**Section 1: Core Concepts (30 mins)**

1. Differentiate **Continuous Integration, Continuous Delivery, Continuous Deployment**.

These three practices represent progressive stages of automation in software development.

* **Continuous Integration (CI):** This is the practice where developers frequently merge their code changes into a central repository. Each merge triggers an automated process that **builds** the code and runs **unit/integration tests**. The main goal is to find and fix integration bugs quickly.
* **Continuous Delivery (CDelivery):** This is an extension of CI. After the build and testing phases are successful, the software is automatically packaged and **released to a testing or staging environment**. The final step of deploying to the production environment requires **manual approval**.
* **Continuous Deployment (CDeployment):** This is the most advanced stage. Every code change that passes the entire automated pipeline (build, test, stage) is **automatically deployed to the production environment** without any human intervention.

In short:

* **CI** = Automated build & test.
* **CDelivery** = CI + Automated release to staging + **Manual** deployment to production.
* **CDeployment** = CI + **Automated** release to production.

* 1. .What is the purpose of a **Jenkinsfile**?

A Jenkinsfile is a text file that defines your entire build, test, and deployment workflow as code. It is the core component of "Pipeline as Code" in Jenkins.

Instead of configuring a job manually through the Jenkins user interface, you write the steps of your pipeline in a file (the Jenkinsfile) and commit it to your project's source control repository (like Git).

The primary purposes and benefits of using a Jenkinsfile are:

* **Versioning:** Your CI/CD pipeline is versioned and tracked alongside your application code. You can see the history of changes to your deployment process.
* **Collaboration:** The pipeline becomes part of the code review process. The entire team can read, write, and approve changes to the build and deployment logic.
* **Reusability:** The pipeline is portable. You can easily reuse the same Jenkinsfile for different branches or even different projects with similar needs.
* **Durability:** Since the pipeline is defined in code, it survives Jenkins server restarts or migrations. The source control repository is the single source of truth.
* **Complex Pipelines:** It enables you to build sophisticated pipelines with conditional logic, parallel execution, and complex steps that are difficult or impossible to manage through the traditional UI.
  1. Contrast **declarative vs scripted pipelines**.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Declarative Pipeline** | **Scripted Pipeline** |
| **Syntax** | Uses a structured, predefined syntax (pipeline {}) | Uses full Groovy scripting |
| **Ease of Use** | Easier to read/write; beginner-friendly | More complex; requires Groovy knowledge |
| **Flexibility** | Limited flexibility | Highly flexible and customizable |
| **Error Handling** | Built-in support (post, options, when, etc.) | Manual error handling with try-catch |
| **Tooling Support** | Better supported by Jenkins UI | Less visual support |
| **Use Case** | Standard CI/CD workflows | Complex or highly dynamic workflows |

* 1. Difference between **Freestyle job** and **Pipeline job**.
* The fundamental difference between a Freestyle job and a Pipeline job in Jenkins lies in how you configure them and what they are designed to do.
* **Freestyle jobs** are the original, classic way of creating jobs in Jenkins. You configure everything through the **web UI** using forms, dropdowns, and buttons.
* **Pipeline jobs** are the modern, standard approach. You define your entire build, test, and delivery process as code in a text file called a Jenkinsfile.

|  |  |  |
| --- | --- | --- |
| Feature | Freestyle Job | Pipeline Job |
| **Configuration** | ✅ **Through the UI.** Easy for simple tasks and beginners. | ⚠️ **As Code** in a Jenkinsfile. This is the core concept. |
| **Version Control** | Stored only on the Jenkins server as XML files. Not easily versioned. | Stored alongside your application code in a source control system (like Git). |
| **Complexity** | Best for simple, linear tasks (e.g., run a script, build code). | ✅ Designed for complex, multi-stage workflows with parallel steps, conditions, and loops. |
| **Durability** | The job configuration can be lost if your Jenkins server fails. | The pipeline definition (Jenkinsfile) is safe in your code repository. |
| **Reusability** | Difficult to reuse or share. You have to copy jobs manually. | ✅ The Jenkinsfile can be easily reused across different branches or projects. |

5. How would you secure **credentials (Git/Docker/Slack tokens)** in Jenkins?

**Store credentials securely** via:

* **Jenkins Dashboard → Manage Jenkins → Credentials**
* Choose appropriate **scope** (Global, System, Folder)
* Add as **Username/Password**, **Secret Text**, **SSH Key**, etc.

**Access in Pipelines**:

* Use credentials() or withCredentials() in the Jenkinsfile:

**Avoid hardcoding secrets** in:

* Jenkinsfiles
* Console outputs (use echo cautiously)
* Environment variables (limit visibility)

**Restrict access**:

* Use **Role-Based Access Control (RBAC)** to limit who can view/manage credentials.

6. If a Jenkins job fails randomly, what are the first 3 things you’d check?

**First 3 Things to Check When Jenkins Job Fails Randomly**

**1. Logs & Error Messages**

- Check the Jenkins console output for the exact failure point.  
- Look for patterns in errors: does it fail on the same step (build/test/deploy)?  
- If it’s a test, suspect flaky tests; if build tools, check dependency resolution or network.

**2. Environment & Agent Stability**

- Verify if the job runs on the same Jenkins agent or varies.  
- Check for resource bottlenecks (memory, CPU, disk).  
- Investigate network instability and workspace conflicts (try wiping workspace).

**3. Dependencies & External Systems**

- Confirm external service dependencies (DB, APIs, artifact repositories).  
- Look for intermittent service downtime, expired credentials, or repo download failures.

**Additional Checks**

- Concurrency issues (parallel builds interfering).  
- Pipeline script instability.  
- Non-deterministic/flaky test cases.

**Section 3: Debugging & Scenarios (30 mins)**

1. Pipeline fails with error Exit code 137. What could cause this?

**Causes of Exit Code 137 in Jenkins**

1. **Out of Memory (OOMKilled)**
   * Node agent/container runs out of memory.
   * The OS or Kubernetes kills the process (npm install, npm test, or zip) when memory exceeds limits.
2. **Timeouts**
   * Jenkins job or stage exceeded its configured timeout.
   * If Jenkins kills it, it can also show exit code 137.
3. **Manual kill**
   * Admin/user clicked “Stop Build” in Jenkins.
4. **Docker/Kubernetes resource limits**
   * If the agent runs inside Docker or K8s pod with limited memory (e.g., 512Mi), heavy steps like npm install or tests can be killed.
5. Jenkins has 2 agents but 5 long jobs. How do you optimize execution?

1. Increase parallelism by scaling agents

Static agents → Add more permanent agents (VMs, containers).

Dynamic agents (better) → Use Jenkins plugins for on-demand scaling:

Kubernetes plugin → Jenkins launches ephemeral pods per job.

EC2 plugin → Jenkins auto-starts EC2 instances as agents.

Docker plugin → Spins up disposable Docker agents per build.

👉 This way, 5 jobs can run in parallel instead of waiting.

2. Optimize job execution (reduce job length)

Split pipelines into stages that run in parallel:

parallel {

stage('Lint') { steps { sh 'npm run lint' } }

stage('Unit Tests') { steps { sh 'npm test' } }

stage('Build') { steps { sh 'npm run build' } }

}

Use caching (npm cache, Docker layers) to reduce repeated install times.

Skip unnecessary steps for non-critical branches (e.g., don’t build artifacts for feature branches).

3. Use agent labels wisely

Tag agents with labels (linux, windows, high-memory).

Assign heavy jobs to high-capacity agents, lightweight jobs to smaller ones.

Example:

agent { label 'high-memory' }

4. Use job throttling / concurrency control

Install Throttle Concurrent Builds plugin → control how many builds of the same job run in parallel.

Prevents one long job from hogging both agents.

5. Pipeline optimization with when conditions

Only run heavy steps when needed:

when { branch 'main' }

→ reduces wasted compute on feature branches.

6. Queue management policies

Use Priority Sorter plugin → prioritize critical jobs.

Example: CI for main branch gets higher priority than feature branches.

1. Jenkins master goes down. How do you recover?

Jenkins master stores all configurations, job definitions, build history, plugins, credentials in its JENKINS\_HOME directory.

Agents can’t run anything without the master.

👉 So, recovery = restore Jenkins + JENKINS\_HOME.

⚙️ 2. Recovery steps

A. If server is still accessible (master crashed but disk OK)

Stop Jenkins service (to avoid corruption):

sudo systemctl stop jenkins

Backup JENKINS\_HOME (default: /var/lib/jenkins):

sudo tar -czvf jenkins\_home\_backup.tar.gz /var/lib/jenkins

Fix root cause (e.g., memory, Java crash, disk full).

Restart Jenkins:

sudo systemctl start jenkins

Validate jobs, plugins, credentials.

B. If master machine is dead (need new server)

Provision a new server (VM, Docker, Kubernetes).

Install same Jenkins version as before (to avoid plugin mismatch).

For Ubuntu:

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

sudo apt-get update

sudo apt-get install jenkins

Restore backup of JENKINS\_HOME to new server:

sudo systemctl stop jenkins

sudo rm -rf /var/lib/jenkins/\*

sudo tar -xzvf jenkins\_home\_backup.tar.gz -C /var/lib/jenkins

sudo chown -R jenkins:jenkins /var/lib/jenkins

sudo systemctl start jenkins

Update DNS/Load Balancer so agents/users point to new master.

C. If no backup exists 😬

You’ll have to:

Reinstall Jenkins

Recreate jobs manually (or pull Jenkinsfiles from Git if pipelines are code-based)

Reinstall plugins and credentials

👉 This is why backing up JENKINS\_HOME is critical.

🔧 3. Preventive best practices

Regular backups of JENKINS\_HOME (cron job, S3 sync, rsync).

Store pipelines as code (Jenkinsfile in Git) → makes recovery easier.

Use High Availability setups:

Run Jenkins in Kubernetes with persistent volumes.

Use Jenkins Operations Center (CloudBees).

Set up warm standby master that can take over quickly.

Monitor master (CPU, RAM, disk) → avoid unplanned crashes.

1. A developer accidentally exposed a GitHub token in Jenkins logs. What actions do you take?

**1. Contain the incident**

* **Immediately revoke the exposed GitHub token** in GitHub (Settings → Developer Settings → Personal Access Tokens).
* If it was an **App/Org token**, revoke it from the organization’s GitHub settings.  
  👉 This prevents attackers from using the leaked token.

**🔎 2. Clean up Jenkins**

* **Remove/rotate the secret** in Jenkins credentials.
* Verify no other jobs are echoing sensitive env vars (echo $TOKEN, etc.).
* Mask secrets in pipelines using withCredentials block:
* withCredentials([string(credentialsId: 'GITHUB\_TOKEN', variable: 'TOKEN')]) {
* sh 'git clone https://$TOKEN@github.com/org/repo.git'
* }

→ ensures token doesn’t appear in logs.

**🔎 3. Sanitize logs**

* Jenkins stores logs in JENKINS\_HOME/jobs/.../builds/.../log.
* Either **delete** or **redact** the affected logs.
* If you’re using **external log storage (e.g., ELK, Splunk, CloudWatch)** → delete/rotate logs there too.

**🔎 4. Audit for misuse**

* Check GitHub security logs → see if the exposed token was used after being leaked.
* Review Jenkins build history for further exposures.
* Rotate any dependent secrets (e.g., if token had repo write access, check repos for tampering).

**🔎 5. Prevent future leaks**

* **Never hardcode tokens** → always store in Jenkins credentials store.
* **Mask secrets** in pipeline logs:
* pipeline {
* options {
* maskPasswords() // requires Mask Passwords plugin
* }
* }
* **Restrict access to logs** in Jenkins (RBAC).
* Enable **Secret Text masking** for credentials.
* Train developers: avoid echo on sensitive vars.

Section 2 :



