**CICD THEORY:**

**### Theory**

**Describe in your own words what is Jenkins and what it is used for**

Jenkins is an open-source automation server primarily used to automate tasks related to building, testing, and deploying software. It enables continuous integration (CI) and continuous delivery (CD) of projects by automatically triggering builds and tests whenever code changes occur, ensuring that issues are detected early in the development process. Jenkins can integrate with various tools, version control systems like Git, and other DevOps tools to streamline the entire software development lifecycle, making it a key component in maintaining a fast, efficient, and reliable release process.

**What do you need to have installed on your machine before installing Jenkins?**

Java Development Kit (JDK):

* Jenkins is a Java-based application, so you need to have the JDK installed. Jenkins requires Java 8 or higher, but it's recommended to use the latest Long-Term Support (LTS) version of Java.
* You can verify Java is installed by running the command: java -version.

Web Browser:

* Jenkins uses a web interface, so you'll need a browser to access and interact with Jenkins after installation.

Git:

If we are pulling code from Git repositories for your Jenkins builds.

I recommended to install Jenkins LTS (Long Term Support). Why do you think I favour LTS over the latest Jenkins release? When would it be more appropriate to use the latest Jenkins release?

I favour Jenkins LTS (Long Term Support) because it provides a stable and thoroughly tested version of Jenkins that receives critical updates, bug fixes, and security patches over a longer period. LTS versions are ideal for production environments where reliability, security, and stability are prioritized. They are less likely to introduce breaking changes and are better suited for enterprise-level or long-running projects where downtime or unexpected issues could be costly.

The latest Jenkins release, on the other hand, includes the newest features, improvements, and bug fixes. It might be more appropriate to use the latest release in development environments or when we need access to cutting-edge features that aren’t yet available in the LTS version. However, it may be less stable and could introduce new bugs or compatibility issues, which makes it less ideal for production use.

**Describe the difference between a job/project and a build**

Jenkins, the terms "job/project" and "build" refer to distinct concepts:

Job/Project:

* + A job or project is a configured task or pipeline in Jenkins that defines the steps for building, testing, and deploying your software. It includes settings like the source code repository, build triggers, environment variables, and post-build actions.
  + Types of jobs include Freestyle, Pipeline, Multi-configuration, etc.

Build:

* + A build is a specific execution or run of a Jenkins job. It refers to the actual process where Jenkins compiles the code, runs tests, and performs any other tasks defined in the job configuration. Each build is a unique instance of a job, and Jenkins maintains a history of all builds for a job, including logs and results (pass/fail).

**What is a workspace?**

A workspace in Jenkins is a directory on the Jenkins server or agent where the files for a specific job's build are stored temporarily during its execution. It holds the job's source code, compiled files, test results, and any other artifacts needed for the build process.

Each job has its own dedicated workspace, ensuring that files from different jobs or builds do not interfere with each other. The workspace is created when a build starts and can be cleaned up either automatically or manually after the build is completed.

Key roles of the workspace include:

* Holding the files checked out from the source control system (e.g., Git).
* Storing files generated or required during the build process, like logs, binaries, or test results.
* Serving as a temporary location for running build scripts or compiling code.

What is a build trigger and how would you use it in a job?

Common Types of Build Triggers:

1. Source Code Change (SCM Polling):
   * Jenkins checks the version control system (e.g., Git, SVN) for changes at regular intervals. If changes are detected, the build is triggered.
   * Example: Configure SCM polling to check for new commits every 15 minutes and trigger a build whenever a change is detected.
2. Build after another project is built:
   * A job can be triggered to start automatically after another job completes (either successfully or regardless of the result). This is useful for creating build pipelines.
   * Example: A deployment job could trigger automatically after a successful build job.
3. Scheduled Builds (Cron):
   * Jenkins can trigger jobs on a specified schedule using a cron-like syntax (e.g., nightly builds).
   * Example: Configure a job to build every day at midnight: H 0 \* \* \*.
4. GitHub or GitLab Webhooks:
   * Jenkins can be configured to listen for webhooks from GitHub or GitLab, triggering a build whenever a commit is pushed, or a pull request is created.
   * Example: Trigger a build whenever a developer pushes code to the main branch.
5. Manual Build:
   * Sometimes, we may want a job to be triggered only when manually started by a user through the Jenkins UI.

How to Use a Build Trigger in a Job:

1. Go to your job configuration.
2. Scroll down to the Build Triggers section.
3. Select the desired trigger option(s), such as "Poll SCM" or "Build Periodically."
4. Configure the specifics of the trigger, like the schedule or polling interval.

**What is a parameterized job and when would you use it?**

parameterized job in Jenkins is a job that accepts inputs or parameters when it's triggered. These parameters can be used to customize the behaviour of the build based on the inputs provided at the time of execution.

Types of Parameters:

1. String Parameter: A simple text input that can be passed to the job (e.g., a version number, environment name).
2. Choice Parameter: Allows the user to select from a predefined list of options.
3. Boolean Parameter: A checkbox (true/false) option.
4. File Parameter: Enables the upload of a file that can be used during the build.
5. Password Parameter: Accepts sensitive information like passwords, which are masked for security.
6. Run Parameter: Refers to another build that can be used as an input.

**When to Use a Parameterized Job:**

* Dynamic Builds: When the same job needs to be run with different configurations, environments, or input values. For example, deploying an application to different environments (e.g., dev, staging, production) by passing the environment name as a parameter.
* Branch-Specific Builds: Running a build for different branches of the same repository by passing the branch name as a parameter.
* Version-Controlled Builds: Triggering a build for a specific version or commit ID by providing the version as an input parameter.
* Conditional Behaviour: Executing specific parts of the build process based on a parameter, such as skipping tests in a quick build by passing a Boolean parameter.

**What is meant by Pipeline as Code and Infrastructure as Code?**

Pipeline as Code refers to the practice of defining the entire build, test, and deployment process of a software application in code, typically using a version-controlled configuration file. In Jenkins, this is done using a Jenkins file, which contains all the steps of the CI/CD pipeline in a scripted or declarative format.

This approach ensures that the pipeline configuration is:

* Version-controlled: Stored alongside the source code, making it easy to track changes, collaborate, and rollback to previous versions.
* Reusable: Once defined, the pipeline can be reused across different environments or projects.
* Automated: The entire process is executed automatically when code changes occur, ensuring consistency in how the pipeline is run.

Example in Jenkins (Declarative Pipeline):

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building...'

// Build commands here

}

}

stage('Test') {

steps {

echo 'Testing...'

// Test commands here

}

}

stage('Deploy') {

steps {

echo 'Deploying...'

// Deployment commands here

}

}

}

}

This Jenkins file defines the stages (build, test, deploy) of the CI/CD pipeline.

Benefits of Pipeline as Code:

* Consistency across environments.
* Easily maintained, versioned, and auditable.
* Collaboration-friendly since the pipeline configuration is part of the source code repository.

Infrastructure as Code (IaC):

Infrastructure as Code refers to the process of managing and provisioning computing infrastructure (e.g., servers, networks, databases, etc.) using machine-readable configuration files or scripts, instead of manually configuring hardware or infrastructure via an interface. Tools like Terraform, Ansible, CloudFormation, or Chef are commonly used for IaC.

With IaC, the entire infrastructure setup (virtual machines, networking, storage) is written as code and can be versioned, reused, and automated, just like software code.

Example of IaC with Terraform (Creating an AWS EC2 instance):

provider "aws" {

region = "us-west-2"

}

resource "aws\_instance" "example" {

ami = "ami-0c55b159cbfafe1f0"

instance\_type = "t2.micro"

}

This Terraform script defines the configuration for provisioning an EC2 instance on AWS.

Benefits of Infrastructure as Code:

* Consistency: Ensures identical environments are created every time, reducing the chance of configuration drift.
* Automation: Automates provisioning, scaling, and management of infrastructure, reducing manual intervention.
* Version Control: Infrastructure changes are tracked and can be rolled back easily.
* Reusability: Infrastructure definitions can be reused across environments (development, staging, production).

**What are the benefits of using pipelines over normal build jobs in Jenkins?**

Code-Based Definition (Pipeline as Code):

* Pipelines allow you to define the build, test, and deployment stages using a code-based configuration (typically through a Jenkinsfile), which can be version-controlled.
* This provides traceability, version history, and easy collaboration since the pipeline is part of the source code repository.

Complex Workflows:

* Pipelines support complex workflows that involve multiple stages, parallel executions, conditionals, loops, and error handling.
* Traditional jobs are limited to linear build flows, while pipelines can handle intricate multi-step workflows and automate entire CI/CD processes.

Durability:

* Pipelines are designed to survive Jenkins server restarts. If the Jenkins master goes down, the pipeline can resume from where it left off when the server is back online.
* Normal build jobs do not have this durability, meaning if the server fails during a job, it must be restarted manually.

Flexibility and Scalability:

* Pipelines can be written to accommodate different environments, stages, or parameters, making them highly flexible and scalable for different projects and use cases.
* Normal build jobs are less flexible and can become cumbersome when scaling with multiple environments, branches, or configurations.

Integrated Continuous Delivery (CD) Support:

* Pipelines natively support continuous delivery by allowing developers to implement multiple stages (build, test, deploy) as part of the same job.
* Normal build jobs usually need post-build triggers or separate jobs to handle this, leading to a more fragmented process.

Pipeline Stages:

* Pipelines are organized into discrete stages (e.g., build, test, deploy), which allow for better visualization and organization of the CI/CD workflow.
* With normal jobs, there's no such clear breakdown of stages within a single job.

Error Handling and Retry Logic:

* Pipelines offer built-in support for error handling, allowing you to define what happens if a stage fails, such as retrying the step or sending notifications.
* Normal build jobs typically fail altogether if an error occurs, with no retry logic or post-failure handling.

Parallel Execution:

* Pipelines support running stages in parallel, which can significantly speed up the overall build and test process. For example, different test suites or environments can be run concurrently.
* Normal jobs do not easily support parallel execution without complex chaining or job coordination.

Environment and Parameter Management:

* Pipelines can pass environment variables, credentials, and parameters between stages, allowing for dynamic control of the build process.
* While normal jobs support parameters, managing and sharing them across multiple jobs is more cumbersome and error-prone.

Integration with Other Tools:

* Pipelines integrate seamlessly with other tools and plugins, such as version control (Git, GitHub, Bitbucket), Docker, and cloud platforms.
* Normal jobs can also use these integrations, but pipelines make it more efficient and organized, particularly for complex workflows.

Reduced Redundancy:

* Pipelines reduce the need for redundant jobs. Instead of creating multiple separate jobs for different branches or environments, you can create one pipeline and parameterize it to handle multiple use cases.
* Normal jobs often result in duplicate configurations when dealing with multiple branches or environments.

Better Visualization:

* Jenkins pipelines provide better visualization of the CI/CD process through the Pipeline View, where you can see the status of each stage (success/failure) at a glance.
* Normal jobs do not have such detailed visualization for the entire process flow.

**What is a Jenkins file?**

A Jenkins file is a text file that contains the definition of a Jenkins pipeline. It allows you to define the steps of your Continuous Integration (CI) and Continuous Delivery (CD) process as code, making the pipeline configuration version-controlled and easily reproducible.

The Jenkins file is typically stored in the root directory of your project repository (e.g., Git), and it defines the stages of your pipeline, such as building, testing, and deploying your application.

Types of Jenkins Pipelines Defined in Jenkins file:

Declarative Pipeline:

* + This is a simpler and more structured way to define pipelines, using a clear and concise syntax.
  + It organizes the pipeline into logical sections like agent, stages, steps, and post (for post-build actions).
  + Example of a Declarative Pipeline Jenkins file:

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building...'

// Add build commands here

}

}

stage('Test') {

steps {

echo 'Testing...'

// Add test commands here

}

}

stage('Deploy') {

steps {

echo 'Deploying...'

// Add deploy commands here

}

}

}

post {

always {

echo 'Cleaning up...'

}

}

}

Scripted Pipeline:

* + This is a more flexible and powerful method of writing pipelines but requires more verbose syntax.
  + It uses traditional Groovy scripting, which gives you full control over the pipeline's flow and logic.
  + Example of a Scripted Pipeline Jenkins file:

node {

stage('Build') {

echo 'Building...'

// Add build commands here

}

stage('Test') {

echo 'Testing...'

// Add test commands here

}

stage('Deploy') {

echo 'Deploying...'

// Add deploy commands here

}

}

**What are the differences between a declarative pipeline and a scripted pipeline?**

Declarative and scripted pipelines in Jenkins both serve the purpose of defining CI/CD processes, but they differ significantly in their syntax, structure, and usage. Here are the key differences:

Syntax and Structure:

* Declarative Pipeline:
  + Uses a more structured and simplified syntax.
  + Organized into specific sections like pipeline, agent, stages, steps, and post.
  + Easier to read and understand, especially for users new to Jenkins.

Example:

pipeline {

agent any

stages {

stage('Build') {

steps {

echo 'Building...'

}

}

stage('Test') {

steps {

echo 'Testing...'

}

}

}

}

Scripted Pipeline:

* + Uses traditional Groovy scripting syntax, which is more flexible but also more complex.
  + Allows for more advanced programming constructs, such as loops and conditionals.
  + Requires a deeper understanding of Groovy and Jenkins internals.

Example:

node {

stage('Build') {

echo 'Building...'

}

stage('Test') {

echo 'Testing...'

}

}

Ease of Use:

* Declarative Pipeline:
  + Generally easier to use for most users, especially those who are not familiar with programming.
  + Enforces best practices through its structure, reducing the likelihood of errors.
* Scripted Pipeline:
  + More powerful and flexible but requires a good understanding of Groovy and Jenkins.
  + Offers more control over the pipeline execution flow, making it suitable for complex scenarios.

Error Handling:

* Declarative Pipeline:
  + Provides built-in support for error handling and post-build actions via the post section (e.g., always, success, failure).
* Scripted Pipeline:
  + Requires manual handling of errors using try-catch blocks and other Groovy constructs.

Pipeline Features:

* Declarative Pipeline:
  + Supports features like parallel stages and simpler syntax for defining input parameters and environment variables.
  + Designed to work with Jenkins's declarative syntax enhancements, making it easier to leverage new features.
* Scripted Pipeline:
  + Provides complete flexibility to create any logic needed within the pipeline but may involve more complex coding to implement features like parallel execution.

Best Use Cases:

* Declarative Pipeline:
  + Best suited for straightforward CI/CD processes that follow standard practices and do not require advanced scripting.
  + Ideal for teams looking for ease of use and maintainability.
* Scripted Pipeline:
  + Better for complex scenarios requiring advanced logic, dynamic configurations, or when integrating with other Groovy scripts.
  + Suitable for experienced developers who need fine-grained control over the pipeline behaviour.

**Which are the advantages of using Blue Ocean?**

Blue Ocean is a user interface for Jenkins designed to provide a more modern, intuitive, and streamlined experience for users working with Jenkins pipelines. Here are some key advantages of using Blue Ocean:

Improved User Experience:

* Blue Ocean offers a visually appealing and user-friendly interface that simplifies navigation and enhances usability compared to the traditional Jenkins UI.
* It focuses on providing a clear visual representation of the pipeline stages, making it easier to understand the CI/CD workflow at a glance.

Pipeline Visualization:

* Blue Ocean provides advanced visualizations of pipelines, including clear stage views and timelines. This allows users to easily track the progress of builds, see which stages are running, and identify failures quickly.
* The visual representation helps teams understand complex pipelines and their relationships, improving collaboration and communication.

Simplified Pipeline Creation:

* Users can create and edit pipelines more easily using Blue Ocean’s intuitive interface, which supports visual editing and configuration.
* The integrated pipeline editor allows users to write and test their Jenkinsfiles directly in the UI, reducing the need for context switching.

Enhanced Collaboration:

* Blue Ocean supports better collaboration among team members by providing a shared view of the pipeline status and logs, making it easier to understand the CI/CD process and results.
* Features like customizable notifications and integrated GitHub or Bitbucket pull request handling enhance team collaboration.

Integration with Modern Development Tools:

* Blue Ocean integrates seamlessly with popular version control systems like GitHub and Bitbucket, allowing users to easily manage pull requests and trigger builds directly from those platforms.
* It provides features like automatic branch indexing, which helps teams keep track of changes in multiple branches effortlessly.

Simplified Pipeline Management:

* Blue Ocean groups related builds and allows for easy management of pipelines, making it simple to find and navigate through different branches and pull requests.
* Users can quickly see the status of various builds across different branches, enhancing visibility.

Built-in Notifications:

* Blue Ocean includes built-in notification support, allowing teams to receive real-time updates on pipeline events directly within the UI or through integrations with communication tools like Slack.

Customizable Views:

* Users can customize their dashboard views to focus on the pipelines and projects that are most relevant to them, improving the overall workflow and efficiency.

Focused on Pipelines:

* Blue Ocean is designed specifically for pipeline-based workflows, making it more suitable for modern DevOps practices than the traditional Jenkins interface, which supports a broader range of job types.

Extensibility:

* Blue Ocean supports plugins and can be extended to incorporate additional functionality as needed, making it adaptable to various team needs.

**What is a multibranch pipeline?**

A multibranch pipeline in Jenkins is a specialized type of pipeline that automatically detects, manages, and executes builds for multiple branches in a version control system (VCS) repository. This allows teams to have separate builds for different branches (like feature, development, and main) without the need to manually create individual jobs for each branch.

Key Features of Multibranch Pipelines:

Automatic Discovery:

* + Jenkins automatically discovers branches in the specified repository and creates a pipeline for each branch that contains a Jenkins file.
  + As branches are created, deleted, or updated in the repository, Jenkins reflects these changes automatically.

Branch-Specific Configuration:

* + Each branch can have its own Jenkins file, allowing for customized pipeline logic tailored to the needs of that specific branch.
  + This is useful for managing different stages or deployment processes for feature branches versus production branches.

Pull Request Support:

* + Multibranch pipelines can be configured to build pull requests from the repository. When a pull request is created or updated, Jenkins can trigger the pipeline to run tests and checks on the proposed changes.
  + This helps ensure that new code integrates smoothly with the existing codebase before merging.

Unified Management:

* + Multibranch pipelines consolidate the management of multiple branches under a single job in the Jenkins UI, making it easier to view and control builds across branches.
  + Users can see the status of each branch's build, logs, and history in one place.

Parallel Execution:

* + Builds for different branches can run in parallel, speeding up the CI/CD process and allowing multiple features to be developed and tested simultaneously.

Simplified Configuration:

* + Instead of creating and maintaining separate jobs for each branch, multibranch pipelines reduce redundancy and streamline configuration management.

**What is a pull request?**

A pull request (PR) is a feature in version control systems, particularly in platforms like GitHub, Bitbucket, and GitLab, that allows developers to propose changes to a codebase. Here’s a breakdown of what a pull request involves:

Key Aspects of a Pull Request:

Proposing Changes:

* + A pull request is created when a developer wants to merge changes made in a separate branch (often a feature or bug-fix branch) into another branch, typically the main or development branch.
  + This allows team members to review the changes before they are integrated into the main codebase.

Code Review:

* + Once a pull request is submitted, other team members can review the code changes, provide feedback, and suggest modifications.
  + This collaborative review process helps ensure code quality and adherence to coding standards.

Discussion:

* + Pull requests provide a platform for discussion regarding the proposed changes. Team members can comment on specific lines of code, ask questions, or request changes.
  + This fosters collaboration and knowledge sharing within the team.

Testing:

* + Many teams integrate automated testing into the pull request workflow. This can include running unit tests, integration tests, and other quality checks to ensure the proposed changes do not introduce bugs.
  + Continuous Integration (CI) tools often trigger builds and tests automatically when a pull request is created or updated.

Approval and Merging:

* + After the review process, once all feedback has been addressed and tests have passed, the pull request can be approved.
  + The changes are then merged into the target branch, completing the integration process.

Closing:

* + After merging, the pull request is typically closed. If the changes are not going to be merged (for example, if the feature is abandoned), the pull request can also be closed without merging.