DEVOPS ASSESMENT 1

SECTION 1

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

ANS:Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD) are sequential stages of an automated software delivery pipeline that together form the backbone of modern DevOps practices. They differ primarily in their scope and the degree of human intervention required

Continuous Integration(CI): Continuous Integration is the practice of merging developer code changes into a central repository frequently and automatically. The key idea is to "integrate" code as often as possible to detect and fix integration bugs early.
CI process typically involves:

1) **Automated builds:** A CI server automatically builds the application every time a developer commits new code.
2) **Automated tests:** The server runs a suite of automated tests (like unit tests and integration tests) to validate the new code.
3) **Immediate feedback:** If the build or tests fail, the developer is immediately notified, allowing for quick fixes.

Continuous Delivery (CD): Continuous Delivery is a software development practice where code changes are automatically built, tested, and prepared for a release to production. It builds upon CI by extending the automation to include the release process.

The CD process includes:

1) **All CI steps** (automatic build and testing).
2) **Automated packaging and staging:** The validated code is automatically packaged and moved to a staging or testing environment that mirrors production.
3) **Manual approval for deployment:** A human still makes the final decision to deploy the software to the live production environment, typically with a single click. This manual gate gives the business or operations team control over when new features go live.

Continuous Deployment (CD): Continuous Deployment is an extension of Continuous Delivery where every code change that passes all automated tests is automatically deployed to production without any manual approval. This represents the highest level of automation in the delivery pipeline.

Key difference in **CD** is Absense of manual gate, If the code passes all checks, it's released to users automatically.

2) What is the purpose of a Jenkinsfile?

ANS:A **Jenkinsfile** is a text file that defines a Jenkins Pipeline, the automated process for building, testing, and deploying your software. Its main purpose is to treat your CI/CD pipeline as **"Pipeline as Code,"** storing it in your version control system (like Git) alongside your application code.

PURPOSE OF JENKINS

1) **Version Control:**Because the Jenkinsfile lives in your code repository, you can track every change to your pipeline. This lets you see who made a change, when they made it, and easily roll back to a previous version if something breaks. This is much more reliable than using the Jenkins UI for configuration.
2) **Collaboration and Audit Trail**: A shared Jenkinsfile provides a clear record of your build process. Teams can collaborate on pipeline changes through standard code reviews, preventing a single person from being the only expert on a complex build configuration.
3) **Consistency**: A Jenkinsfile guarantees that every developer and every branch uses the exact same build process. This eliminates the classic "it works on my machine" problem and ensures a consistent, reliable workflow across your entire team.
4) **Automated Multi-Stage Pipelines**: The Jenkinsfile lets you define the entire software delivery lifecycle, from a code commit all the way to deployment. You can specify different stages, like Build, Test, and Deploy, and the specific steps within each one.

3) About Contrast declarative vs scripted pipelines.

Declarative Pipeline:Declarative is a modern and more structured approach. It was designed to be easier to read and write, especially for those who are new to Jenkins or don't have extensive programming knowledge. It uses a rigid, predefined structure, which makes pipelines more consistent and less prone to errors.

Declarative Pipeline is best for:Standard CI/CD workflows, simpler projects, and teams that prioritize readability and maintainability.

Key Features:

1) **Structure:**It has a strict, well-defined structure.
2) **Directives**: Uses directives like agent, stages, when, and options for defining the pipeline's behavior.
3) **Validation**: Syntax is validated before execution, catching errors early.
4) **Restart from Stage**: Supports restarting a failed build from the stage that failed.

Scripted Pipeline: Scripted is the original and more flexible pipeline syntax. It's essentially a Groovy script that is executed on the Jenkins master. This gives it a high

degree of control and power, but it also makes the code more complex and harder to maintain for larger teams.
Scripted Pipeline is best for : Complex workflows, advanced logic, dynamic stages, and teams with strong Groovy scripting skills.
Key Features:

- **Flexibility:**Allows for complex flow control and logic using standard Groovy constructs like if/else, try/catch, and loops.
- **Full Groovy**: You can use almost any Groovy feature, making it highly customizable.
- **No Predefined Structure**: Lacks a mandatory structure, which can lead to inconsistencies if not well-documented.

5) Difference between Freestyle job and Pipeline job.

ANS: The main difference between a Jenkins Freestyle job and a Pipeline job is the **method of configuration and the level of automation**. Freestyle jobs are configured primarily through the graphical user interface (GUI), while Pipeline jobs are defined as code in a Jenkinsfile.

Freestyle Job:A **Freestyle job** is Jenkins's traditional, simple, and flexible job type. It's configured by manually adding a sequence of build steps and post-build actions via the web UI.

· **GUI-based:** All configuration is done through the Jenkins web interface. You select from a list of available build tools and steps.

· **Simple Use Cases:** Ideal for straightforward, single-step tasks like running a simple script, compiling a project, or archiving artifacts.

· **Limited Scalability:** Managing many Freestyle jobs can become difficult and error-prone. It's not suited for complex, multi-stage, or cross-platform workflows.

· **Less Repeatable:** Since the configuration isn't stored as code, it's hard to replicate a Freestyle job exactly or track changes over time.

Pipeline Job:A Pipeline job is a modern and more robust approach to defining a CI/CD workflow. It's based on the "Pipeline as Code" principle, where the entire delivery process is scripted in a file called a Jenkinsfile.

- **Code-based (Jenkinsfile):** The entire pipeline is defined in a text file that's committed to a version control system (like Git) with the rest of your project's code. This allows for versioning, auditing, and collaboration.
- **Complex Workflows:** Perfectly suited for complex, multi-stage workflows involving building, testing, deploying to different environments, and parallel execution.
- **Scalability and Consistency:** Pipelines are highly repeatable and easily shareable across projects and teams, ensuring a consistent build process everywhere.

- **Built-in Features:** Pipelines can survive Jenkins master restarts and include built-in features for checkpoints, retries, and manual approval steps, making them incredibly durable.

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

ANS: We would secure credentials in Jenkins by using the **built-in Credentials Plugin**, which provides a centralized, encrypted storage for sensitive data. Never store credentials in plain text within your Jenkinsfiles or build scripts.

Using the Credentials Plugin

The **Credentials Plugin** is the standard and recommended way to manage secrets like passwords, tokens, and SSH keys in Jenkins. It allows you to define credentials in the Jenkins web UI, where they are encrypted and stored on the Jenkins controller.

1) **Add Credentials**: In Jenkins, navigate to Dashboard > Manage Jenkins > Credentials. Here, you can add various types of credentials, such as:

- **Secret Text**: For API tokens, Slack tokens, or other single-line secrets.
- **Username with password**: For Git, Docker, or other services requiring both.
- **SSH Username with Private Key**: For connecting to servers via SSH.
- **Secret File**: For storing entire configuration files that contain secrets.

2) **Use Credentials in a Pipeline:**Once a credential is saved, you reference it in your Pipeline job using a unique ID. The withCredentials step in a Jenkinsfile is the standard way to securely inject these secrets into your build environment. This step binds the credential to an environment variable that is only available for the duration of that specific code block. This prevents the secrets from being exposed in logs or other parts of the build.

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

ANS: If a Jenkins job fails randomly, the first three things you'd check are the **console output**, **dependencies and environment**, and the **build history**.

- **Console Output:**First, check the **console output** of the failed job. This is the log of the build process and will often contain the exact error message, stack trace, or a non-zero exit code that caused the failure. Look for clues like a failed test, a timeout error, or a file access permission issue.
- **Dependencies and Environment:**Next, investigate the **dependencies and environment**. Random failures are frequently caused by non-deterministic factors. Check external services like databases or artifact repositories to see if they were intermittently unavailable or slow. Also, inspect the build agent for resource issues like low disk space or memory. Ensure that tool versions (e.g., Node.js, Maven, Python) are consistent and correctly configured across all agents.
- **Build History:**Finally, analyze the **build history**. Compare the failed build to recent successful ones. Look for any recent changes to the source code, a new

pull request, or a change in a configuration file that might have introduced a flaky test or a bug. Also, check if the failure started after a recent update to Jenkins itself, a specific plugin, or a system dependency.

SECTION 3:Debugging & Scenarios

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

ANS:A pipeline failure with exit code 137 typically means the process was **forcibly terminated**. This is usually due to the operating system's out-of-memory killer (OOM killer) stopping the process because the Jenkins agent ran out of available memory.

- **npm install**: This command can be very memory-intensive, especially with a large number of dependencies. If your project has many packages or a complex dependency tree, the node process might consume all available memory on the build agent, causing the OOM killer to terminate it.
- **npm test**: Similarly, running a large test suite can require a significant amount of memory. Some test runners or test environments may leak memory or be very resource-heavy.

How to troubleshoot and fix

- **Increase Agent Memory**: The most direct solution is to provision a Jenkins build agent with more RAM.
- **Allocate More Memory to Node**: You can try to explicitly give Node.js more memory by setting the NODE_OPTIONS environment variable. For example: NODE_OPTIONS="--max-old-space-size=4096" npm install.
- **Clean Up the Workspace**: Make sure your deleteDir() step is configured correctly in the post section to remove old files that might consume disk space and contribute to memory pressure.

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

ANS: The key to optimizing execution is to ensure your jobs are not idle while an agent is free.

Here are three main strategies:

- **Optimize Resource Usage with Parallelism:**

You've already started this by using the parallel block for Install Dependencies and Run Unit Tests. This is excellent, as it allows two different agents to work on the same job simultaneously, assuming the stages are independent. For example, if your test suite is split, you could run different sets of tests in parallel.

- **Implement a Smart Queuing Strategy:**

Instead of just letting jobs queue up, you can prioritize them. You might have a "critical bug fix" job that needs to be built and deployed immediately, while a "daily report" job can wait. Jenkins has a built-in priority queue that you can configure to ensure important jobs are executed first, reducing their overall wait time.

- **Split Long-Running Tasks into Separate Jobs:**

The parallelization you've implemented in your current Jenkinsfile is a great start. To further optimize, you could explore Jenkins' built-in queuing features or consider more advanced setups with dedicated agents for specific tasks, which would further maximize the efficiency of your pipeline execution.

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

ANS: The recovery process for a Jenkins master depends heavily on how it was set up and what kind of backups are available.

- **Stop and Diagnose the Failure:**The first step is to prevent any further issues. Stop the Jenkins service on the server. Then, check the system logs (/var/log/jenkins/jenkins.log on Linux) to understand what caused the failure.
- **Restore from Backup:** The most reliable way to recover is to restore from a recent backup. Jenkins stores all of its configuration data, job history, and plugins in the $JENKINS_HOME directory.

The steps would typically be:

- Set up a new server or use a spare one.
- Install the same version of Jenkins.
- Copy the $JENKINS_HOME directory from your most recent backup to the new server.
- Start the Jenkins service.
- **Re-connect Agents:**Once the master is back online, the next step is to ensure all of your agents reconnect successfully. If your agents are configured with the JNLP launch method, they should automatically reconnect.

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

ANS: Here are the steps you would take:

- **Revoke the Compromised Token Immediately:**The very first action is to go to GitHub and **revoke the exposed token**. This is the most critical step to prevent unauthorized access to the repository and any associated accounts. Don't wait to investigate; the priority is to cut off the attacker's access.
- **Purge the Logs from Jenkins:**Next, you need to remove the sensitive information from the Jenkins logs. The Jenkins log files contain the plaintext token, and they can be accessed by anyone with sufficient permissions.
- **Rotate All Affected Credentials:**A single token might be part of a larger set of credentials. You should investigate if the same token or a similar one was used for other services like Docker or Slack. As a best practice, **rotate all credentials that could have been affected**, even if you're not 100% sure they were compromised.

- **Implement Proactive Measures:**To prevent this from happening again, you must review and strengthen your security practices.

1. **Credentials Plugin**:Ensure that all secrets are being managed using the Jenkins Credentials Plugin, as outlined in the Jenkinsfile you provided.
2. **Secret Masking**: Configure Jenkins to automatically mask sensitive information in the console logs.
3. **Review Permissions**: Re-evaluate who has access to the Jenkins master, agents, and logs. Follow the principle of least privilege.
4. **Educate the Team**: Conduct a brief session with the development team to explain the risks of exposing secrets and the importance of using secure methods for handling them.