
Answer: Liveness and readiness probes are configured within the pod's container specification to check the health of the application running inside the container.

```
apiVersion: v1
kind: Pod
metadata:
    name: my-pod
spec:
    containers:
    - name: my-container
      image: nginx
      livenessProbe:                Liveness Probe checks if the container is still running. If it fails, the
        httpGet:                                container will be restarted.
            path: /healthz
            port: 80
        initialDelaySeconds: 5
        periodSeconds: 10
      readinessProbe:               Readiness Probe checks if the container is ready to handle traffic. If it
        httpGet:                            fails, Kubernetes will stop routing traffic to that pod.
            path: /readiness                so the pod is removed from the service endpoints.

            port: 80
        initialDelaySeconds: 5
        periodSeconds: 10
      StartupProbe:                 Checks if the app has started successfully.
        httpGet:
            path: /
            port:80
        initialDelaySeconds: 5
        periodSeconds: 10
        failureThreshold: 30        recommended for slow startup
```

9. Scenario: You want to create a persistent volume for a pod to store data. How do you configure a persistent volume and claim in Kubernetes?

Answer:

1. Create a PersistentVolume (PV):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /mnt/data
```

Type of Reclaim Policy	What Happens When PVC Is Deleted?	Manual Action Needed?	Use Case
Retain	PV and storage are preserved	Yes	Backup, auditing, reuse
Delete	PV and storage are deleted automatically	No	Ephemeral or temp storage
Recycle	Data wiped and PV reused (deprecated)	No	Legacy/simple use cases

2. Create a PersistentVolumeClaim (PVC) to request the storage:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

STEP 2

3. Use the PVC in a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    volumeMounts:
    - mountPath: /mnt/data
      name: my-storage
  volumes:
  - name: my-storage
    persistentVolumeClaim:
      claimName: my-pvc
```

STEP 3

This configuration ensures that the pod has access to persistent storage.

10. Scenario: You want to limit the CPU and memory usage for a container. How do you set resource requests and limits?

Answer: In the pod or container spec, you can define resource requests (minimum resources) and limits (maximum resources) for CPU and memory.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
```

Requests: Kubernetes uses these values to schedule the pod (ensures the pod gets enough resources). Limits: Kubernetes enforces these limits (pod will be killed if it exceeds them).

11. Scenario: You have a Kubernetes cluster with multiple worker nodes. One of the nodes becomes unresponsive and needs to be replaced. Explain the steps you would take to replace the node without affecting the availability of applications running on the cluster.

Answer:

To replace the unresponsive node without affecting application availability, I would follow these steps:

1. Drain the unresponsive node using kubectl drain <node_name> --force --ignore-demonset –delete-emptydir-data This ensures that the pods are rescheduled on other healthy nodes.

2. Cordon the unresponsive node using kubectl cordon <node_name> to mark the node as unschedulable. This prevents new pods from being scheduled on the node while it's being replaced.

3. Once all the pods are safely rescheduled remove the unresponsive node using kubectl delete node <node_name> , Then you either by repairing it by configuring kubelet service properly or provisioning a new node.

4. Once the new node is ready uncordon the node , use kubectl uncordon <node_name> This allows new pods to be scheduled on the replacement node.

12. Scenario: What is StatefulSets? You have a stateful application running on Kubernetes that requires persistent storage. How would you ensure that the data is retained when the pods are rescheduled or updated? Deploy a stateful application and ensure persistence across restarts?

Answer:

1. Use StatefulSets to manage the stateful application and ensure each pod has a unique identity and stable network and storage even during rescheduling or updates.

2. Create a Persistent Volume that represents the storage resource (e.g., a networkattached disk) and then create a Persistent Volume Claim to binds the PV. This ensures that the same volume is attached to the pod when it's rescheduled or updated.

3. Define Storage to provision Persistent Volumes based on predefined storage requirements. This allows for automated volume provisioning when new PVCs are created.

By leveraging these features, the stateful application can maintain its data even when pods are rescheduled, updated, or scaled up/down.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx-statefulset
spec:
  serviceName: "nginx"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
       - name: nginx
         image: nginx:1.25
         ports:
         - containerPort: 80
           name: web
         volumeMounts:
         - name: www
           mountPath: /usr/share/nginx/html
  volumeClaimTemplates:          ----------  used to create (PVCs) automatically for each replica (pod)
  - metadata:
      name: www
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Service for Statefulset

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  clusterIP: None  # 👈 Headless service
  selector:
    app: nginx
  ports:
  - name: web
    port: 80
    targetPort: 80
```

storageClassName: standard

13. Scenario: A Kubernetes pod is stuck in a "Pending" state. What could be the possible reasons, and how would you troubleshoot it?
Answer:
Possible reasons for a pod being stuck in the "Pending" state could include:
1. Insufficient resources: Check if the cluster has enough resources (CPU, memory, storage) to accommodate the pod. You can use the kubectl describe pod <podname> command to view detailed information about the pod, including any resource-related issues.
2. Unschedulable nodes: Use kubectl get nodes to check if all the nodes in the cluster are in the "Ready" state and can schedule the pod.
3. Pod scheduling constraints: Verify if the pod has any scheduling constraints or affinity/anti-affinity rules that are preventing it from being scheduled. Check the pod's YAML kubectl get pod pod_name –o yaml or manifest file.
4. Persistent Volume (PV) availability: If the pod requires a Persistent Volume, ensure that the required storage is available and accessible. kubectl describe pod <pod-name> -n <namespace> (status could be FailedMount / VolumeNotFound / timeout waiting for volume / PVC not bound)
5. Network-related issues: Check if there are any network restrictions or misconfigurations preventing the pod from being scheduled or communicating with other resources.
Using kubectl describe pod <pod-name> -n <namespace> you can see events for pod scheduling issue (failed to assign the pod to a node / node(s) didn't match node selector / node(s) had taints that the pod didn't tolerate / network plugin is not ready) for Pod Network Accessibility go into the pod kubectl exec -it <pod-name> -n <namespace> -- sh and ping the service

14. Scenario: You have a Kubernetes Deployment with multiple replicas, and some pods are failing health checks. How would you identify the root cause and fix it?
Answer:
To identify the root cause and fix failing health checks for pods in a Kubernetes Deployment:
1. Check the pod's logs: Use the kubectl logs <pod-name> command to retrieve the logs of the failing pod. Inspect the logs for any error messages or exceptions that could indicate the cause of the failure.
2. Verify health check configurations: Examine the readiness and liveness probe configurations in the Deployment's YAML or manifest file. Ensure that the endpoints being probed are correct, the expected response is received, and the success criteria are appropriately defined. kubectl get pod <pod-name> -n <namespace> -o yaml check liveness and readiness prob
3. Debug container startup: If the pods are failing to start, check the container's startup commands, entrypoints, or initialization processes. Use the kubectl describe pod <pod-name> command to get detailed information about the pod, including any container-related errors. Run the command kubectl get deployment <deployment-name> -n <namespace> -o yaml or kubectl get pod <pod-name> -n <namespace> -o yaml and check

```
containers:
- name: <container-name>
  image: <image-name>
  command: ["./start.sh"]      # <-- This overrides ENTRYPOINT
  args: ["--config", "prod"]   # <-- These are passed to the command
```

4. Resource constraints: Inspect the resource requests and limits for the pods. It's possible that the pods are exceeding the allocated resources, causing failures. Adjust the resource specifications as necessary.
5. Image issues: Verify that the Docker image being used is correct and accessible. Ensure that the image's version, registry, and repository details are accurate.
6. Rollout issues: If the pods were recently deployed or updated, ensure that the rollout process completed successfully. Check the deployment's status using kubectl rollout status <deployment-name> and examine any rollout history with kubectl rollout history <deployment-name> . When you run kubectl rollout history <deployment-name> it will show image version , env variables and affinity rules.

15. Scenario: You need to scale a Kubernetes Deployment manually. How would you accomplish this?
Answer:
To manually scale a Kubernetes Deployment:
Use the kubectl scale command: Run kubectl scale deployment/<deployment-name> --replicas=<number-of-replicas> to scale the deployment.
Alternatively: Modify the replicas field in the Deployment's YAML or manifest file to the desired number of replicas. Then, apply the changes using kubectl apply -f <path-to-deployment-yaml> .

16. How you connect pod to node?

Answer:

Using nodeName , nodeSelector and nodeAffinity to connect pod to the node.

You can use the nodeName field when you need to run a debug pod on a specific node to inspect node-specific logs. Not recommended for production.

The nodeSelector selects nodes by matching labels—for example, scheduling a databse pod on nodes labeled for SSD storage.

The nodeAffinity provides flexibility with priorities and is preferred when scheduling pods in specific zones or regions or GPU , but allow fallback using "preferredDuringSchedulingIgnoredDuringExecution" soft constraints or "requiredDuringSchedulingIgnoredDuringExecution" hard constraints.

User of nodeAffinity

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gpu-app-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gpu-app
  template:
    metadata:
      labels:
        app: gpu-app
    spec:
      containers:
        - name: gpu-container
          image: nvidia/cuda:12.0-base
          ports:
            - containerPort: 8080
              name: http
          resources:
            limits:
              nvidia.com/gpu: 1
          volumeMounts:
            - name: gpu-logs
              mountPath: /var/log/gpu
      volumes:
        - name: gpu-logs
          hostPath:
            path: /var/log/gpu
            type: DirectoryOrCreate
      affinity:
        nodeAffinity:
          # Hard constraint: must run on GPU node
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: hardware-type
                    operator: In
                    values:
                      - gpu
          # Soft constraint: prefer zone us-central1-a
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 1
              preference:
                matchExpressions:
```

Use of nodeName , nodeSelector

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx-deployment
spec:
replicas: 1
selector:
    matchLabels:
    app: nginx
template:
    metadata:
    labels:
        app: nginx
    spec:
    nodeName: py_node-1   # Actual node name
    nodeSelector:
        disktype: ssd     # Actual node label
        env: stage
    containers:
    - name: nginx
        image: nginx:1.25
        ports:
        - containerPort: 80
```

```
- key: topology.kubernetes.io/zone
  operator: In
  values:
        - us-west-a
```

17. What is the difference between nodeAffinity and podAffininty , podAntiAffinity?

The nodeAffinity is used to schedule a Pod onto a specific node based on labels on nodes.
The podAffinity used to schedule a Pod on a node based on the presence of other Pods with certain labels which is already running on that node.
Pod Anti-Affinity: Ensures pods are placed on different nodes (e.g., for high availability)

```
nodeAffinity
=============
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype       --> Pod will only be scheduled on nodes
            operator: In            where nodes labeled with  disktype=ssd
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
```

```
podAffinity
==================
apiVersion: v1
kind: Pod
metadata:
name: pod-with-pod-affinity
spec:
affinity:
    podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app          --> The frontend Pod will be scheduled
          operator: In            on the same node where the
          values:               Pod labeled "app=backend" is running
          - backend
        topologyKey: "kubernetes.io/hostname"
containers:
- name: frontend
    image: nginx
```

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: app
              operator: In
              values:
                - backend
        topologyKey: "kubernetes.io/hostname"
```

18. How to run a container as non-root user?
when deploying a container in Kubernetes with a securityContext that includes runAsNonRoot: true and a specific runAsUser (e.g., UID 1041 in K8), the container must have a user with that UID defined.
Example

```
FROM node:18-alpine
        # Create a non-root user and group
RUN addgroup -S appgroup &&
        adduser -S appuser  -D -u 101  -G appgroup
        # Set permissions (optional but recommended)
WORKDIR /app
COPY . /app
RUN chown -R appuser:appgroup /app
        # Switch to the non-root user
USER appuser
CMD ["node", "server.js"]
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-non-root
spec:
  containers:
  - name: nginx
    image: nginx:1.25-alpine
    securityContext:
      runAsNonRoot: true
      runAsUser: 101  # nginx user in official image
```

19. You have a secret file in kubernetes, how you pass the value from the secret file to container?
Answer:
Create secret from file or literal using

From file → kubectl create secret my_secret_file --from-file =username=./username.txt
Once the secret file is created, the secret needs to be mounted. When we access the mounted file inside the container, we will get the value of the secret.
From literal → kubectl create secret generic myenvsecret --from-literal=pasword=123456abc
Once the secret is created use the secret as an env variable. Now the env variable will be available in the container.

**EX → kubectl create secret generic my-secret **
--from-literal=db_password='S3cr3tP@ssw0rd'

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: secret-example
spec:
    replicas: 1
    selector:
        matchLabels:
          app: secret-app
template:
    metadata:
        labels:
            app: secret-app
    spec:
        containers:
        - name: myapp
          image: python:3.11-slim
          command: ["python"]
          args: ["app.py"]
          env:
          - name: DB_PASSWORD
            valueFrom:
                secretKeyRef:
                    name: my-secret
                    key: db_password
```

```
In python  # app.py
----------
import oss
db_password = os.getenv("DB_PASSWORD")
print("DB_PASSWORD from environment:", db_password)
```

```
dockerfile
----------
FROM python:3.11-slim
COPY app.py .
CMD ["python", "app.py"]

docker build -t
my-registry/secret-app:latest .
```

```
kubectl create secret generic mysecret
    --from-file=username=./username.txt
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  containers:
  - name: mycontainer
    image: myimage
    volumeMounts:
    - name: secret-volume
      mountPath: "/etc/secret-volume"
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: mysecret
```

Accessing secret from secret file to container

20. What is a ConfigMap and Secret? How do you use them in deployment and access the value in container?
ConfigMap: Used to store non-sensitive configuration data as key-value pairs.
Secret: Used to store sensitive data like passwords, tokens, or keys, base64-encoded.
They can be consumed by Pods as environment variables or mounted as files , So that it will be available for the container.

Access ConfigMap and Secret from environment variables

```
spec:
  containers:
  - name: demo-container
    image: nginx
    env:
      - name: CONFIG_VAR
        valueFrom:
          configMapKeyRef:
              name: myconfigmap
              key: mykey
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
              name: my-secret
              key: db_password
```

Access ConfigMap and Secret from mounted files

```
spec:
    containers:
    - name: my-container
      image: nginx
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
      - name: secret-volume
        mountPath: /etc/secret
    volumes:
    - name: config-volume
      configMap:
        name: my-configmap
    - name: secret-volume
      secret:
        secretName: my-secret
```

21. What is Helm and why would you use it?
Answer:

Helm is a package manager for Kubernetes, simplifying app deployment and management.
It uses Charts to define, install, and upgrade Kubernetes applications.
It helps manage complex apps and their dependencies declaratively.

22. Deploy NGINX App with Helm?

Step 1 → Create the folder structure / Skeleton for Helm Chart

helm create my-nginx-test → it will create below folder structure

my-nginx-test/
- **Chart.yaml → Defines metadata about the chart (name, version, hart dependencies)**
- **values.yaml → it is use to pass label, image name, replica count, ports to yaml**
- **templates/ value in the variable can override at install/upgrade time (--set or -f custom-values.yaml)**
  - **deployment.yaml**
  - **service.yaml**
  - **_helpers.tpl → it contain reusable helper functions for naming conventions, labels, annotations**
    So that other templates can call these to avoid duplication

Step 2 → create values.yaml

```
replicaCount: 2
image:
  repository: nginx
  tag: latest
  pullPolicy: IfNotPresent
service:
  type: ClusterIP
  port: 80
resources: {}
nodeSelector: {}
tolerations: []
affinity: {}
```

```
my-nginx/                          # Github repo
├── .github/
├── helm/
│   └── my-nginx-test/             # Helm chart folder
│       ├── Chart.yaml             # Chart metadata
│       ├── values.yaml            # Default configuration values
│       ├── templates/             # Kubernetes YAML templates
│       │   ├── deployment.yaml
│       │   ├── service.yaml
│       │   └── _helpers.tpl
│       └── README.md              # Chart-specific README
├── nginx.conf                     # NGINX configuration file
├── Dockerfile                     # To build custom NGINX image (optional)
├── README.md                      # Project overview and instructions
├── LICENSE                        # License file (e.g., MIT)
└── .dockerignore                  # Ignore files for Docker build
```

Step 3 → _helpers.tpl is the file use to store reusable template helpers

```
{{/* Generate chart name */}}
{{- define "my-nginx-test.name" -}}
{{- .Chart.Name -}}
{{- end }}

{{/* Generate full resource name */}}
{{- define "my-nginx-test.fullname" -}}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" -}}
{{- end }}

{{/* Common labels for resources */}}
{{- define "my-nginx-test.labels" -}}
app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
app.kubernetes.io/version: {{ .Chart.AppVersion }}
helm.sh/chart: {{ .Chart.Name }}-{{ .Chart.Version }}
{{- end }}
```

Step 4 → deployment.yaml which manages pods, specifying things like replicas, container images, ports, and labels

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: {{ include "my-nginx-test.fullname" . }}
  labels:
    {{ include "my-nginx-test.labels" . | indent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
      app.kubernetes.io/instance: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
        app.kubernetes.io/instance: {{ .Release.Name }}
    spec:
      containers:
        - name: nginx
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - containerPort: {{ .Values.service.port }}
          livenessProbe:
            httpGet:
              path: /
              port: {{ .Values.service.port }}
          readinessProbe:
            httpGet:
              path: /
              port: {{ .Values.service.port }}
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
```

Step 5 → service.yaml exposes your pods to other services or the outside world.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "my-nginx-test.fullname" . }}
  labels:
    app: {{ include "my-nginx-test.name" . }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.port }}
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: {{ include "my-nginx-test.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

Step 6 → create Chart.yaml

```
apiVersion: v2                  # Helm chart API version (v2 is for Helm 3+)
name: my-nginx-test             # Name of the chart
description: A simple NGINX web server
type: application               # Can be "application" or "library"
```

```
version: 1.0.0                  # Chart version (used for Helm chart packages)
appVersion: "1.21.6"            # Version of app being deployed (e.g. NGINX version)

keywords:                       # Using these keywords helm chart able to search the repository
 - nginx                                                ex→ helm  search  repo  http
 - web
 - http

maintainers:
 - name: sumanta
   email: sk@example.com

sources:
 - https://github.com/example/my-nginx
   home: https://nginx.org
```

Step 7 → Install Your Chart (Use helm install to Installs Kubernetes resources when deploying your app for the first time)

```
helm install nginx-app  ./my-nginx-test-proj
```

Step 8 → update replicaCount value to 3 in values.yaml we can upgrade using helm / when you want to update an existing deployment with image versions, replicas, labels, etc. we can use helm upgrade

```
helm upgrade   nginx-app ./ my-nginx-test-proj

helm upgrade nginx-app ./my-nginx-test-proj  --set image.tag=1.27   image version will upgrade by overriding  variable value
```

Step 9 → when you want to perform install If the release doesn't exist else upgrade the release if release exist

```
helm  upgrade  –install  nginx-app  ./my-nginx-test-proj
```

23. How to create a cluster from screatch using kubernetes ?

```
k8s/
├── base/                       # Common, reusable Kubernetes manifests for all environments
│   ├── namespace.yaml          # Defines Kubernetes namespaces for app isolation
│   ├── configmap.yaml          # Application configuration (non-sensitive key-value pairs)
│   ├── secret.yaml             # Encrypted secrets (e.g., DB credentials, API keys)
│   ├── deployment.yaml         # Core application Deployment (replicas, containers, etc.)
│   ├── service.yaml            # Service to expose the deployment internally in the cluster
│   ├── hpa.yaml                # Horizontal Pod Autoscaler configuration (for auto-scaling)
│   └── ingress.yaml            # Ingress resource to expose the app externally via domain
├── overlays/                   # Environment-specific configuration overrides
│   ├── dev/                    # Development environment configs
│   │   ├── kustomization.yaml  # Kustomize file to apply overlays for dev
│   │   └── patch-deployment.yaml  # Patches like replica count, image tag, etc.
│   ├── staging/                # Staging (pre-prod) environment configs
│   │   ├── kustomization.yaml
│   │   └── patch-deployment.yaml
│   └── prod/                   # Production environment configs
│       ├── kustomization.yaml
│       └── patch-deployment.yaml
├── ingress-controller/         # Ingress controller setup (e.g., NGINX, ALB)
│   └── ingress-controller.yaml # Deployment and Service for ingress controller
├── monitoring/                 # Monitoring stack (Prometheus, Grafana, etc.)
│   └── prometheus-stack.yaml   # Prometheus/Grafana stack via Helm or YAML
├── cert-manager/               # TLS automation
│   ├── issuer.yaml             # Let's Encrypt ClusterIssuer or Issuer
│   └── certificate.yaml        # TLS certificate resource for the domain
└── README.md                   # Documentation on how to deploy and manage the cluster
```

24. How Kubernetes resource manages external access to services within a cluster, acting as a single entry point for incoming traffic and routing it to the appropriate services?
Answer:
Ingress: Controls external traffic coming into your Kubernetes cluster and routing it to the right service.

25. You have two applications in different namespaces. How would you enable secure communication between them?
Answer:
Use NetworkPolicy to allow traffic explicitly from the pod in one namespace to the other. The policies check to the match labels and namespace selectors to restrict permitted traffic.

```
Apiversion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-app1-to-app2
  namespace: namespace-2
spec:
  podSelector:
    matchLabels:
      app: app2
  ingress:            -------> is a load balancer which provide a control in an efficient way
    - from:                                    to expose applications to the outside world
        - podSelector:
            matchLabels:
              app: app1
          namespaceSelector:
            matchLabels:
              name: namespace-1
  policyTypes: [Ingress]
```

26. What are the Service Types & Traffic Flow in k8s?
ClusterIP ------> ClusterIP is the default communication process in k8s. It is use for internal communication between services.
Ex -: You have two microservices: auth-service and user-service. auth-service needs to call user-service internally by using clusterIP.
NodePort ----> Exposes a service on a specific port on all node IPs
Ex-: You're running one application on Kubernetes on-prem, You want to access your web app from your browser. You don't have a cloud load balancer. Then you can use NodePort.
LoadBalancer -----> Provisions an external IP address from the cloud provider and routes traffic to NodePort service
Ex -: Your application is on cloud (EKS) and It needs a stable external IP for users worldwide in that case we need to use LoadBalancer.
Ingress ----------> Uses a single external IP or domain to route (HTTP) traffic to multiple services.
Ex -: You have three HTTP services: api-service, web-service, and admin-service. You want to expose them under one domain with different paths.In this case we need to use Ingress.