Jenkins

1) Jenkins installation on Cloud and Server (VM) ?

Ans: For **Linux (Ubuntu/CentOS)**:

- I update the system, install Java (typically OpenJDK 11), then download and install Jenkins using the official Jenkins repo.
- I manage Jenkins as a **systemd service**, enabling it to auto-start on reboot.
- Commands

```
sudo apt update
sudo apt install openjdk-11-jdk -y
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ >
/etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins -y
sudo systemctl start jenkins
```

Jenkins Installation on **Cloud**

- AWS EC2 instances (manually and using automation tools like Terraform or Ansible).
- The installation process is nearly the same as on local VMs, but I ensure:
- Proper **security group/firewall rules** to allow port `8080` access.
- Set up **EBS volume (AWS)** or **managed disks (Azure)** for persistent Jenkins home

I always configure a **separate volume** for `/var/lib/jenkins` to back up job configurations.

- Use **Nginx or Apache** as a reverse proxy for TLS (HTTPS) setup.
- Secure Jenkins with **LDAP/SAML integration**, secrets management (Vault/SSM), and proper RBAC.

2) How do you **upgrade Jenkins**? Explain the steps and strategies (on On-Prem & vm) ?

Ans: When upgrading Jenkins (whether on an on-prem server or a VM), I follow a structured approach to **minimize downtime and ensure rollback is possible if needed**:

Step- 1: Take a Full Backup

- Backup the **entire Jenkins home directory**: /var/lib/jenkins/
- It contains jobs, plugins, configurations, credentials, and user data.
- Tools like `rsync`, `tar`, or Jenkins backup plugins can be used.

- Commands

  sudo systemctl stop jenkins

  sudo tar -czvf jenkins_backup_$(date +%F).tar.gz /var/lib/jenkins

  sudo systemctl start jenkins

Step 2: Note Current Version

- Record the current Jenkins version using:
- java -jar jenkins.war --version

Step 3: Check Plugin Compatibility

- Go to **Manage Jenkins** → **Plugin Manager** → **Updates**.
- Ensure critical plugins are compatible with the target Jenkins version.
- Upgrade important plugins first, especially Pipeline-related ones

Step 4: Download the Latest Jenkins

- manually download the new `.war` file if running Jenkins standalone:
- wget https://get.jenkins.io/war-stable/latest/jenkins.war

Step 5: Replace WAR File (if using WAR-based Jenkins)

- commands

  sudo systemctl stop jenkins
  sudo cp jenkins.war /usr/share/jenkins/jenkins.war
  sudo systemctl start jenkins

Step 6: Post-Upgrade Validation

- Access Jenkins UI.
- Verify:
  - Plugins are working
  - Jobs are intact
  - Credentials and users are retained
- Check logs:
- tail -f /var/log/jenkins/jenkins.log

Step- 7: Blue-Green Upgrade (Zero Downtime – Advanced)

- Spin up a new VM with the upgraded Jenkins version.
- Restore data from backup.
- Test and validate on the new instance.
- If successful, switch DNS or load balancer to point to the new server.

Step-8: In Case of Rollback

If issues are found post-upgrade:

- Stop Jenkins.
    - Restore from the tar backup.
    - Revert to previous WAR or reinstall the old version using `apt/yum`.

3) How Do You Take Backups in Jenkins (Server or VM), and What's the Process Post Backup for Jenkins Upgrade?

- When Jenkins is running on a standalone server or VM (not on managed cloud services), I follow this manual + automated backup strategy:
- What Do I Backup?
- The entire Jenkins Home Directory:
- Usually located at `/var/lib/jenkins`, which includes:
    - Job configurations
    - Build history
    - Plugins
    - Credentials (encrypted)
    - User configs
    - Secrets (`secrets/`, `credentials.xml`)
- Configuration files like:
    - `/etc/default/jenkins` (for port and environment settings)

    - `/etc/init.d/jenkins` (service script, if applicable)

- Stop Jenkins to avoid data corruption:

  sudo systemctl stop jenkins

- Take compressed backup:

  sudo tar -czvf jenkins-backup-$(date +%F).tar.gz /var/lib/jenkins

- Optionally back up plugin list:

  ls /var/lib/jenkins/plugins > plugin-list.txt

- Start Jenkins:

  sudo systemctl start jenkins

I have also used plugins like **ThinBackup** or **SCM Sync Configuration**, but for more control and safety, I prefer **filesystem-level backups**.

4) **What is the Difference Between Declarative and Scripted Pipeline in Jenkins?**

Ans:

1. Declarative Pipeline:

- **Introduced later** to simplify pipeline creation.
- Follows a **structured and predefined syntax**.
- Starts with a `pipeline {}` block.
- Easier to read, maintain, and version control.
- Enforces best practices and **validates syntax during compilation**.

2. Scripted Pipeline:

- The original format of Jenkins pipelines.
- Based entirely on Groovy scripting.

- Offers more flexibility and programmatic control (loops, conditionals, complex logic).
- Written inside a `node {}` block.
- Suitable for complex workflows or dynamic steps.

In most projects, I use Declarative Pipelines for their readability and standardization. But for advanced use cases like dynamically generating stages or handling external scripts, I switch to Scripted Pipelines or combine both using `script {}` blocks inside declarative pipelines.

5) **What is a Multibranch Pipeline in Jenkins, When Do We Use It, and What Do We Achieve?**

Ans:

A Multibranch Pipeline in Jenkins is a special type of pipeline project that:

- Automatically discovers, creates, and manages pipelines for each branch in a source control repository (like GitHub, GitLab, Bitbucket).
- Each branch can have its own Jenkinsfile, and Jenkins automatically builds each branch based on that file.

It eliminates the need to manually create separate pipeline jobs for every feature or release branch.

We use multibranch pipelines in:

- **Projects with multiple active branches** (e.g., `dev`, `qa`, `staging`, `main`, `feature/*`)
- **CI/CD automation for every branch**, so each team or developer can test code independently.
- **Pull Request builds** (GitHub/Bitbucket integration).
- When following **GitFlow or trunk-based development** where branches are actively pushed.

In my projects, I used **multibranch pipelines** to automate builds for multiple feature and release branches.

Jenkins automatically scanned the Git repo, detected new branches, and triggered builds whenever code was pushed.

This helped us maintain **branch-wise CI**, ensure **quality gates on PRs**, and allowed **environment-specific deployments** based on branch names — all with minimal manual effort.

6) **What Are Shared Libraries in Jenkins and How Do We Use Them?**

Ans:

Shared Libraries in Jenkins are a way to **reuse common pipeline code** (like functions, stages, steps) across multiple Jenkinsfiles or pipeline projects.

- They are **custom Groovy libraries** stored in a **separate Git repository** or a folder structure within an SCM.
- They allow teams to follow **DRY (Don't Repeat Yourself)** principles and **standardize CI/CD pipelines**.

We use shared libraries when:

- Multiple Jenkins pipelines have **common logic** (e.g., build, test, deploy steps).
- We want to centralize and **version control** pipeline functions.
- To maintain **consistency** across teams and reduce duplication.

The typical directory structure of a shared library looks like:

```
(root)
 └── vars/
    └── deployApp.groovy      # Declarative-style entry points (functions)
 └── src/
    └── org/company/Utils.groovy   # Helper classes (Java-style)
 └── resources/
    └── templates/email.html       # Groovy templates, files
```

└── README.md

I use Jenkins Shared Libraries to build reusable and standardized pipeline components across multiple projects.
This helps maintain consistency, reduces code duplication, and speeds up onboarding for new teams.
I usually keep the shared library in a separate Git repo, configure it globally in Jenkins, and call it using `@Library()` annotation inside Jenkinsfiles.

7) **write a groovy script showing different stages of pipeline, it should be prod kind of pipeline.**

Ans: in VS CODE

Kubernetes

1) **What is authentication and Authorization and how it happens in k8s cluster, explain it step by step.**

Ans:

When a request is made to the Kubernetes API Server (e.g., `kubectl get pods`), the following steps are executed:

Step 1: Authentication – Who Are You?

The API Server first checks who is making the request by validating credentials such as:

- **X.509 client certificate** (used in kubeconfig)
- **Bearer Token** (used by service accounts or external OIDC)
- **Static token files**
- **OpenID Connect (OIDC)** – integrates with external identity providers like Azure AD, Okta, Google
- **Webhook token authentication**

📌 **If authentication fails**, the API server returns `401 Unauthorized`.

Step 2: Authorization – What Can You Do?

Once authenticated, the API server checks if the identity has permission to perform the requested action using:

- **RBAC (Role-Based Access Control)** → most widely used.
- **ABAC (Attribute-Based Access Control)** → legacy.
- **Webhook authorization** → custom external logic.

📌 The request is evaluated against:

- **Verb**: get, list, create, delete, update
- **Resource**: pods, deployments, secrets
- **Namespace**: where the resource resides
- **User identity** (User or ServiceAccount)

📌 **If authorization fails**, it returns `403 Forbidden`.

Step 3: Admission Controllers – Should We Allow It?

After the request is authenticated and authorized, admission controllers run to:

- Mutate the request (add labels, enforce policies)
- Validate request against custom rules (e.g., no privileged containers)

Examples:

- `NamespaceLifecycle`, `LimitRanger`, `PodSecurity`, `MutatingAdmissionWebhook`

📌 If the request fails here, it is rejected before resource creation.

Real-World Example:

When I run `kubectl get pods -n dev`:

1. The kubeconfig token or client cert is sent → Authentication
2. K8s checks if this user/service account has `get` access to `pods` in `dev` namespace → Authorization
3. If a Pod creation, admission controllers might mutate or validate the spec → Admission Control

In Kubernetes, authentication verifies who you are, authorization checks what you're allowed to do, and admission controllers enforce cluster policies.

I usually work with RBAC, and manage access via ServiceAccounts, Roles, and RoleBindings, ensuring least privilege principle and using tools like OPA Gatekeeper or Kyverno for policy enforcement.

2) Is it necessary to use openshift if we already have EKS and AKS.? difference between in details.

Ans:

No, it's not necessary to use OpenShift if you already use managed Kubernetes services like **EKS (AWS)** or **AKS (Azure)**.

OpenShift is an enterprise Kubernetes platform with additional features on top of upstream Kubernetes, while **EKS and AKS** are **vanilla Kubernetes** services managed by their respective cloud providers.

However, there are key **differences in features, flexibility, and use cases** that can help decide **when to use OpenShift**.

Feature	OpenShift (Red Hat)	EKS (AWS) / AKS (Azure)
Kubernetes Base	Uses Kubernetes + enhancements	Uses upstream vanilla Kubernetes
Managed By	Red Hat (or self-managed)	AWS (EKS), Azure (AKS)
Installation	Complex (unless using OpenShift Dedicated or ROSA)	Fully managed (1-click cluster creation)
Web Console (UI)	Built-in developer/admin console	Minimal or none (AKS has basic dashboard)

Integrated CI/CD	Has OpenShift Pipelines (Tekton-based)	No built-in CI/CD (you use GitHub Actions, CodePipeline, Azure DevOps)
Security / RBAC	Built-in **SecurityContextConstraints (SCC)**, stricter policies	Standard Kubernetes RBAC only
Container Registry	Comes with **integrated image registry**	You integrate ECR (EKS) or ACR (AKS)
Developer Tools	Source-to-Image (S2I), Developer Catalog, Templates	Not built-in; you need external tooling
Upgrades & Maintenance	Managed by Red Hat (ROSA/OpenShift Online) or self-managed	AWS or Azure manages upgrades
Cost	OpenShift license required (for self/ROSA)	Pay-as-you-go for compute and services only
Ecosystem Flexibility	More opinionated stack	More flexibility, bring your own tools
Support for Custom Workloads	Might be restricted by SCC/PodSecurity	More open and flexible for workload customization

OpenShift is **not necessary** if you're using EKS or AKS, but it's a **powerful enterprise platform** when you need **built-in CI/CD, tighter security controls**, and **pre-integrated developer tools**.

I choose between them based on project needs — if I need flexibility, I go with EKS/AKS; if I need security, governance, and full-stack enterprise support, I consider OpenShift.

3) Tell me about operators and use case, how do we manage operators in openshift?

Ans:

An Operator is a Kubernetes-native application controller that extends Kubernetes capabilities to manage complex stateful or custom applications.

It codifies operational knowledge (installing, configuring, upgrading, recovering) into a Kubernetes-native custom controller, using CRDs (Custom Resource Definitions).

How Operators Work (Simple Explanation):

- The operator defines a CRD like `PostgresCluster`.
- You apply a custom resource (YAML) that describes the desired state.
- The Operator watches that resource and performs all actions needed to bring it to life — install, scale, backup, restore, upgrade, etc.

OpenShift has a dedicated GUI and CLI support for managing Operators.

OperatorHub (GUI):

- **Navigate to:** *Operators* > *OperatorHub* (in OpenShift Console)
- Browse certified, community, and custom operators
- Click to **Install**, choose:
    - Target namespace or cluster-wide
    - Manual or automatic updates
- It creates CRDs + deploys Operator controller

OpenShift uses **OLM** to:

- Install, upgrade, and manage Operators
- Handle versioning and dependency resolution
- Auto-update operators (if configured)

Operators in OpenShift are Kubernetes-native controllers that manage complex applications automatically.

I use them to deploy tools like **PostgreSQL, Prometheus, ArgoCD**, etc., with custom configurations, scaling, and recovery logic.

In OpenShift, we manage operators via **OperatorHub (UI)** or the **`oc` CLI**, and **Operator Lifecycle Manager** ensures smooth updates and version control.

4) How our Kubernetes cluster authenticate with docker hub or other container registry, how will you manage password change for docker and manage secrets and make sure authentication does not fail, also maintaining creds as secure ?

ANS:

Why Authentication is Needed-

Kubernetes needs to pull images from private registries (e.g., Docker Hub, AWS ECR, Azure ACR).

When pulling from a private repository, Kubernetes must authenticate using valid credentials (username/password or token).

How Authentication Works with Docker Hub -

You create a Docker Registry Secret of type `kubernetes.io/dockerconfigjson`.

```
kubectl create secret docker-registry regcred \
  --docker-username=<your-docker-username> \
  --docker-password=<your-docker-password> \
  --docker-email=<your-email> \
  --namespace=your-app-namespace
```

This stores the credentials securely in Kubernetes.

- You attach the secret in your pod spec using `imagePullSecrets`.

- Now Kubernetes uses the `regcred` secret to authenticate with Docker Hub when pulling the image.

I automate Docker credential rotation using external secret stores like **AWS Secrets Manager or Vault**, combined with a shell script or Kubernetes CronJob that updates Kubernetes secrets across namespaces.

This ensures that when passwords change, the cluster always has the latest credentials securely — reducing downtime and manual intervention.

Automate Image Pull Secrets with ServiceAccounts

- Automate Image Pull Secrets with ServiceAccounts
- If multiple pods in a namespace need the same image pull secret: then we use serviceaccount for that to reference secret.

What If Docker Password Changes?

To pull private images, Kubernetes uses `docker-registry` type secrets which store the Docker Hub or container registry credentials.

I manage these using `kubectl create secret`, attach them using `imagePullSecrets`, and rotate credentials via automation when passwords change.

For security and scale, I prefer **external secret managers** (Vault, AWS Secrets Manager) and GitOps tools like **Sealed Secrets** to avoid exposing sensitive data in plaintext or Git.

7) What Are the Common Errors You Face During Deployment with Pods and Containers in Kubernetes?

ANS:

In real-world Kubernetes deployments, I've encountered **various pod and container-level errors**. These usually occur due to **misconfigurations, image issues, resource constraints, or missing dependencies**.

Below are **common categories of deployment errors**, their causes, how I debug them, and how I fix them.

1. ImagePullBackOff / ErrImagePull

💬 Description: Pod fails to pull the image from registry.

🔍 Causes:

- Incorrect image name or tag
- Image doesn't exist in the registry
- No access to private registry (auth missing)
- Wrong or missing `imagePullSecrets`

🛠️ Fixes:

- Check image path: `docker.io/org/app:tag`
- Verify `kubectl describe pod` → check event logs
- Recreate `imagePullSecrets`
- Ensure internet access in private clusters

2. CrashLoopBackOff

Description: Container starts → crashes → Kubernetes restarts it → repeat

🔍 Causes:

- App exits due to unhandled exceptions
- Bad environment variables or missing config
- Database not ready / service dependency down
- Wrong command or entrypoint in Dockerfile

🛠️ Fixes:

- Use `kubectl logs <pod>` to debug crash
- Run locally with `docker run` to test

- Add `livenessProbe` and `readinessProbe` properly
- Add `sleep` or retry logic in the app startup

3. ContainerCreating (Stuck Pod)

💬 **Description:** Pod is stuck in "ContainerCreating" status

🔍 **Causes:**

- Volume mount issues (PVC not bound)
- Image pull delay
- Network or CNI plugin not working
- Node resource pressure

🛠️ **Fixes:**

- Check with `kubectl describe pod`
- Check PV/PVC status using `kubectl get pvc`
- Ensure CNI plugin is healthy (`kube-flannel`, `calico`)
- Check node disk/memory with `kubectl describe node`

4. Pending Pod

💬 **Description:** Pod is stuck in "Pending" state

🔍 **Causes:**

- No available nodes match resource requests
- Node selector/taints mismatch

- Insufficient CPU or memory on nodes
- PVC not bound

🛠️ **Fixes:**

- Check with `kubectl describe pod`
- Validate taints, tolerations, node selectors
- Scale up nodes or reduce resource limits
- Ensure PVC is bound to a PV

5. OOMKilled (Out of Memory Killed)

💬 **Description:** Container exceeded memory limits and was killed by the system

🔍 **Causes:**

- Memory leak in app
- `resources.limits.memory` too low
- Heavy operations during startup

🛠️ **Fixes:**

- Review container memory usage: `kubectl top pod`
- Increase memory limit
- Optimize app memory usage

6. Readiness Probe / Liveness Probe Failed

💬 **Description:** Health check fails → container restarted or not added to service

🔍 **Causes:**

- Wrong path/port in `readinessProbe`
- App takes too long to start
- Probe response is not 200 OK

🛠️ **Fixes:**

- Tune `initialDelaySeconds`, `timeoutSeconds`
- Validate probe path manually (e.g., `curl localhost:8080/health`)
- Add logging in probe handler

7. VolumeMount Errors

💬 **Description:** Pod fails due to missing or misconfigured volume mounts

🔍 **Causes:**

- PVC not bound
- Wrong mount path
- Access denied or permission error on mounted directory

🛠️ **Fixes:**

- Check PVC status and storage class
- Ensure file permissions are correct
- Validate volume definitions in pod spec

8. Node Affinity / Taints Errors

💬 **Description:** Pod can't be scheduled to any node due to taints or affinity rules

🔍 **Causes:**

- Tolerations missing in pod spec
- Node affinity rules too restrictive

🛠️ **Fixes:**

- Add matching tolerations
- Review affinity rules and relax constraints if needed

9. DNS Resolution Failure

💬 **Description:** App fails to resolve internal service names

🔍 **Causes:**

- CoreDNS pods not running
- Wrong service name
- App not using internal cluster DNS format

🛠️ **Fixes:**

- Restart `coredns` deployment
- Check `resolv.conf` inside container
- Use proper DNS name like `myservice.my-namespace.svc.cluster.local`

8) explain more about secret management securely and insecurely in k8s?

ANS:

Kubernetes Secrets are objects used to store sensitive data, such as:

- Passwords
- API keys

- TLS certificates
- Docker registry credentials

They are base64-encoded and mounted into pods as environment variables or volumes.

- **Apply RBAC policies so only specific service accounts or namespaces can access them.**
- **Instead of storing secrets inside the cluster, fetch them securely from external tools:**

In Kubernetes, secrets are base64-encoded and can be easily mismanaged if not protected properly.

I use **RBAC**, **encryption at rest**, and tools like **External Secrets Operator** or **Sealed Secrets** to handle secrets securely.

I avoid storing raw secrets in Git and follow the principle of **least privilege** when exposing secrets to pods.

9) How Would You Troubleshoot an Application Failing to Connect to a Third-Party Service Due to Certificate Issues?

1. Identify the Error Clearly

- Check logs of the application or container:

  kubectl logs <pod-name>

Look for typical certificate-related errors like:

- `x509: certificate signed by unknown authority`
- `SSL certificate problem: unable to get local issuer certificate`
- `certificate has expired or is not yet valid`
- `hostname mismatch` or `untrusted CA`

2. Validate the Remote Service Certificate

Use `openssl` or `curl` to inspect the service certificate from a test

Check for:

- Expired certificate
- Self-signed certificate
- Certificate chain issues
- Hostname mismatch

If certificate expired or hostname mismatch:

- Contact 3rd-party vendor to issue a valid certificate.
- Use correct FQDN (not IP) to avoid hostname mismatch.

If the app runs in Kubernetes:

- **Mount custom CA as a secret or configMap**:

When troubleshooting TLS/certificate issues, I first examine the exact error from logs, then validate the remote service certificate using `openssl` or `curl`.

I check if the certificate chain is trusted by the app's runtime, and if not, I import the CA cert to the appropriate trust store.

In Kubernetes, I use configMaps or rebuild images with updated trust stores, and validate connectivity from debug pods.

This structured approach helps prevent downtime and ensures secure connections.

10) How do you manage TLS certificates for apps in Kubernetes?

ANS:

TLS certificates in Kubernetes are crucial for securing in-cluster and external communication (HTTPS, mTLS). I use multiple approaches depending on the use case:

- I manage TLS in Kubernetes primarily using **cert-manager**, which automates certificate issuance, renewal, and storage using Let's Encrypt or internal CAs.
- For manually managed certs, I store them in **Kubernetes TLS secrets** and mount them securely to apps or Ingress.
- I also implement **RBAC and automatic rotation policies**, and make sure apps or ingress reload the certs dynamically to avoid downtime.

10) How do you know if a certificate used by your app has expired or is about to expire?

1. If Using `cert-manager` (Best Practice)

- **cert-manager** maintains TLS certificates using `Certificate` and `CertificateRequest` resources.
- You can track certificate status and expiry directly via CRDs.

kubectl describe certificate myapp-cert -n my-namespace

What to look for:

- `NotAfter` (expiration date)
- `Renewal Time`
- `Ready: False` (means expired or failed renewal)

2. Query Secret and Decode Certificate

If cert is stored in a Kubernetes TLS Secret (e.g., `myapp-tls`), you can manually decode and inspect expiry.

kubectl get secret myapp-tls with grep command

4. External Health Checks (Ingress)

If cert is used in an Ingress (like NGINX or Traefik), you can:

- Use `curl` or `openssl` from inside or outside the cluster:

I monitor certificate expiration in Kubernetes using `cert-manager`'s status fields and Prometheus metrics.

For manual TLS secrets, I decode the certificate using `openssl` to check the `Not After` date.

I also automate alerts using Prometheus rules or custom scripts/CronJobs to ensure we are alerted before expiry — typically 30 days in advance to ensure smooth renewals or rotations.

14) How does helm pull artifacts from nexus artifactory, how the authentication happens and how its pulls artifactory artifact and puts it in K8s cluster ?

ANS:

Helm pulls packaged application charts (`.tgz`) from a Helm repository.

If your **Nexus Artifactory** is configured as a **Helm repository**, you can host and retrieve Helm charts from it just like any remote Helm repo (e.g., Bitnami, ArtifactHub, etc.).

2. Hosting Helm Charts on Nexus

In Nexus (Sonatype Nexus 3.x+ or JFrog Artifactory), you:

- Create a Helm repository (hosted or proxy)
- Upload your `.tgz` Helm charts
- Nexus auto-generates an `index.yaml` file for the repo

When using Helm with Nexus Artifactory, I add the Helm repo via authenticated `helm repo add`, which pulls Helm charts (`.tgz`) hosted on Nexus.

Helm authenticates using basic auth or tokens, downloads the chart, renders it into Kubernetes YAMLs, and deploys them to the cluster using `kubectl`.

I always use token-based auth, and automate this via CI/CD for consistent, secure deployments.

15) Deployed an app in k8s cluster, how do you troubleshoot, give realtime issues what you have faced and why you faced those issues?

When I deploy an app to Kubernetes, I follow a systematic approach to troubleshoot issues:

1 . Check Deployment & Pod Status

- kubectl get pods -o wide -n my-namespace

  Whether pods are running or stuck (e.g., `CrashLoopBackOff`, `ImagePullBackOff`, `Pending`)

2. Check Logs

- If pods are running but not working as expected:

  kubectl logs <pod-name> -n my-namespace 0r kubectl logs <pod-name> -c container-name

Issue 1: `ImagePullBackOff` or `ErrImagePull`

Reason: Wrong image name, invalid tag, or private image repo without credentials.

Fix:

- Corrected the image name/tag.

Created a `docker-registry` secret:

bash
CopyEdit

```
kubectl create secret docker-registry regcred --docker-username=xxx
--docker-password=xxx ...
```

-
- Added `imagePullSecrets` to deployment.

⚠️ **Issue 2: `CrashLoopBackOff`**

Reason: App failed to start repeatedly due to:

- Missing environment variable

- Wrong DB credentials
- Service dependency not ready

Fix:

- Used `kubectl logs` to check crash stack trace.
- Validated environment variables in the `deployment.yaml`.
- Used readiness/liveness probes to delay traffic until app was ready.

⚠️ Issue 3: `Pending` Pods

Reason: Pod couldn't be scheduled due to:

- No node with enough resources
- Taint/toleration mismatch

Fix:

- Checked pod events: `kubectl describe pod`
- Scaled up the cluster or adjusted resource requests/limits.
- Updated tolerations or nodeSelector if required.

⚠️ Issue 4: Readiness Probe Failed

Reason: Application took time to boot up, and Kubernetes considered it unhealthy.

Fix:

- Increased `initialDelaySeconds` in readiness probe.

- Changed the probe path to actual working `/health` endpoint.
- Added retry logic in the app startup to wait for DB or cache.

⚠️ Issue 5: Service Not Reachable

Reason: Misconfigured `service` or wrong port mapping

Fix:

- Verified `targetPort` matched container port.
- Checked `kubectl describe svc` and compared with pod's container spec.
- Used a debug pod (`kubectl run -it busybox`) to `curl` the internal service.

⚠️ Issue 6: Ingress Not Routing Properly

Reason: Wrong host in ingress or certificate misconfiguration.

Fix:

- Checked ingress status: `kubectl describe ingress`
- Verified DNS points to LoadBalancer IP
- Ensured TLS secret existed and was valid

⚠️ Issue 7: PVC Stuck in `Pending`

Reason: No PersistentVolume matched PVC request (wrong size, storage class mismatch)

Fix:

- Described PVC and saw "no matching PV".

- Updated PVC to match available PV.
- Added storage class annotations.

16) Explain about kubelet and what is the purpose of it in k8s?

ANS:

Kubelet is the node agent in Kubernetes that ensures containers are running as expected.

It communicates with the control plane, receives pod specs, and manages pod lifecycles by interacting with the container runtime.

Without kubelet, the node cannot run or manage workloads — it is a critical component of the Kubernetes architecture.

17) Explain more about probes in Kubernetes, like startup, liveliness and readiness?

ANS:

In Kubernetes, **probes** are used by the **kubelet** to check the health and availability of containers in a pod. They determine when a container:

- Is ready to serve traffic
- Is still running as expected
- Is just starting up and needs time

Probe Type	Purpose	Impact on Pod/Service
✅ Readiness	Checks if the container is **ready to serve traffic**	Controls whether pod receives traffic via Services

✅ Liveness	Checks if the container is **still alive/running**	Restarts the container if the check fails
✅ Startup	Checks if the container **startup is complete**	Delays other probes until the app is fully started

🧪 Real Example Scenario

In one project, a Spring Boot app took ~90 seconds to start, but the liveness probe started too early and restarted the container repeatedly.

Fix: Added a `startupProbe` with proper delay, so liveness was paused until the app booted fully.

In Kubernetes, probes help ensure app stability and availability.

I use **readiness probes** to control service traffic, **liveness probes** to restart unhealthy containers, and **startup probes** to delay checks for slow-starting apps.

Correct probe tuning is essential to avoid false failures and unnecessary restarts, especially in production environments.

18) Even If Liveness, Readiness, and Startup Probes Show Healthy — But the App Fails, What Should You Do?

Ans:

This is a real-world problem: the probes return `200 OK`, but the application is functionally broken or degraded internally.

1. Check Application-Level Logs

Even though Kubernetes thinks the container is "healthy", the app might be:

- Throwing internal exceptions
- Returning `500` to real clients

- Stuck due to unresponsive external services

```
kubectl logs <pod-name>
```

🔹 2. Improve Your Probe Design

The default health endpoints (`/health`, `/status`) often return `200 OK` for basic checks only.

✅ Modify them to:

- Check database connectivity
- Verify dependent services are reachable
- Run a functional test (e.g., mock query)

Example:

java

CopyEdit

```
GET /healthz → should check:
- DB
- Redis
- External API
```

🔹 3. Add Application Metrics/Monitoring

Use Prometheus + Grafana to monitor:

- Application error rate
- Latency

- Queue size
- Custom metrics

Alert on business-level failures, not just infra-level health.

🔹 4. Check for Stateful Failures

The app might pass health checks but be internally broken due to:

- Memory leaks
- Deadlocks
- Disk full
- Thread exhaustion

Use:

- `kubectl top pod`
- APM tools like Datadog, New Relic, Dynatrace

🔹 5. Enable Better Observability

Add:

- Structured logging
- Tracing (`OpenTelemetry`)
- External synthetic tests (e.g., ping app from outside)

🟡 Summary (Interview Style)

Even if probes show green, apps can still be broken. I enhance probe endpoints to test critical dependencies like DB/API, and add Prometheus alerts and logs.
Health checks only tell Kubernetes that the container is alive—not that it's functionally correct.

19) What Are CRDs in Kubernetes? How Do You Manage and Use Them?

What Are CRDs (Custom Resource Definitions)?

CRDs are a way to extend Kubernetes by defining custom resources.

- Kubernetes natively has resources like `Pod`, `Service`, `Deployment`.
- With CRDs, you can create your own like:
    - `KafkaTopic`
    - `MySQLBackup`
    - `PrometheusRule`

These are managed like any K8s resource:

bash

CopyEdit

```
kubectl get kafkatopics

kubectl describe mysqlbackup my-backup
```

🧱 CRD = Schema + Controller

Component	Role

CRD	Defines the schema (YAML structure)
Controller	Watches the CRD and takes action

For example:

- `cert-manager` uses CRDs like `Certificate`, `Issuer`
- `ArgoCD`, `Prometheus`, `Istio`, `OpenShift` all use CRDs

CRDs allow me to extend Kubernetes with custom resource types. I've used them in tools like cert-manager and Prometheus, and I manage them like any other K8s resource via GitOps and RBAC.

I ensure schema validation and pair them with controllers to automate resource behavior effectively.

SHELL

1) **How Do You Install a Package on an Isolated Linux Server (No Internet, No Bastion, No NAT Gateway)?**

ANS:

There are two approaches — one without S3 and one with S3, depending on what access is available inside the private environment.

Option 1: Without S3 or Any Public Access

You can manually download dependencies, move them into the server, and install them offline.

Approach A: Create an Offline Package Bundle

For YUM-based systems (RHEL/CentOS/Amazon Linux):

bash

CopyEdit

```
# On an internet-connected machine:

yum install --downloadonly --downloaddir=./offline-nginx nginx


# Transfer the .rpm files using a secure USB or encrypted EBS

scp *.rpm ec2-user@localhost:/tmp/


# On the isolated server:

sudo yum localinstall *.rpm
```

For APT-based systems (Ubuntu/Debian):

bash

CopyEdit

```
# On a connected machine:

apt-get download <package-name>

apt-cache depends <package-name> | grep Depends


# Move .deb files and install:

sudo dpkg -i *.deb
```

> 🔁 You can automate the dependency resolution using `apt-offline` or `yumdownloader`.

Approach B: Create a Local Repo on a USB or Volume

1. Download all `.rpm` or `.deb` files
2. Use `createrepo` (YUM) or `dpkg-scanpackages` (APT)
3. Mount it on the target server as a local repo

Example (RHEL):

bash

CopyEdit

```
mount /dev/xvdf /mnt

createrepo /mnt

echo "[local-repo]

name=Local Repo

baseurl=file:///mnt
```

```
enabled=1

gpgcheck=0" > /etc/yum.repos.d/local.repo

yum install nginx
```

For isolated Linux servers, I use two main approaches.

Without S3 or internet, I prepare offline `.rpm` or `.deb` packages on a connected machine and transfer them via EBS, USB, or AMI customization.

With S3 access via a VPC endpoint, I upload the packages and use `aws s3 cp` to dynamically pull them at runtime and install them via `yum localinstall` or `dpkg`.

This ensures clean, secure, and automated package installation in air-gapped environments.

2) Count how many times each ip address is present in a file (ip's are given in either list, or array or dictionary). ?

Example File Content (`ips.txt`):

["192.168.1.1", "10.0.0.1", "192.168.1.1"]

{"server1": "10.0.0.1", "server2": "172.16.0.1"}

192.168.1.1

10.0.0.1

```
#!/bin/bash

# File name containing the IPs

FILE="ips.txt"

# Extract all IPv4 addresses using grep and regex

grep -oE '\b([0-9]{1,3}\.){3}[0-9]{1,3}\b' "$FILE" |

 sort |

 uniq -c |

 sort -nr
```

What This Does:

- `grep -oE`: Extracts only the IP addresses from each line (even inside JSON, lists, arrays, or text)
- `sort`: Sorts them so `uniq` can count duplicates
- `uniq -c`: Counts occurrences of each unique IP
- `sort -nr`: Sorts the result by descending count

3) Using Istio or in Istio how do you expose endpoints externally.?

In Istio, we expose services externally using the Istio Ingress Gateway, which acts as a layer 7 load balancer (similar to a reverse proxy) for incoming traffic into the mesh.

🔹 Key Resources Used to Expose Services Externally in Istio:

Resource	Purpose

`Gateway`	Defines how incoming traffic is received (host, port, TLS, etc.)
`VirtualService`	Routes traffic from the gateway to the correct service in the mesh
`Service`	The internal Kubernetes service

🧱 Basic Architecture Flow

External client → Istio Ingress Gateway → VirtualService → Internal K8s Service → Pod