Section 1: Core Concepts

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

Continuous Integration:

- Developers commit code frequently (often daily).
- Each commit triggers an automated build and test.
- Detects integration issues early (e.g., build failures, test failures).

Continuous Delivery:

- Automated testing (unit, integration, UI).
- Automated deployment to staging environments.
- Manual step to approve release to production.

Continuous Deployment:

- Fully automated pipeline from commit to production.
- High confidence in automated testing (since there's no manual gate).
- Very fast feedback loops.

2. What is the purpose of a Jenkinsfile?

Purpose	Description
1. Pipeline as Code	The Jenkins file turns CI/CD workflows into version-controlled code, ensuring consistency and traceability.
2. Automation	Automates build, test, and deployment processes to reduce manual effort and errors.
3. Reproducibility	Ensures that pipelines run the same way every time, across environments and team members.
4. Collaboration	Enables teams to collaborate on CI/CD logic just like they do on application code (via pull requests, code reviews, etc.).
5. Integration	Works seamlessly with Jenkins plugins and tools (e.g., Docker, Maven, Gradle, Kubernetes).

3. Contrast declarative vs scripted pipelines.

Scripted Pipeline	Declarative Pipeline
Uses Groovy scripting.	Uses structured Groovy DSL.
Highly customizable.	Easier to use, less flexible.

Scripted Pipeline	Declarative Pipeline
Manual handling is needed.	Built-in error handling.
More reusable & modular.	Less focus on reuse.
Can be complex.	More readable & beginner-friendly.
Best for advanced workflows.	Standard CI/CD tasks.

4. Difference between Freestyle job and Pipeline job.

Freestyle Job	Pipeline Job
Basic job type with a simple UI-based configuration	Advanced job type defined as code (Jenkins file)
Jenkins (early versions)	Jenkins 2.x and newer
Point-and-click in Jenkins UI	Code-based (written in Groovy syntax)
Limited	Highly flexible (conditionals, loops, logic)
Not easily version-controlled	Stored as code in Git
Low (manual duplication needed)	High (via shared libraries, parameterization)
Basic (not very customizable)	Advanced (try-catch, post blocks, etc.)
Difficult to manage	Ideal for complex CI/CD pipelines
Not supported natively	Supported via parallel block
Basic build logs and console output	Enhanced UI (especially with Blue Ocean plugin)

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

1. Use Jenkins Credentials Store

Steps:

1. Go to Jenkins Dashboard → Manage Jenkins → Credentials.

2. Choose the appropriate domain (usually “Global”).
3. Click "Add Credentials".
4. Select a type:
    - Username & Password (for Git auth, Docker registries)
    - Secret Text (for tokens like Slack, GitHub PAT)
    - SSH Username with Private Key
    - Certificate, Secret File, etc.

2. Access Credentials Securely in Pipelines

Declarative Example:

```
pipeline {
  agent any

  environment {
    GITHUB_TOKEN = credentials('GITHUB_TOKEN') // from Jenkins credentials store
  }

  stages {
    stage('Clone Repo') {
      steps {
        sh 'git clone https://user:${GITHUB_TOKEN}@github.com/your-org/your-repo.git'
      }
    }
  }
}
```

Scripted Example:

```
node {
  withCredentials([string(credentialsId: 'GITHUB_TOKEN', variable: 'TOKEN')]) {
    sh 'curl -H "Authorization: token ${TOKEN}" https://api.github.com/user'
  }
```

}

3. Restrict Access to Credentials

- Use Credential Domains to scope credentials only to specific jobs or folders.
- Apply Role-Based Access Control (RBAC) via plugins to ensure only authorized users can view/edit credentials.
- Avoid using echo $SECRET in shell steps — this can expose secrets in logs.

4. Use with Credentials Block for Extra Security

The withCredentials block ensures that secrets are only available temporarily during the block execution and are masked in logs.

```
withCredentials([usernamePassword(credentialsId: 'docker-creds', usernameVariable: 'DOCKER_USER', passwordVariable: 'DOCKER_PASS')]) {
    sh "docker login -u $DOCKER_USER -p $DOCKER_PASS"
}
```

5. Integrate with Secret Management Tools (Optional Advanced)

- HashiCorp Vault
- AWS Secrets Manager
- Azure Key Vault
- CyberArk

Jenkins plugins exist to pull secrets dynamically at runtime without storing them in Jenkins.

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

1. Check Build Logs and Error Stack Trace

Why?

- The logs will show where and why the failure happened.
- Random failures may come from timeouts, flaky tests, missing resources, or network issues.

What to Look For:

- Exceptions or stack traces (NullPointerException, TimeoutException, etc.)
- Network failures (Could not resolve host, Connection reset)
- Permission denied or file-not-found errors
- Messages like test failed, dependency not found, port already in use, etc.

2. Check for Environment Issues (Agent/Node Problems)

Why?

- If a node is unstable, overloaded, or misconfigured, builds may randomly fail on that node.
- Randomness can stem from node-specific setup, such as:
    - Missing tools (e.g., Java, Docker, Python)
    - Temporary file system issues
    - RAM/CPU exhaustion
    - Intermittent network or DNS problems

What to Check:

- Node status in Jenkins (Jenkins → Manage Nodes)
- Disk space, memory, CPU usage
- Check if the job always fails on the same agent
- Reboot or take the node offline if needed

3. Check for Flaky Tests or Race Conditions

Why?

- Tests that pass sometimes and fail other times are called flaky.
- This can happen due to:
    - Uncontrolled timeouts
    - Lack of synchronization in test code
    - Dependency on external systems (APIs, databases)
    - Poor isolation between tests

What to Check:

- Which tests fail? Are they consistent?
- Can you reproduce the failure locally or by rerunning the same job?
- Consider retrying flaky tests automatically using testing frameworks or Jenkins plugins

Section 2: Hands-On Pipeline

github.com

Section 3: Debugging & Scenarios

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

Common causes of Exit code 137:

1. Out of Memory (OOM) Killer
    - The process consumed more memory than allowed or available on the node, so the OS killed it to free memory.
    - Happens often in resource-intensive builds, tests, or Docker containers.
2. Docker container killed due to resource limits
    - If your pipeline runs inside a Docker container with memory limits, hitting that limit causes the container to be killed with exit 137.
3. Manual kill or timeout
    - Someone or something forcibly stopped the job.
    - Jenkins or underlying systems killing the process after timeout.
4. System-level issues
    - Low disk space or other resource exhaustion leading to process termination.

2. Jenkins has 2 agents but 5 long jobs. How do you optimize execution?

1. Use Job Queuing and Labeling Strategically

- Jenkins will automatically queue jobs when agents are busy.
- Make sure all jobs are assigned to agents with appropriate labels.

Action:

- Assign both agents the same label (e.g., long-runner) and ensure the jobs are targeting that label.

```
agent { label 'long-runner' }
```

2. Parallelize within Pipelines (if possible)

- If some jobs have independent tasks or stages, parallelize those tasks within a single pipeline.

```
stage('Parallel Tasks') {
  parallel {
    stage('Task 1') {
      steps {
        sh './task1.sh'
      }
    }
```

```
        stage('Task 2') {
            steps {
                sh './task2.sh'
            }
        }
    }
}
```

3. Throttle Concurrent Builds Per Job

- Prevent one job from hogging agents.

In the job configuration:

- Go to Build Triggers → Throttle Concurrent Builds
- Limit to 1 concurrent build per job (or tune as needed)

4. Use Job Prioritization (Optional)

- If some jobs are more important, install and configure the Priority Sorter Plugin.

5. Time-Based Scheduling (for Nightly/Batch Jobs)

- Move non-critical or long-running batch jobs to off-peak hours using cron schedules.

```
triggers {
    cron('H 2 * * *')
}
```

6. Split or Optimize Long Jobs

- Break long jobs into smaller pipelines or incremental stages.
- Cache dependencies or build artifacts to save time (e.g., use node_modules caching for Node.js).

7. Scale Out (If Possible)

- Add more agents (on-demand, cloud-based, or static)
- Use Kubernetes plugin or EC2 plugin to spin up agents as needed

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

If the Jenkins master (controller) goes down, recovery depends on how Jenkins is backed up and deployed. Below is a step-by-step recovery plan to get Jenkins back online with minimal data loss.

1. Identify the Cause of Failure

- A hardware or VM issue (host crash, disk full, etc.)
- A Jenkins software issue (corrupt configs, plugin failure)
- A resource issue (memory exhaustion, OOM)
- A misconfiguration (e.g., recent plugin install/update)

Action:

- Check logs (/var/log/jenkins/jenkins.log or system logs)
- Check disk space, memory, and CPU usage

2. Restore from Backup (if needed)

If Jenkins master is unrecoverable, restore from your most recent backup.

What to Back Up Regularly:

Component	Location
Job configs	$JENKINS_HOME/jobs/
Build history	$JENKINS_HOME/jobs/*/builds/
Plugins	$JENKINS_HOME/plugins/
Credentials	$JENKINS_HOME/credentials.xml
User data	$JENKINS_HOME/users/
Global config	$JENKINS_HOME/config.xml

3. Redeploy Jenkins (if server is unrecoverable)

If the Jenkins server is completely lost:

1. Provision a new machine/VM/container
2. Install Jenkins (same version as before)
3. Restore $JENKINS_HOME directory from backup
4. Start Jenkins service

4. Reconnect Agents

- SSH or JNLP: Reconnect them manually or script reconnection.

- Cloud agents: Jenkins should spin them up automatically once configured.

5. Recheck Credentials and Secrets

- The credentials store is intact (credentials.xml)
- Secret keys haven't changed (important for pipeline decryption)

6. Test Critical Jobs

- Trigger a few test builds to confirm functionality.
- Check plugin compatibility if you restored from backup or upgraded Jenkins.

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

1. Revoke the Exposed Token Immediately

- Go to the GitHub Developer Settings → Personal Access Tokens (or GitHub Apps / OAuth Apps, depending on the token type).
- Revoke or regenerate the token that was exposed.

2. Remove the Token from Jenkins Logs

- Locate the job and delete the console output or entire build history:
  - Jenkins → Job → Build # → Delete build
- Or manually clean logs in the Jenkins filesystem:
- rm -rf $JENKINS_HOME/jobs/<job-name>/builds/<build-number>
- If using log aggregation (e.g., ELK, Datadog), scrub those too if possible.

3. Audit Who Accessed the Logs

- Check Jenkins user access logs (Audit Trail, if enabled)
- Check GitHub for any suspicious activity on that token (API calls, clone/push events)

4. Use Jenkins Credentials Store

Never hardcode tokens in Jenkinsfiles or shell scripts.

Correct approach:

```
withCredentials([string(credentialsId: 'github-token', variable: 'GITHUB_TOKEN')]) {
   sh 'curl -H "Authorization: token ${GITHUB_TOKEN}" https://api.github.com/user'
}
```

5. Enable Credentials Masking (built-in)

Jenkins automatically masks secrets stored in the credentials store from console output, but only if:

- They are referenced correctly (not echoed or written to file)
- They're used inside with Credentials {} blocks

6. Train Developers on Secure CI Practices

Conduct regular code reviews and training on:

- Handling secrets
- Using credentials plugins
- Avoiding echo $TOKEN or logging sensitive data

7. Implement Secret Scanning Tools

Use tools to detect secrets in pipelines or source code, e.g.:

- GitHub Advanced Security
- Gitleaks
- TruffleHog
- Jenkins plugins like "Credentials Binding" + masking