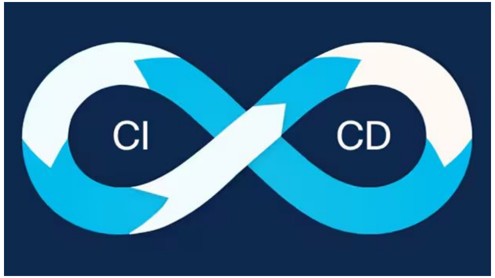
Jenkins

Course Content

- What is Continuous Integration, Continuous Deployment & Continuous Delivery
- Introduction to Jenkins
- Jenkins Installation & Setup
- Understanding Jenkins Builds and Jobs
- Jenkins and Source Control
- Working with Jenkins Plugins
- Jenkins Pipelines - Declarative & Scripted
- Jenkins and Test Automation
- Reporting in Jenkins
- Jenkins Best Practices and Tips

Understanding CI, CD, and Their Role in Software Development

- **Continuous Integration (CI)**: Automating the integration of code changes from multiple developers into a shared repository.
- **Continuous Deployment (CD)**: Automatically deploying every change that passes the tests to production.
- **Continuous Delivery**: Ensuring that code is always in a deployable state and can be released on demand.



Continuous Integration (CI)

- CI is a **development practice** where developers **frequently merge** their code changes into a central repository.
- Each integration triggers an **automated build** and test cycle to detect issues early.
- Ensures that **code conflicts** are found and resolved quickly.

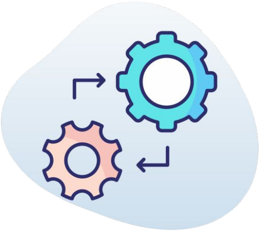
Benefits of Continuous Integration:

- **Faster detection** of integration bugs.
- **Reduced merge conflicts**.
- **Improved collaboration** between team members.

Continuous Deployment (CD)

- Continuous Deployment is the practice of **automatically deploying** every code change that passes automated tests to **production**.
- Eliminates the need for manual intervention in the deployment process.

Benefits of Continuous Deployment:



- **Faster release cycles**, enabling frequent updates.
- **No human error** in deployment.
- Real-time feedback from production, allowing faster **iteration** and **fixes**.

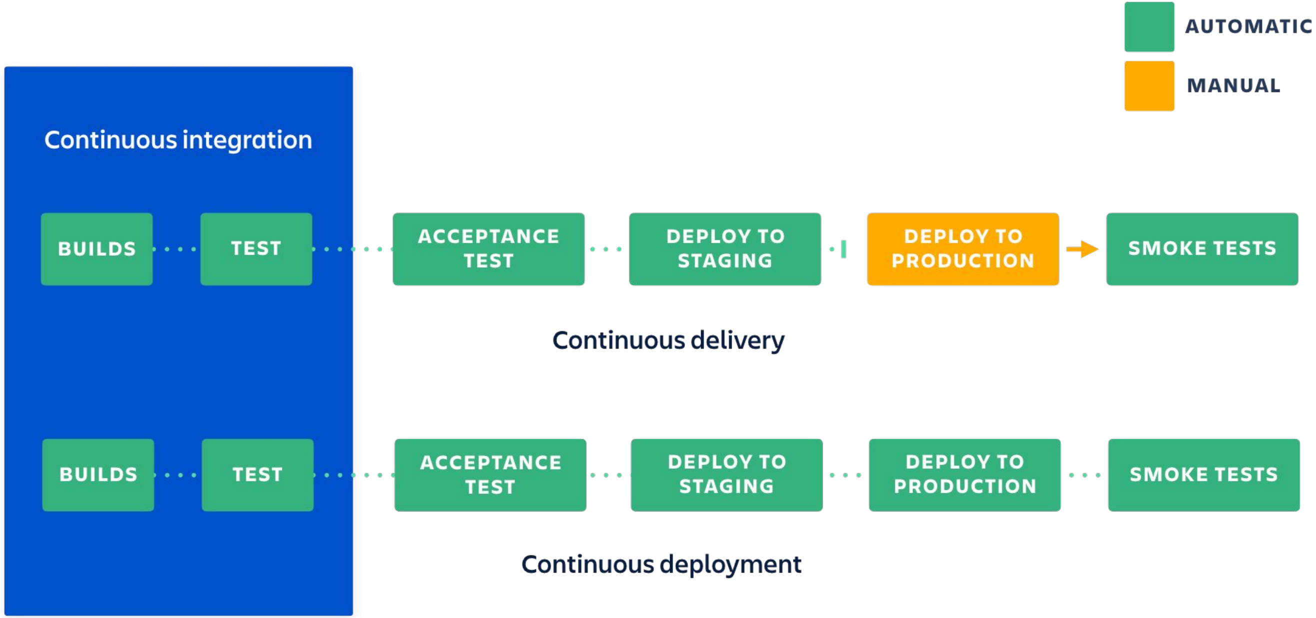
Continuous Delivery (CD)

- Continuous Delivery ensures that the codebase is always in a **deployable state**.
- It involves deploying to a **staging** or **production-like** environment after every build, but **manual approval** is required to release it to production.

Benefits of Continuous Delivery:

- Continuous Delivery stops short of automatic deployment to production, allowing for a **controlled release**.
- Provides flexibility to release whenever necessary.

Continuous Integration vs. Delivery vs. Deployment



🟩 AUTOMATIC

🟧 MANUAL

Continuous integration

BUILDS · · · TEST · · · ACCEPTANCE TEST · · · DEPLOY TO STAGING · | DEPLOY TO PRODUCTION → SMOKE TESTS

Continuous delivery

BUILDS · · · TEST · · · ACCEPTANCE TEST · · · DEPLOY TO STAGING · · · DEPLOY TO PRODUCTION · · · SMOKE TESTS

Continuous deployment

Image Reference:
https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Continuous Integration vs. Delivery vs. Deployment

Aspect	Continuous Integration (CI)	Continuous Delivery (CD)	Continuous Deployment (CD)
Definition	Automating the integration of code changes into a shared repository.	Ensures code is always in a deployable state, but manual approval is needed to release to production.	Automatically deploys code to production after passing all tests.
Goal	Detect and fix integration issues early.	Ensure that the codebase is always ready for deployment.	Fully automate the release process to quickly deliver new features.
Deployment	No automatic deployment; focuses on testing and merging code.	Deployment to staging or production-like environment; manual approval required for production.	Automatic deployment to production without manual intervention.
Frequency of Deployment	Not applicable.	As often as needed, but requires manual release.	Deploys frequently, after every successful build.

Continuous Integration vs. Delivery vs. Deployment

Aspect	Continuous Integration (CI)	Continuous Delivery (CD)	Continuous Deployment (CD)
Test Automation	Essential; integrates unit, integration, and other automated tests.	Requires extensive automated testing to ensure code is ready for deployment.	Requires thorough automated testing (unit, integration, etc.) to ensure safety.
Main Benefit	Early detection of issues, faster feedback to developers.	Code is always ready for release, allowing flexibility for the team.	Fast delivery of changes to production without delay.
Risk Level	Low; mainly focused on code testing and integration.	Medium; requires confidence in code readiness but allows manual oversight.	High; needs robust testing to ensure no issues are pushed to production.
Manual Intervention	No, fully automated tests run after code changes are merged.	Yes, manual approval required before deployment to production.	No, deployment to production is automatic after passing tests.
Ideal For	Teams with frequent commits needing fast feedback on integration.	Teams wanting fast release cycles but maintaining manual control over production releases.	Teams seeking to fully automate and streamline their release process.

Benefits of CI/CD in Test Automation

- CI/CD **accelerates feedback loops** by running automated tests continuously.
- Reduces the risk of bugs making it to production due to **early detection**.
- Ensures **consistent testing environments**, reducing the potential for flaky tests.
- **Enables rapid delivery** of new features and fixes with confidence.

Popular Tools for CI/CD:

- **CI Servers**: Jenkins, CircleCI, Travis CI, GitHub Actions, GitLab CI/CD.

Introduction to Jenkins

- Jenkins is an **open-source automation server**.
- It enables **Continuous Integration (CI)** and **Continuous Delivery (CD)**.
- Jenkins helps to **automate the build, test, and deploy** phases of the software development lifecycle.

Key Benefits of Jenkins for Automation:

- **Highly extensible**: Jenkins has over 1,800 plugins for integrating with other tools and technologies.
- **Easy to set up**: Jenkins provides a simple web-based GUI for configuration.
- **Automates repetitive tasks**: Automates testing, building, and deployment, reducing manual effort.
- **Supports a wide range of languages**: Compatible with Java, Python, JavaScript, and more.

Key Features of Jenkins

- **Automated Builds**: Automatically compile and build projects when code changes.
- **Automated Testing**: Integrates with test automation tools (Selenium, JUnit, Cucumber).
- **Plugins**: Extensible with a variety of plugins to support different build, test, and deployment tools.
- **Distributed Builds**: Supports **Master-Slave** architecture to run jobs on multiple nodes.

Jenkins Architecture

Master-Slave Architecture:

- **Jenkins Master**: Handles scheduling of build jobs, dispatching builds to slaves, and monitoring progress.
- **Jenkins Slaves (Agents)**: Execute build jobs assigned by the master.

This architecture enables **distributed builds**, improving speed and scalability.

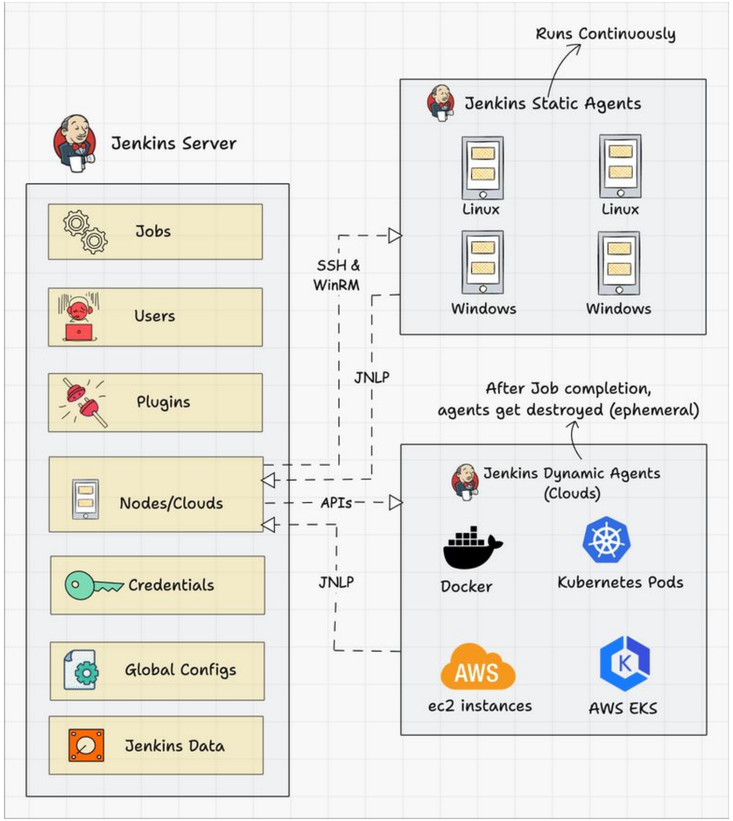


***Image Reference:** https://devopscube.com/jenkins-architecture-explained/*

Jenkins Workflow

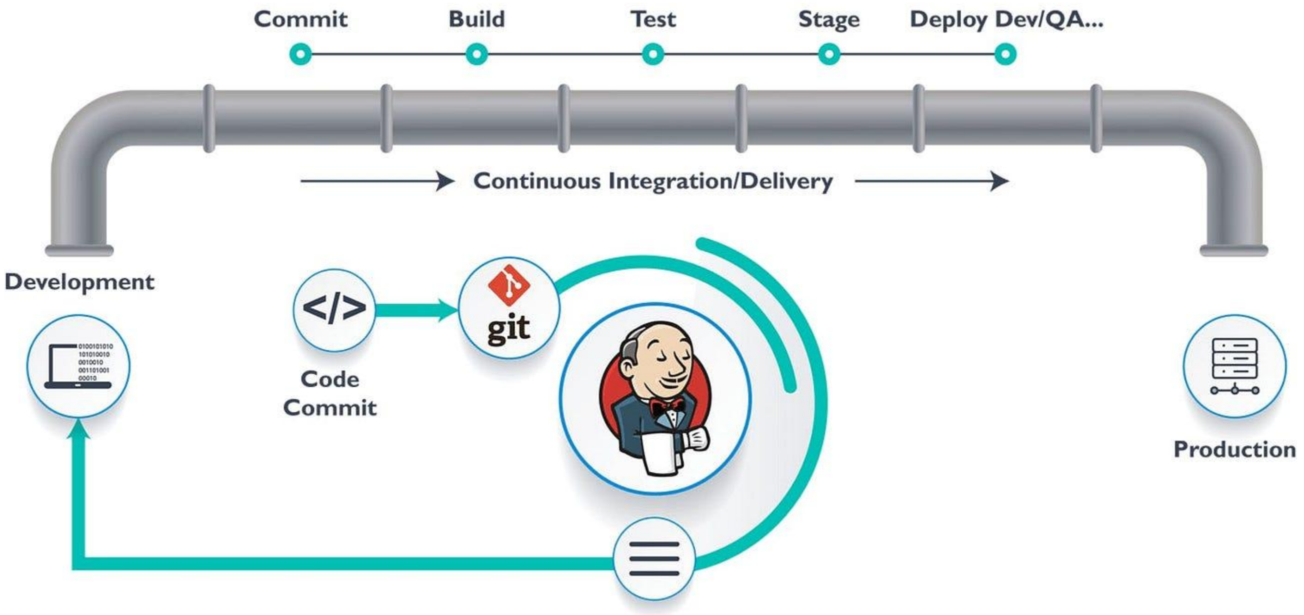


Image Reference: *https://medium.com/@faisalkuzhan/day-37-90-jenkins-pipeline-4936c3c557eb*

Jenkins Installation - Prerequisites:

1. Java should be installed in the system

```
java -version
openjdk version "11.0.19" 2023-04-18
OpenJDK Runtime Environment Temurin-11.0.19+7 (build 11.0.19+7)
OpenJDK 64-Bit Server VM Temurin-11.0.19+7 (build 11.0.19+7, mixed mode)
```

2. Maven should be installed in the system

```
mvn -version
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: ~/softwares/apache-maven-3.8.4
```

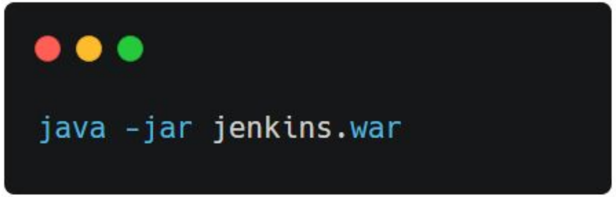
3. GIT should be installed in the system

```
git --version
git version 2.45.2.windows.1
```

Jenkins Installation

There are many ways to install Jenkins. Simplest method is to install it via WAR file.

- Visit the Jenkins download page: https://www.jenkins.io/download/
- Under **LTS (Long-Term Support)**, download the latest **jenkins.war** file.
- Save the file to a folder on your local machine (e.g., C:\jenkins\).
- Navigate to the folder and run below command



```
java -jar jenkins.war
```

- Jenkins will start on **port 8080** by default. Once Jenkins starts, open your browser and go to:

  http://localhost:8080.

Jenkins Installation (Contd.)

Perform these additional tasks to complete the installation and setup.

- Unlock Jenkins for the first time use
- Install the suggested plugins
- Create Admin user

Jenkins System Configuration

- **System Configuration** in Jenkins involves setting up **global settings** that affect how Jenkins operates.
- This includes configuring **system-wide tools**, **global environment variables**, and defining **workspace locations** for Jenkins jobs.
- Proper configuration is key to ensuring smooth operation of Jenkins pipelines.

Accessing System Configuration:

- From the Jenkins dashboard, click on **Manage Jenkins** in the left sidebar.
- Select **System** from the **System Configuration** section.
- This will take you to the main system configuration page.

Global Configuration Settings

Home Directory:

- Jenkins stores all build-related data in its **home directory** (e.g., C:\Users\<User>\.jenkins).
- You can change the location if needed to a larger storage space.

Jenkins URL:

- Configure the base URL for your Jenkins instance, which will be used in notifications and job links (e.g., http://localhost:8080 or custom domain).

System Message:

- Add a message that appears on the Jenkins dashboard for all users (useful for system-wide notifications).

Configuring Global Tools

JDK (Java Development Kit):

- Jenkins needs Java to run builds for Java projects.
- Under **Global Tool Configuration**, set the location of the **JDK** or allow Jenkins to automatically install it.

Maven:

- Configure **Apache Maven** to automate building Java applications.
- Specify the location or let Jenkins download and install it when needed.

Git:

- Jenkins uses **Git** for source control integration.
- Set the path to the **Git executable** or enable automatic installation.

Other Build Tools:

- You can configure additional build tools like **Ant**, **NodeJS**, or **Python** as needed.

Managing Jenkins Plugins

Plugin Manager:

- Navigate to **Manage Jenkins** > **Manage Plugins** to install, update, or configure Jenkins plugins.
- Plugins are critical for integrating with tools like **Maven**, **Docker**, **Kubernetes**, and more.

Installation:

- Search for a plugin and click **Install** without restart, or **Install and restart** to apply changes immediately.

Update Plugins:

- Regularly check for plugin updates to keep your Jenkins instance secure and feature-complete.

Other Configurations

- System Log Configuration
- Configuring Email notifications
- Workspace and Build directories
- Configuring System Security

Jenkins Jobs and Builds

- **Jenkins Jobs** represent tasks or processes to be automated, such as building code, running tests, or deploying applications.
- A **Build** is a specific instance or execution of a Jenkins job. Builds are typically triggered by events like code changes or scheduled times.

Different Types of Jenkins Jobs:

1. Freestyle Project
2. Pipeline Job
3. Multibranch Pipeline
4. Multiconfiguration Project
5. External Job
6. Folder

Creating a Jenkins Freestyle Job

1. **Open Jenkins Dashboard**:
   - Click on **New Item** in the Jenkins dashboard.
2. **Enter Job Name**:
   - Provide a meaningful job name (e.g., Test_Project).
3. **Select Job Type**:
   - Choose **Freestyle Project** and click **OK**.
4. **Configure the Job**:
   - Define source code management (e.g., Git), build triggers (e.g., scheduled builds), and build steps (e.g., running scripts or compiling code).
5. **Save the Job**:
   - Once the job is configured, click **Save** to create the job.

Jenkins Build Lifecycle

A **build** in Jenkins refers to the execution of a job.

The build lifecycle includes:

1. **Triggering the Job**:
   - Can be triggered manually, by code commits, or via scheduling.
2. **Source Code Checkout**:
   - Jenkins fetches code from the configured repository.
3. **Executing Build Steps**:
   - The build executes tasks such as compiling code, running tests, and generating artifacts.
4. **Post-Build Actions**:
   - Actions such as sending notifications, archiving artifacts, or triggering other jobs.
5. **Build Completion**:
   - The build ends with either a **success**, **unstable**, or **failed** status.

Configuring Build Triggers

Manual Trigger:

- Click **Build Now** to run the job manually.

SCM Polling:

- Jenkins can poll the source code management (SCM) tool (e.g., Git) at regular intervals to detect code changes and trigger builds.

Webhook/Push Notification:

- Set up webhooks to automatically trigger a build when there is a code push or pull request (e.g., GitHub webhook).

Scheduled Builds:

- Use a **Cron** expression to schedule jobs to run at specific intervals (e.g., nightly builds).
- Example: `H 0 * * 1-5` (runs at midnight Monday to Friday).

Jenkins Build Statuses

1. **Success**:
   - The build completes all tasks successfully with no errors.
2. **Unstable**:
   - The build completes but with some warnings or failing tests.
3. **Failure**:
   - The build fails due to compilation errors, failed tests, or other issues.
4. **Aborted**:
   - The build was manually stopped or interrupted.
5. **Not Built**:
   - The job configuration is incomplete or skipped in a pipeline.

Monitoring Builds and Build History

Tracking Build Progress and Results

1. **Build Console Output**:
   - View real-time build progress and logs for debugging and monitoring purposes.
2. **Build History**:
   - Jenkins retains a history of all builds, accessible through the job page.
   - You can check past build results and access logs, artifacts, or test results.
3. **Trend Reports**:
   - Jenkins plugins like **Test Results Analyzer** provide visual trends of build health, test failures, and build duration over time.

Post-Build Actions

What Happens After a Build Completes?

After the build completes, Jenkins can trigger **post-build actions**, including:

1. **Archiving Artifacts**:
   - Store build outputs (e.g., binaries, reports) for future use.
2. **Sending Notifications**:
   - Email build results to developers, or use integrations like Slack/Teams for team alerts.
3. **Trigger Other Jobs**:
   - Jenkins can trigger downstream jobs or pipelines (e.g., deployment jobs) once a build completes.
4. **Publish Test Reports**:
   - Automatically display test results, coverage reports, or other metrics.

Jenkins and Source Control

What is Source Control in Jenkins?

- **Source Control (Version Control)** manages code changes, tracks versions, and allows multiple developers to collaborate on a project.
- **Jenkins integrates with version control systems** (e.g., Git, Subversion, Mercurial) to automatically build, test, and deploy code whenever changes are made.
- Source control is a critical part of Continuous Integration (CI), ensuring that each change is tested and validated before deployment.

Popular Version Control Systems Integrated with Jenkins

- Git
- Subversion (SVN)
- Mercurial
- Bitbucket
- GitHub

Installing Source Control Plugins

1. **Install Plugins**:
   - Go to **Manage Jenkins** > **Manage Plugins** and search for the relevant source control plugin (e.g., Git Plugin, Subversion Plugin).
2. **Configure Source Control**:
   - After installing the plugin, configure the source control settings by adding the repository URL and credentials.
3. **Set Credentials**:
   - Configure Jenkins to authenticate with the source control system using **SSH keys**, **username/password**, or **OAuth tokens**.

Configuring Jenkins to Use Git

1. **Install Git Plugin**:
   - Install the **Git Plugin** from the Plugin Manager.
2. **Create a New Job**:
   - Go to **New Item** and create a **Freestyle** or **Pipeline** job.
3. **Set the Repository URL**:
   - Under the **Source Code Management** section, select **Git** and enter the **repository URL** (e.g., https://github.com/sample-repo.git).
4. **Add Credentials**:
   - Add **SSH keys** or **personal access tokens** to authenticate with the Git repository.
5. **Specify Branches**:
   - Specify which branch Jenkins should build from (e.g., main or develop).
6. **Build Triggers**:
   - Set up **polling** or **webhooks** to trigger the build when changes are pushed to the repository.

Triggering Jenkins Jobs via SCM Changes

1. **SCM Polling**:
   - Jenkins can periodically poll the repository for changes. When a new commit is detected, it triggers the job.
   - Configure the **polling frequency** using a cron expression (e.g., `H/15 * * * *` for every 15 minutes).
2. **Webhooks**:
   - **GitHub Webhooks** or **Bitbucket Webhooks** allow Jenkins to react instantly to changes.
   - Webhooks notify Jenkins immediately when code is pushed, creating faster feedback loops for continuous integration.
   - Configured in the repository settings on GitHub or Bitbucket.

Source Control Best Practices for Test Automation

- **Version Your Tests**: Store test scripts alongside application code to ensure that changes to both are tracked together.
- **Run Tests on Every Commit**: Automatically trigger test suites on every commit to detect defects early.
- **Use Branch-Specific Tests**: Run different sets of tests based on the branch (e.g., run integration tests on the main branch but unit tests on feature branches).
- **Maintain Test Artifacts**: Archive test results and logs in source control or cloud storage for future reference and debugging.

Build Parameters in Jenkins

- **Build Parameters** allow you to pass dynamic values to Jenkins jobs at runtime.
- They enable the customization of build processes without changing the job configuration.
- Commonly used to set up different configurations, environments, or to trigger builds with specific conditions.

Types of Build Parameters

1. **String Parameter**:
   - A simple text input field for user-defined strings.
2. **Boolean Parameter**:
   - A checkbox to toggle a true/false value.
3. **Choice Parameter**:
   - A dropdown list for selecting from predefined options.
4. **File Parameter**:
   - Allows users to upload a file that the build can use.
5. **Password Parameter**:
   - A text field that hides input, ideal for sensitive data.

Adding Build Parameters to a Job

1. **Create a New Job** or configure an existing one.

2. Go to the **Job Configuration** page.

3. Under the **General** section, check the option **"This project is parameterized."**

4. Click **Add Parameter** and select the type of parameter you want to add.

5. Fill in the required details:

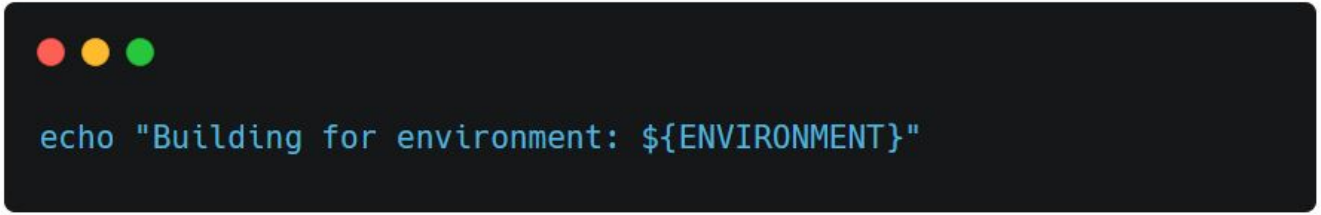
    a. **Name**: Identifier for the parameter.

    b. **Default Value**: (optional) A default value for the parameter.

Using Build Parameters in Build Steps

Accessing Build Parameters in Your Job:

- Use the syntax ${PARAMETER_NAME} or %PARAMETER_NAME% to reference build parameters in build steps, environment variables, or scripts.
- Example:



```
echo "Building for environment: ${ENVIRONMENT}"
```

- This would print the value of the ENVIRONMENT parameter during the build.

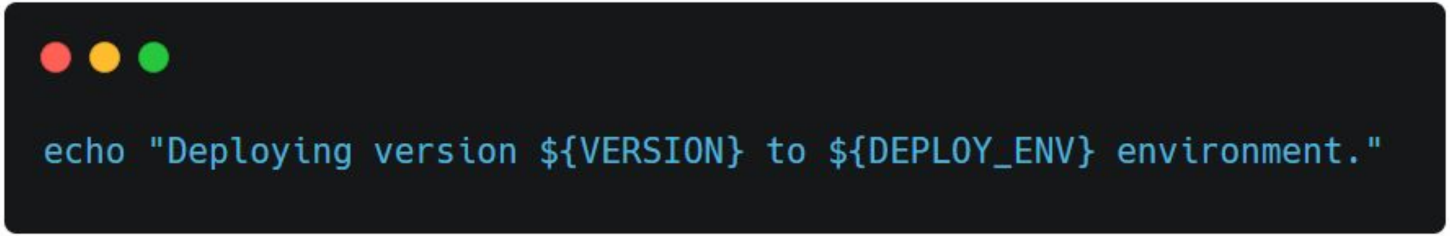
Example: String and Choice Parameters

String Parameter: For example, VERSION.

- Prompt: "Enter the version number:"

Choice Parameter: For example, DEPLOY_ENV.

- Choices: Development, Staging, Production.
- Accessing these in a build step:



```
echo "Deploying version ${VERSION} to ${DEPLOY_ENV} environment."
```

Build Parameter Usage Scenarios

1. **Environment-Specific Builds**:
   - Build and deploy applications for different environments (development, staging, production).
2. **Versioning**:
   - Allow users to specify the version of the application to build.
3. **Feature Toggles**:
   - Enable or disable specific features during builds using boolean parameters.
4. **File Uploads**:
   - Upload configuration files or artifacts needed for the build.

Jenkins Plugins

- **Jenkins Plugins** extend the functionality of Jenkins, allowing it to integrate with various tools, frameworks, and services.
- Thousands of plugins are available to automate tasks, manage jobs, visualize results, and integrate with CI/CD tools.
- Plugins are essential for scaling Jenkins to meet the needs of specific projects, especially for **test automation**.
- Access the Plugin Manager from **Manage Jenkins** > **Manage Plugins**.

Why Plugins Are Important for Test Automation

How Plugins Enhance Test Automation in Jenkins

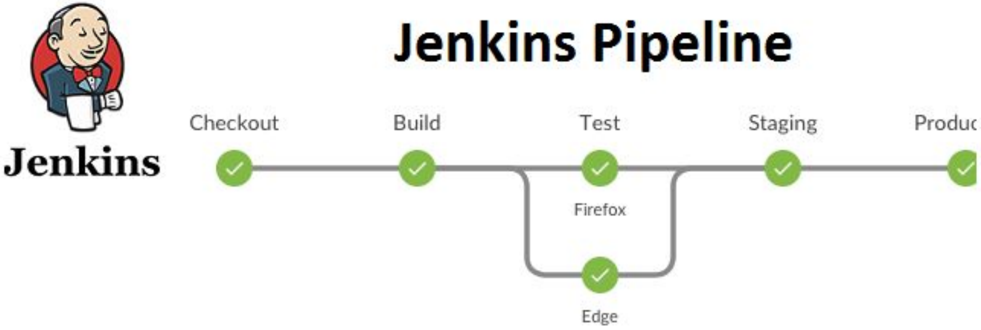
1. **Test Execution Integration**:
   - Plugins allow Jenkins to execute tests written in different frameworks (JUnit, TestNG, etc.).
2. **Reporting**:
   - Plugins generate test reports, visualize test results, and provide detailed insights into test failures.
3. **CI/CD Integration**:
   - Plugins help integrate testing into CI/CD pipelines, enabling faster feedback loops and early defect detection.
4. **Notification and Collaboration**:
   - Plugins facilitate sending notifications (email, Slack) and trigger alerts based on test results.

Key Plugins for Test Automation

1. **JUnit Plugin**:
    - This plugin **publishes test results** from **JUnit** or other xUnit test frameworks.
    - Displays test reports in the Jenkins UI, tracking test pass/fail history across builds.
2. **TestNG Results Plugin**:
    - Used to publish and display results for **TestNG**-based tests.
    - Displays detailed test reports, including passed, failed, and skipped tests.
3. **Maven Integration Plugin**:
    - Automatically builds and tests Maven projects, handling **build lifecycle**, **dependency management**, and **report generation**.
4. **HTML Publisher Plugin**:
    - Allows Jenkins to publish and display **custom HTML reports**, such as those generated by test automation frameworks (e.g., Selenium, JUnit).
5. **Allure Report Plugin**:
    - Integrates with **Allure** to generate beautiful, detailed reports for various test frameworks, including **JUnit**, **TestNG**, **Selenium**, and more.
    - Provides in-depth insights into test execution, including test duration, environment, and failure analysis.

Jenkins Pipelines

- **Jenkins Pipelines** allow you to define your entire CI/CD process as code.
- Pipelines automate tasks such as building, testing, and deploying applications in a structured and repeatable way.
- Two types of pipelines are supported in Jenkins:
  - **Declarative Pipelines**
  - **Scripted Pipelines**



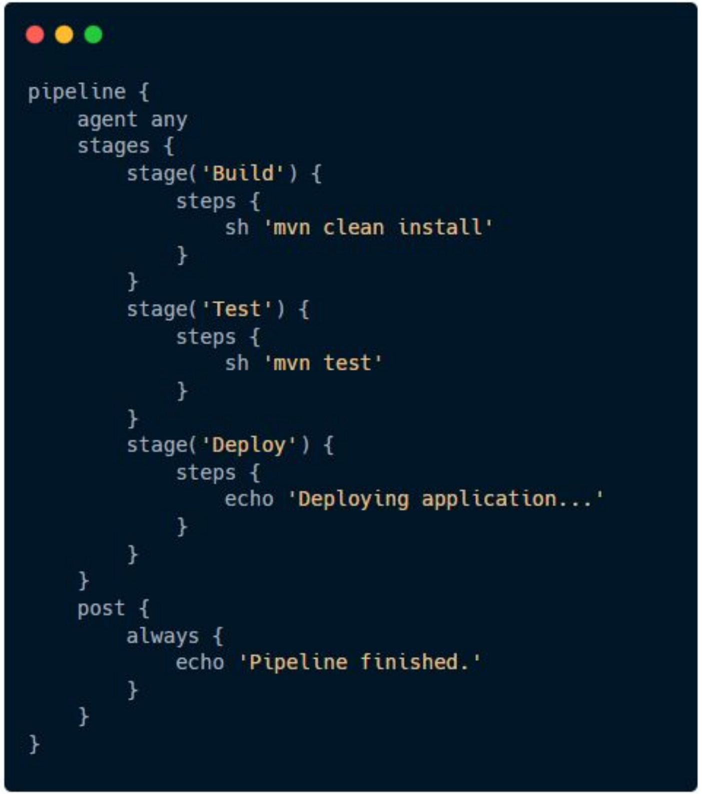
Declarative Pipelines Overview

- **Declarative Pipelines** provide a structured, simplified syntax for defining Jenkins pipelines.
- Uses a **predefined syntax** with specific sections such as pipeline, agent, stages, and post.
- Recommended for **most users** as it’s easier to read, write, and maintain.
- Enforces a **well-defined structure** with built-in validations, helping to prevent common errors.

Declarative Pipeline Structure

Key Sections:

1. agent: Defines where the pipeline will run.
2. stages: Contains multiple stages like Build, Test, and Deploy.
3. steps: The specific commands or actions to be executed in each stage.
4. post: Defines actions to run after the pipeline, such as notifications or cleanup.



```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying application...'
            }
        }
    }
    post {
        always {
            echo 'Pipeline finished.'
        }
    }
}
```

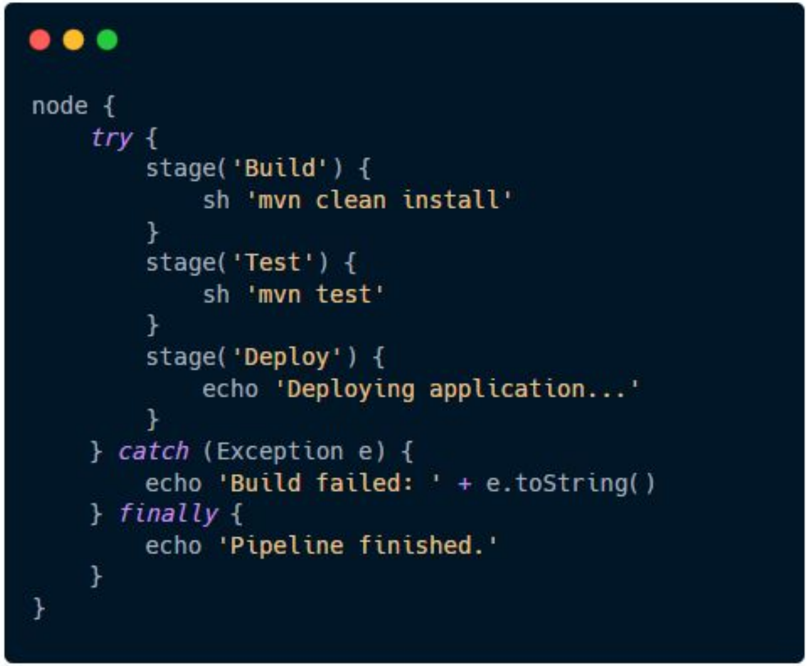
Scripted Pipelines Overview

- **Scripted Pipelines** use traditional **Groovy scripting** and offer **more flexibility** than Declarative Pipelines.
- Suitable for users who need advanced features and custom logic.
- Less structured and requires more manual management of steps and stages.
- Offers full access to the underlying **Jenkins API**, making it more powerful but harder to maintain.

Scripted Pipeline Structure

Key Sections:

1. node: Represents the machine Jenkins will run the pipeline on.
2. try/catch: Error handling mechanism for controlling failures.
3. Customizable stages and steps, written in Groovy.



```
node {
    try {
        stage('Build') {
            sh 'mvn clean install'
        }
        stage('Test') {
            sh 'mvn test'
        }
        stage('Deploy') {
            echo 'Deploying application...'
        }
    } catch (Exception e) {
        echo 'Build failed: ' + e.toString()
    } finally {
        echo 'Pipeline finished.'
    }
}
```

Key Differences: Declarative vs. Scripted

Feature	Declarative Pipeline	Scripted Pipeline
Syntax	Predefined, easier to read and write	Groovy-based, more flexible
Error Handling	Implicit, managed within post blocks	Requires explicit try/catch blocks
Structure	Strict structure with stages, steps	Free-form, can define custom logic
Suitability	Recommended for most users	Ideal for complex pipelines
Flexibility	Limited to predefined blocks	Full control via Groovy scripting
Learning Curve	Easier for beginners	Steeper, requires Groovy knowledge

When to Use Declarative vs. Scripted

Use Declarative Pipelines:

- When simplicity and readability are key.
- For **standard CI/CD pipelines** with well-defined stages like build, test, and deploy.
- If your team is new to Jenkins or scripting.

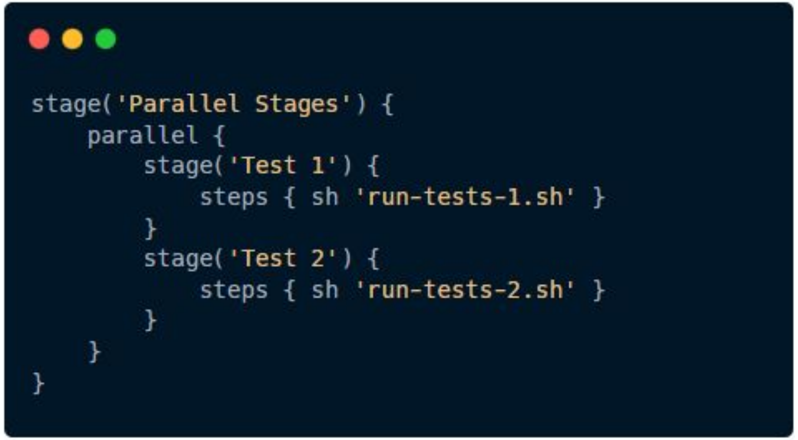
Use Scripted Pipelines:

- When you need **advanced logic**, such as looping, dynamic stages, or complex conditionals.
- For **complex workflows** that require extensive customization.
- If you need **fine-grained control** over pipeline execution.

Key Features of Declarative Pipelines

Parallel Execution:

- Run stages in parallel to speed up the pipeline.

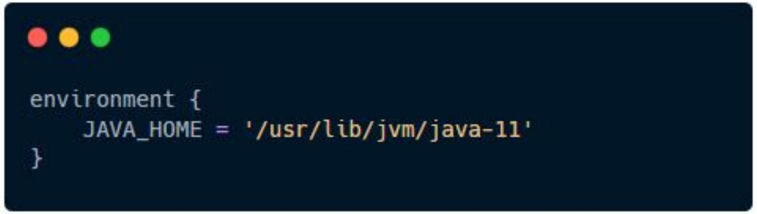


```
stage('Parallel Stages') {
    parallel {
        stage('Test 1') {
            steps { sh 'run-tests-1.sh' }
        }
        stage('Test 2') {
            steps { sh 'run-tests-2.sh' }
        }
    }
}
```

Key Features of Declarative Pipelines

Environment Variables:

- Declare environment variables to pass data between stages.

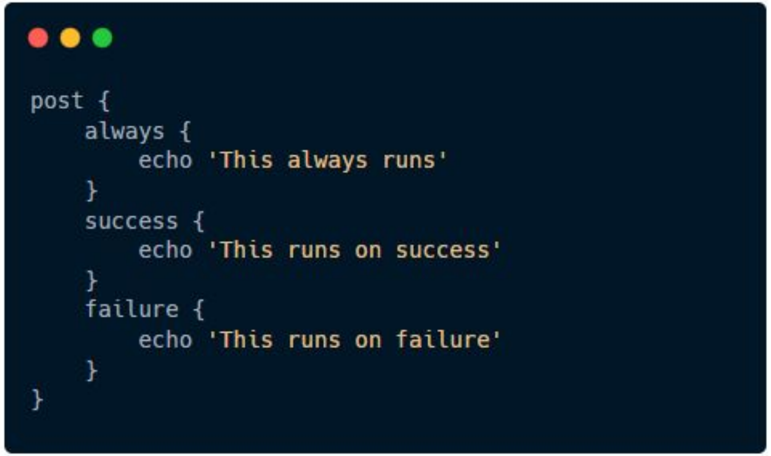


```
environment {
    JAVA_HOME = '/usr/lib/jvm/java-11'
}
```

Key Features of Declarative Pipelines

Post Actions:

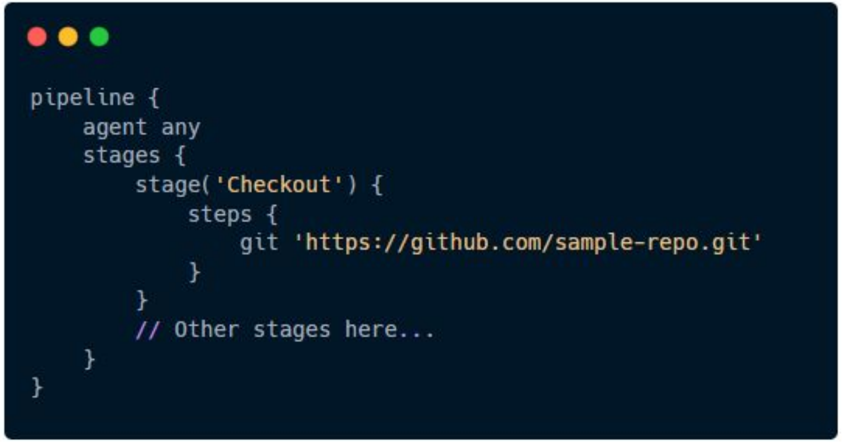
- Define steps to be executed after the pipeline finishes (success or failure).



```
post {
    always {
        echo 'This always runs'
    }
    success {
        echo 'This runs on success'
    }
    failure {
        echo 'This runs on failure'
    }
}
```

Pipeline as Code: Using Jenkinsfiles

- Both **Declarative** and **Scripted Pipelines** can be stored as a **Jenkinsfile** in the source control repository.
- The Jenkinsfile is versioned along with the project code, ensuring that pipeline definitions are consistent with the application.
- Example:



```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/sample-repo.git'
            }
        }
        // Other stages here...
    }
}
```

Best Practices for Jenkins Pipelines

- **Keep Pipelines Simple**: Use **Declarative Pipelines** for standard CI/CD workflows.
- **Version Pipelines as Code**: Store pipelines in source control using a Jenkinsfile.
- **Reuse Code**: Use shared libraries to centralize common pipeline logic.
- **Use Environment Variables**: Use environment variables to pass data between stages.
- **Parallel Execution**: Run tests and builds in parallel to reduce pipeline execution time.
- **Fail Fast**: Set up pipelines to fail quickly on errors, avoiding wasted resources.

Reporting in Jenkins

Why Reporting Matters in Test Automation:

- **Visibility**: Provides insight into the quality of the application.
- **Feedback Loop**: Offers immediate feedback to developers on code changes.
- **Decision Making**: Helps teams make informed decisions about releases.
- **Traceability**: Maintains historical data on test results for accountability.

Types of Reports in Jenkins

Common Types of Reports Generated in Jenkins:

1. **Test Result Reports**:
   - Shows pass/fail status of tests.
2. **Code Coverage Reports**:
   - Indicates how much of the codebase is covered by tests.
3. **Performance Reports**:
   - Measures the performance metrics of the application.
4. **Custom Reports**:
   - User-defined reports based on specific project needs.

Using JUnit for Reporting

Common Types of Reports Generated in Jenkins:

- JUnit is a widely used testing framework for Java applications.
- Jenkins has built-in support for JUnit report generation.

Configuration Steps:

1. Ensure tests are written using JUnit.
2. In the Jenkins job configuration, under **Post-build Actions**, add **Publish JUnit test result report**.
3. Specify the path to the test report XML files (e.g., **/target/surefire-reports/*.xml).

Using TestNG for Reporting

Common Types of Reports Generated in Jenkins:

- TestNG is another popular testing framework that provides extensive reporting options.
- Similar to JUnit, TestNG generates XML reports that Jenkins can publish.

Configuration Steps:

1. Use the TestNG framework for your tests.
2. In the Jenkins job configuration, add **Publish TestNG Results** under **Post-build Actions**.
3. Specify the path to TestNG results (e.g., **/test-output/testng-results.xml).

Custom Test Reports with Allure

Integrating Allure for Advanced Reporting:

- **Allure** is a flexible, lightweight multi-language test report tool.
- Provides a visually appealing report with detailed test information.

Setup Steps:

1. Add Allure dependency in your project (Maven/Gradle).
2. Generate Allure results in your build.
3. Install the **Allure Jenkins Plugin**.
4. Add **Allure Report** under **Post-build Actions** and specify the results directory.

Code Coverage Reporting with JaCoCo

Integrating JaCoCo for Code Coverage Reports:

- **JaCoCo** is a Java code coverage library that provides insights into test coverage.
- **Setup Steps**:
  1. Add JaCoCo plugin to your project (Maven/Gradle).
  2. Run tests with JaCoCo enabled to generate coverage reports.
  3. In Jenkins, add **Publish JaCoCo coverage report** under **Post-build Actions** and specify the report directory.

Generating Performance Reports with JMeter

Using JMeter for Performance Testing Reports:

- **Apache JMeter** is widely used for performance and load testing.
- JMeter generates detailed performance reports that can be integrated into Jenkins.
- **Setup Steps**:
  1. Create JMeter test scripts and save results in a specific format (e.g., XML or CSV).
  2. Add **Publish JMeter test result report** under **Post-build Actions** in Jenkins and specify the results file path.

Viewing and Analyzing Reports in Jenkins

How to Access and Analyze Reports:

- After builds complete, navigate to the job's build history.
- Click on the specific build to view the results.
- Analyze test trends over time and identify areas of improvement.
- Use visual reports to communicate findings with the team.

Best Practices for Reporting in Jenkins

How to Access and Analyze Reports:

- **Consistent Reporting**: Ensure reports are generated for every build to maintain visibility.
- **Automate Report Generation**: Configure Jenkins to automatically generate reports after every build/test run.
- **Use Visual Tools**: Leverage reporting tools that provide clear visualizations for easier understanding.
- **Monitor Trends**: Analyze report data over time to identify trends in test results and code quality.
- **Integrate with Notifications**: Set up notifications to alert teams on test failures or significant changes in test coverage.

Jenkins Best Practices and Tips

Importance of Best Practices:

- **Consistency**: Ensures reliable and repeatable test execution.
- **Efficiency**: Optimizes resource utilization and reduces build times.
- **Quality**: Improves code quality and reduces defects.
- **Team Collaboration**: Facilitates better communication and collaboration among team members.

1. Use Pipeline as Code

Adopt Pipeline as Code:

- Define CI/CD pipelines using **Jenkinsfile** stored in version control.
- Benefits:
  - Versioning of pipeline definitions.
  - Easier collaboration and code reviews.
  - Better traceability of changes.

BEST PRACTICES

2. Implement Parameterized Builds

Utilize Parameterized Builds:

- Allow users to specify parameters at build time for flexible test execution.
- Use cases:
  - Running tests against different environments (DEV, QA, PROD).
  - Testing specific features or components.
- Helps streamline build processes without hardcoding values.

BEST PRACTICES

3. Keep Jobs Simple and Modular

Design Simple and Modular Jobs:

- Break complex jobs into smaller, manageable pieces.
- Use shared libraries or reusable pipeline steps to avoid duplication.
- Easier to maintain, debug, and scale jobs over time.

BEST PRACTICES

4. Optimize Test Execution Time

Reduce Test Execution Time:

- Use parallel execution to run tests simultaneously.
- Implement selective testing to only run relevant tests based on code changes.
- Utilize caching for dependencies and artifacts to speed up builds.

BEST PRACTICES

5. Monitor and Analyze Build Performance

Regularly Monitor Build Performance:

- Use Jenkins' built-in monitoring tools or integrate with external solutions (e.g., Grafana, Prometheus).
- Analyze trends in build times and test results to identify bottlenecks.
- Optimize configurations based on performance metrics.

BEST PRACTICES

6. Manage Test Environments Effectively

Optimize Test Environment Management:

- Use containers (e.g., Docker) to create isolated test environments.
- Implement infrastructure as code (IaC) to manage environments consistently.
- Ensure that environments are reproducible and can be easily provisioned.

BEST PRACTICES

7. Integrate Reporting and Notifications

Implement Reporting and Notifications:

- Generate detailed test reports for every build.
- Set up notifications for build failures, test results, and other critical events (via email, Slack, etc.).
- Ensure stakeholders are informed about the status of builds and tests promptly.

BEST PRACTICES

8. Version Control Everything

Use Version Control for All Artifacts:

- Store Jenkinsfiles, test scripts, configuration files, and reports in version control.
- Enables rollback capabilities and easier collaboration.
- Ensures that the entire CI/CD process is traceable and manageable.

BEST PRACTICES

9. Regularly Update Jenkins and Plugins

Keep Jenkins and Plugins Updated:

- Regularly update Jenkins to the latest stable version to benefit from new features and security patches.
- Ensure plugins are up to date to maintain compatibility and access enhancements.
- Conduct regular audits of installed plugins to remove unused ones.

BEST PRACTICES

10. Foster Team Collaboration

Encourage Collaboration Among Team Members:

- Use tools like Slack, Microsoft Teams, or Jenkins notifications to facilitate communication.
- Share insights from test results and build performance metrics regularly.
- Encourage team members to contribute to pipeline definitions and test scripts.

BEST PRACTICES