**Jenkins:**

Jenkins is an **open-source automation server** used to automate tasks related to building, testing, delivering, and deploying software. It supports Continuous Integration (**CI**) and Continuous Delivery/Deployment (**CD**) pipelines. Jenkins can be extended via plugins, making it highly versatile and adaptable to different workflows.

* **Why do we need Jenkins:**

1. **Automates the CI/CD Process:** Simplifies the software delivery process by automating **builds**, **tests**, and **deployments**.
2. **Early Error Detection:** Continuous Integration enables developers to detect errors early in the development lifecycle.
3. **Improved Collaboration:** Multiple developers can work on the same codebase, with automated testing ensuring compatibility.
4. **Faster Delivery:** Speeds up the software delivery process through automation.
5. **Cross-Platform Compatibility:** Works on multiple operating systems and integrates with numerous tools (**Git**, **Docker**, **Kubernetes**, etc.).
6. **Extensibility:** A rich ecosystem of plugins allows integration with popular DevOps tools.

* **Where do you install/run Jenkins Server:**

The Jenkins server is typically installed on a **separate dedicated server** to manage **CI/CD** pipelines.

**This ensures:**

* **Resource Isolation**: The application and Jenkins do not compete for system resources.
* **Scalability**: Jenkins can handle larger workloads without impacting the application's performance.
* **Security**: Better separation of concerns reduces the risk of one affecting the other.

However, Jenkins can also be installed within the same infrastructure (e.g., on **Docker** or **Kubernetes**) if resources and use case permit.

* **CI/CD Pipelines using Jenkins:**

A Jenkins pipeline automates the steps involved in **Continuous Integration**, **Delivery**, and **Deployment**. Below is a breakdown of each stage:

1. **Continuous Integration (CI):**
   1. **Build:**

* **Purpose:** Convert source code into executable artifacts.
* **Jenkins Setup:**
  + Trigger a Jenkins job when code is pushed to the repository.
  + Clone the repository using a Jenkins Git plugin.
  + Compile the code using build tools like Maven, Gradle, or npm.
  1. **Test:**
* **Purpose:** Run automated tests to validate the code.
* **Jenkins Setup:**
  + Run unit tests, integration tests, and functional tests.
  + Use tools like **JUnit**, **Selenium**, or **Pytest**.
  1. **Merge:**
* **Purpose:** Merge validated code into the main branch.
* **Jenkins Setup:**
  + Use Jenkins pipelines to trigger Git operations, merging only when tests pass.

1. **Continuous Delivery (CD):**

**Automatically Release the Repository.**

* **Purpose:** Prepare code for deployment by packaging and versioning.
* **Jenkins Setup:**
  + Package the code into deployable formats (e.g., **JAR**, **Docker** **images**).
  + Store artifacts in a repository (e.g., Nexus, Artifactory).

1. **Continuous Deployment (CD):**

**Automatically Deploy the Application in Production.**

* **Purpose:** Automate the deployment of the application to production after successful delivery.
* **Jenkins Setup:**
  + Deploy the application using tools like **Docker**, **Kubernetes**, or **Ansible**.
  + Ensure the deployment is triggered only after tests and packaging succeeds.
* **Key Points for Each Stage:**

1. **Continuous Integration:** Focus on frequent integration and testing to catch bugs early.
2. **Continuous Delivery:** Automate artifact preparation to speed up releases.
3. **Continuous Deployment:** Automate production deployment but include manual approvals for critical systems if needed.

* **Why Jenkins for CI/CD:**
* **Automation:** Eliminates manual tasks, reducing errors and time.
* **Scalability:** Can handle large projects with multiple parallel pipelines.
* **Customizability:** Extends functionality via plugins.
* **Community Support:** Backed by a vast community and robust documentation.

**Jenkins Dashboard Modules:**

1. **New Item:**

The **"New Item"** section in Jenkins is where you create jobs or projects to define the tasks you want Jenkins to perform. **Here’s a breakdown:**

1. **Freestyle Project:**

A **Freestyle Project** in Jenkins is a type of job that allows you to create and configure a simple, customized automation process for **continuous integration (CI)** and **continuous delivery (CD)**. It is the most basic type of Jenkins project, providing an easy-to-use interface for automating tasks such as **building**, **testing**, and **deploying** software.

**When to Use:**

1. **Simple Automation:** When you have a straightforward build process without the need for complex orchestration. If you just need a simple series of build steps (e.g., **compiling code, running unit tests, deploying to a server, runs the shell scripts**).
2. **Small or Individual Projects:** Ideal for smaller teams or individual developers who do not need the complexity of Jenkins pipelines.
3. **Beginner User:** Freestyle projects are beginner-friendly. If you are just getting started with Jenkins, this is a good way to learn the basics of Jenkins job configuration without diving into pipeline scripts.
4. **Legacy System:** If your application or build system doesn’t require the flexibility of pipelines (such as complex conditional logic, parallel execution, or environment-specific workflows), a Freestyle project may suffice.
5. **Non-Complex Overflow:** Projects that involve simple tasks like code compilation, testing, and deploying, which don’t require multi-step or conditional logic.
6. **Pipeline:**

A **Pipeline** in Jenkins is a suite of plugins that support the integration and implementation of **continuous delivery (CD)** pipelines within Jenkins. It defines the entire software delivery process, from **building** and **testing** the code to **deploying** it to production, all in a single workflow. Pipelines provide an automated, repeatable process for building, testing, and deploying software applications.

Jenkins pipelines can be described using a **domain-specific language** (**DSL**), which is typically written in **Groovy**. The Jenkins pipeline can be divided into two types:

1. **Declarative Pipeline:**

* This is a simpler and more structured way to define a pipeline.
* It uses a predefined block structure (e.g., **pipeline**, **stages**, **and steps**) and enforces best practices.

1. **Scripted Pipeline:**

* This gives you full flexibility and control over the pipeline but requires more complex **Groovy** scripting.
* It’s more powerful but can be harder to maintain compared to declarative pipelines.

**Key Components of a Jenkins Pipeline:**

* **Stages:** A pipeline is divided into different stages (e.g., **Build**, **Test**, and **Deploy**). Each stage represents a specific part of the process.
* **Steps:** These are individual tasks within a stage, like running a **shell** command, invoking a build tool (e.g., **Maven**), or deploying artifacts.
* **Agents:** Defines where the pipeline or specific stages should run (e.g., a specific node or a Docker container).
* **Post Actions:** You can define actions that run after the pipeline is complete, like sending notifications, archiving artifacts, or cleaning up resources.

**When to Use Jenkins Pipeline:**

Jenkins Pipelines are ideal in the following scenarios:

1. **Complex and Multi-Steps Windows:** If your build process involves multiple stages, tools, or deployment environments, a pipeline offers a structured way to manage this complexity. For example, you can define separate stages for building, testing, deploying, and notifying stakeholders.
2. **Continuous Delivery/Deployment:** Pipelines are particularly useful when automating **continuous delivery** (CD) and **continuous deployment** workflows, where code is frequently **pushed** to production or staging environments after being tested.
3. **Versioned and Consistent Configuration:** If you need to ensure that your build and deployment configurations are versioned, shareable, and consistent across different environments, using a **Jenkinsfile** stored in version control is a strong use case for pipelines.
4. **Multi-Environment and Cross-Platform Builds:** For projects that need to be built, tested, and deployed in multiple environments (e.g., development, staging, production) or on different platforms (e.g., Linux, Windows), Jenkins Pipelines can manage these variations efficiently.
5. **Parallel Execution:** Pipelines allow you to run steps or stages in parallel, which can drastically reduce build times in large, complex projects that need to run multiple tasks simultaneously, such as testing in different environments or compiling different components of an application.
6. **Microservices or Containerized Environments:** For Microservices architectures or when deploying to containerized environments (like **Docker** and **Kubernetes**), Jenkins Pipelines are ideal for managing the workflows that involve **building, testing, and deploying multiple services or containers**.
7. **Conditional and Dynamic Workflows:** If your build pipeline needs to include conditional logic (e.g., different actions based on the branch being built, or only running certain tests if the code has changed), Jenkins Pipelines provide a great way to include this logic.
8. **Integrated with External Tools:** When your build, test, or deploy processes require interaction with external tools (like **Docker**, **Kubernetes**, or **cloud services**), Jenkins Pipelines can easily integrate with them through plugins or custom scripts.
9. **Multi-Configuration Project:**

A project type for testing across multiple configurations (e.g., operating systems, environments).

Ideal for projects requiring cross-platform testing or matrix builds.

1. **Folder:**

Used to organize projects in Jenkins. Helps manage and categorize multiple jobs for easier navigation.

1. **Multi-Branch Pipeline:**

A pipeline project that automatically creates pipelines for each branch in a repository. Useful for managing **CI/CD** workflows for feature branches and main branches in Git.

1. **Organization Folder:**

Automatically creates pipelines for all repositories in an organization. Scans an entire GitHub or Bitbucket organization for repositories and sets up pipelines for them.

1. **System Configuration:**

In Jenkins, **System Configuration** refers to the settings that control how Jenkins operates at a global level. This configuration defines how Jenkins interacts with **tools**, **nodes**, **plugins**, and **cloud** resources to enable automated builds, testing, and deployment processes.

1. **System:**

The **System Configuration** in Jenkins contains global settings that impact the entire Jenkins environment. These settings help to configure how Jenkins interacts with the operating system, networking, security, and other tools.

**Key Components in the System Configuration:**

1. **Jenkins URL:** The base URL that Jenkins uses to access the system. It’s important for proper operation of webhooks, notifications, and other integrations.
2. **System Message:** A custom message that can be displayed on the Jenkins dashboard. Often used to display important announcements or alerts.
3. **Security Settings:**

* **Security Realm**: Determines how users are authenticated, e.g., through Jenkins' internal user database, LDAP, or Active Directory.
* **Authorization Strategy**: Defines who has access to what within Jenkins (e.g., Matrix-based security, Project-based security, etc.).

1. **Global Properties:** Defines environment variables that are available to all Jenkins jobs and pipelines.
2. **Environment Variables:** System-wide variables that are available during the execution of Jenkins jobs and pipelines.
3. **Logging:** Configures system-wide logging levels (like **INFO**, **DEBUG**, **WARN**) for various Jenkins components, which can be used for debugging and tracking the execution of jobs.
4. **Tools:**

The **Tools Configuration** area in Jenkins allows you to define and configure external tools that Jenkins will use during builds. These tools might include **compilers**; **build systems**, **version control systems**, and **testing frameworks**.

**Commonly Configured Tools in Jenkins:**

1. **JDK (Java Development Kit):** Jenkins can be configured to use specific versions of **JDK** to compile and run Java applications. Jenkins supports different versions of **JDK**, and you can define which version of **JDK** should be used globally or for specific jobs.
2. **Maven:** If your project uses **Maven** as a build tool, Jenkins can be configured to point to a specific installation of **Maven**.
3. **Gradle:** For projects that use Gradle as their build automation tool, you can configure Gradle installations globally.
4. **Git:** If you're using Git for source control, you can specify the Git binary that Jenkins will use.
5. **Other Tools:** Depending on your projects, you might need to configure additional tools like **Node.js**, **Docker**, **Python**, or **Ant**.

**How to Configure:**

* Go to **Manage Jenkins > Global Tool Configuration.**
* You can add paths to these tools, or specify automatic installations that Jenkins will handle for you.

1. **Plugins:**

**Plugins** are essential in Jenkins because they **extend its functionality**. Jenkins can be customized to support a wide variety of languages, tools, and services, all thanks to plugins.

**Key Concepts in Jenkins Plugins:**

1. **Installation:**

* Jenkins has a **Plugin Manager** (under **Manage Jenkins > Manage Plugins**) where you can install, update, or remove plugins.
* Plugins can be installed from the **Jenkins Plugin Repository**, or you can upload your own custom plugins.

1. **Types of Plugins:**

* **Build Tools**: Plugins like Maven, Gradle, or Docker to integrate with your build system.
* **Version Control**: Git, SVN, and Mercurial plugins to integrate Jenkins with your source code repositories.
* **Notification Plugins**: Slack, Email, or HipChat plugins to send notifications about build statuses.
* **Deployment Plugins**: AWS, Kubernetes, Docker, and others for deploying your application.
* **Testing Plugins**: JUnit, Cucumber, and others for integrating test results into Jenkins.

1. **Plugin Configuration:**

* Once a plugin is installed, it can be configured under **Manage Jenkins > Configure System.**
* For example, configuring the **Slack Plugin** would require adding your Slack workspace credentials and setting up the notification preferences.

1. **Important Consideration:**

* Plugins can be installed either globally or for specific jobs.
* It’s important to periodically update plugins to ensure compatibility with new versions of Jenkins and maintain security.

1. **Nodes:**

**Nodes** in Jenkins refer to different machines (physical or virtual) where Jenkins executes its jobs. The master node is the main Jenkins server, while **agent nodes** (also known as **slaves**) are additional machines that offload builds from the master node. Nodes allow Jenkins to scale, enabling distributed builds across different environments and systems.

**Key Concepts:**

1. **Master Node:** This is the main Jenkins server that manages job scheduling, user access, and plugin management. It handles the overall operation but is typically not used for executing builds directly.
2. **Agent Nodes (Slaves):**

* These are additional machines that Jenkins can use to run builds, freeing up the master node. They can be physical machines, virtual machines, or even Docker containers.
* You can configure **multiple agent nodes** on different operating systems or hardware to run specific tasks (e.g., testing on multiple platforms).

1. **Node Configuration:**

* Nodes can be configured through **Manage Jenkins > Manage Nodes and Clouds.**
* For each node, you can specify the following:
  + **Remote root directory**: The directory where Jenkins will store files for builds on the agent node.
  + **Labels**: Used to assign specific jobs to particular nodes (e.g., only run a job on a node with a specific label like "**windows**" or "**Linux**").
  + **Launch Method**: How Jenkins will connect to the agent (e.g., SSH, JNLP, or Windows service).

1. **Node Allocation:** Once nodes are configured, Jenkins can distribute workloads across these nodes using the **Label** mechanism. This helps in managing resource allocation for builds.
2. **Clouds:**

**Clouds** in Jenkins are configurations that allow Jenkins to dynamically provision resources for building jobs, typically on cloud-based environments like **AWS**, **Google** **Cloud**, or **Azure**. Using clouds, Jenkins can spin up new virtual machines or containers based on demand, allowing jobs to run on cloud infrastructure instead of local machines or agents.

**Key Concepts:**

1. **Cloud Configuration:** Jenkins supports a wide variety of cloud integrations, where you can configure your Jenkins instance to use cloud services to run jobs dynamically. These integrations can be set up under **Manage Jenkins > Manage Nodes and Clouds > Configure Clouds**.
2. **Cloud Providers:**

* **Amazon EC2**: Jenkins can provision Amazon EC2 instances dynamically to run builds.
* **Google Cloud**: Jenkins can launch virtual machines on Google Cloud Compute Engine.
* **Kubernetes**: Jenkins can use Kubernetes to provision containers on-demand for build jobs.

1. **Dynamic Agents:**

* With cloud configurations, Jenkins can automatically scale up and down based on the number of running jobs, creating temporary agents in the cloud when needed and terminating them after the jobs are finished.
* This reduces the need for dedicated, always-on hardware and helps in scaling Jenkins' resources efficiently.

1. **Advantages of Using Cloud:**

* **Elasticity**: Cloud resources can be scaled automatically based on workload.
* **Cost Efficiency**: You only pay for resources when they are being used, avoiding the costs associated with idle hardware.
* **Isolation**: Cloud environments can provide isolated environments for testing and building, reducing conflicts between jobs.

1. **Security:**
2. **Security:**

Configures authentication and authorization settings.

Ensures only authorized users can access and modify Jenkins.

1. **Credentials:**

Stores secure credentials like **API keys**, **passwords**, and **SSH** keys.

Provides a secure way to manage sensitive information used in pipelines.

1. **Credentials Providers:**

Determines where Jenkins looks for credentials (local or external stores).

Supports centralized or external credential management systems.

1. **Manage & Assign Roles:**

Configures role-based access control (**RBAC**).

Assigns specific permissions to users based on their roles.

1. **Users:**

Manages user accounts for **Jenkins** access.

Helps in identifying and authorizing individuals using Jenkins.

1. **Status Information:**
2. **System Information:**

Displays details about the Jenkins server and environment (**OS**, **Java version**, **plugins**).

Helps in troubleshooting or performance tuning.

1. **System Logs:**

Provides logs of Jenkins server activities.

Essential for debugging issues or monitoring operations.

1. **Load Statistics:**

**Displays data about job execution and node utilization**.

Helps monitor system performance and resource utilization.

1. **User Management:**

Handles user creation, role assignments, and permissions.

Ensures secure, controlled access to Jenkins.

1. **Environment Variables:**

Configurable variables used in Jenkins jobs or pipelines.

Provides flexibility and dynamic configurations.

**Development Lifecycle:**

When building an application, the development lifecycle typically involves several environments to ensure quality, performance, and user satisfaction. **Here is a detailed explanation of each environment with an example:**

1. **Dev/Build Environment:**

* **What it is:**
* The development or build environment is where developers **write**, **compile**, and initially **test** their code.
* This environment is designed for rapid iterations and debugging.
* **Purpose:**
* To provide a space where developers can safely make and test changes without impacting other environments.
* **Example:**
* Developers working on a chatbot app write Python code to add features like natural language processing (NLP).
* They use tools like **PyCharm** or **Visual Studio Code**.
* They execute and debug the code locally and commit changes to a version control system like GitHub.
* **Tools Used:**
* IDEs, local virtual environments, Docker for containerization.
* **Key Characteristics:**
* Highly flexible.
* Bugs are expected.
* Minimal hardware requirements.

1. **Q.A (Quality Assurance) Environment:**

* **What it is:**
* A dedicated environment for testers to validate the **functionality**, **performance**, and **security** of the application.
* **Purpose:**
* To identify and fix bugs, ensuring the application meets functional requirements.
* **Example:**
* QA engineers deploy the chatbot app in a controlled environment.
* They run test cases to verify features like login, user input handling, and database connections.
* Tools like **Selenium** (for UI testing) or **JMeter** (for performance testing) are used.
* **Tools Used:**
* Test automation frameworks and tools.
* **Key Characteristics:**
* Simulates a production-like environment.
* All issues must be resolved before moving to UAT.

1. **UAT (User Acceptance Testing) Environment:**

* **What it is:**
* A pre-production environment where end-users or clients test the application to ensure it meets their requirements.
* **Purpose:**
* To confirm that the application aligns with business needs.
* **Example:**
* The chatbot app is deployed in the **UAT** environment.
* End-users (e.g., teachers in a **K12** chatbot project) interact with the chatbot to ensure it accurately answers questions and generates quizzes as expected.
* Feedback is collected, and necessary adjustments are made.
* **Tools Used:**
* Tools for bug tracking and feedback (e.g., **JIRA**).
* **Key Characteristics:**
* Focuses on user requirements rather than technical details.
* Minimal bugs are expected.

1. **Pilot (Pre-production) Environment:**

* **What it is:**
* A staging environment used for final validation before production.
* It is also referred to as the staging or sandbox environment.
* **Purpose:**
* To simulate the production environment as closely as possible for performance, scalability, and deployment testing.
* **Example:**
* The chatbot app is deployed in a pre-production environment with production-like settings, including load balancers and security configurations.
* Real-world scenarios like concurrent user interactions are simulated.
* Developers ensure the app scales well and handles high traffic without issues.
* **Tools Used:**
* Monitoring and performance tools like **Prometheus**, **Grafana**, and **Splunk**.
* **Key Characteristics:**
* Mirrors production.
* Final checkpoint before deployment.

1. **Production Environment:**

* **What it is:**
* The live environment where the application is available to end-users.
* **Purpose:**
* To serve the application to real users with high reliability, performance, and security.
* **Example:**
* The chatbot app is deployed to production servers (e.g., AWS EC2 or Kubernetes cluster).
* It interacts with databases like MongoDB or Pinecone for real-time user requests.
* Users access the chatbot via a public URL, and the app is monitored for uptime and errors.
* **Tools Used:**
* Production monitoring tools like **AWS** **CloudWatch** or **Datadog**.
* **Key Characteristics:**
* Highly reliable.
* No tolerance for downtime or critical bugs.

**Jenkins Master Node and Worker Nodes:**

1. **Jenkins Master Node:**

* The **Jenkins Master** node is the core component of Jenkins. Where you Install Jenkins and others related configurations.
* **Responsibilities:**
  + Manages the Jenkins UI, configurations, and job scheduling.
  + Delegates build tasks to worker nodes.
  + Stores build results and logs.
* **Example:** A company sets up a **Jenkins** **Master** on a central server to coordinate **CI/CD** tasks. The Master schedules builds and assigns them to Worker nodes based on resource availability.

1. **Jenkins Worker Nodes:**

* **Definition:** These are additional servers (physical or virtual) connected to the Master.
* **Purpose:**
  + Perform the actual **build**, **test**, and **deploy** tasks.
  + Allow distribution of tasks to handle heavy workloads.
* **Example: A team uses 3 Worker Nodes:**
  + **Node 1:** Executes builds for backend services.
  + **Node 2:** Runs automated tests for frontend applications.
  + **Node 3:** Deploys applications to production.

**Jenkins Pipeline Components:**

Jenkins Pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins. Below are the key components of Jenkins Pipelines:

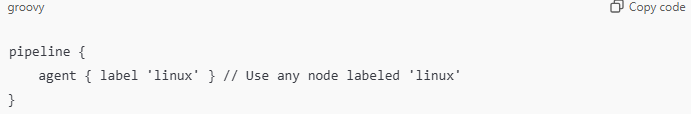
1. **Pipeline:**

* A pipeline defines the entire workflow for building, testing, and deploying an application.
* It can be written as a **Declarative Pipeline** (structured, user-friendly) or a **Scripted Pipeline** (flexible, Groovy-based).
* **Example:**

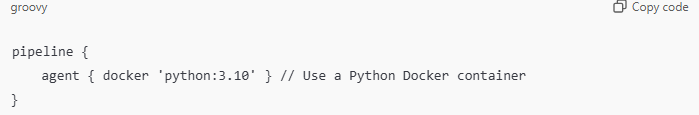


1. **Agent:**

* **Definition**: Specifies where the pipeline or specific stages will run.
* **An Agent can refer to**:
  1. **Worker Nodes**: Physical or virtual machines.
  2. **Docker Containers**: Containers for isolated builds.
* **Example:**



* **Docker Example:**



* **Dynamic Agent Example:**



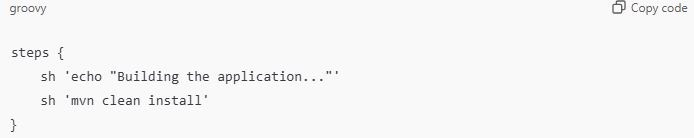
1. **Stages:**

* **Definition:** Logical steps or phases of the pipeline, such as **Build**, **Test**, and **Deploy**.
* **Purpose:** Organizes the pipeline into manageable and readable sections.
* **Example:**



1. **Steps:**

* **Definition:** Smallest unit in a pipeline; describes a single task performed within a stage.
* **Example of Steps:**
  + Running shell commands (**sh 'command'**)
  + Executing scripts (**bat 'command'** for Windows).
  + Cloning repositories.
  + Deploying applications.
* **Example:**



1. **Post:**

* **Definition**: Defines actions to be taken after the pipeline (or a stage) completes.
* **Usage**: Post-actions are categorized into **‘always’**, **‘success’**, **‘failure’**, and **‘unstable’**.
* **Example:**



1. **Environment:**

* **Definition**: Defines environment variables that are used throughout the pipeline.
* **Use Case**: Simplifies configuration management, such as setting **API** keys or paths.
* **Example:**



1. **Parallel:**

* **Definition:** Executes multiple stages or steps simultaneously to speed up the pipeline.
* **Use Cases:** Running different test suites concurrently.
* **Example:**

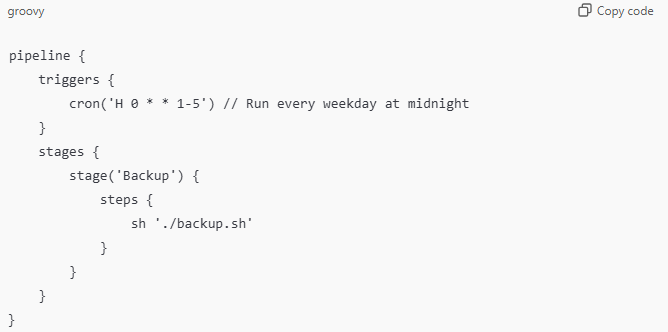


1. **Triggers:**

* **Definition**: Automatically triggers a pipeline based on specific events like code commits or scheduled times.
* **Example**:
* **Poll SCM:** Checks for code changes in the repository at a specific interval.

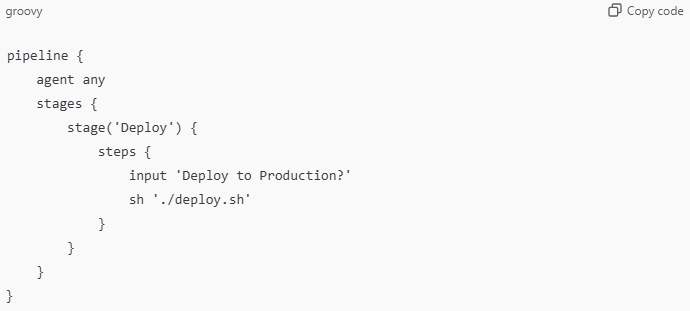


* **Cron Scheduling:** Runs the pipeline at scheduled times.



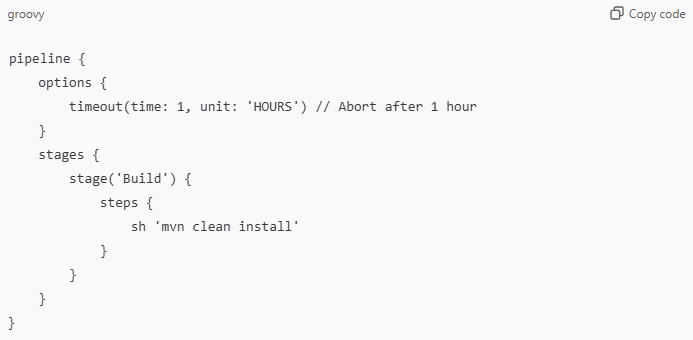
1. **Input:**

* **Definition**: Pauses the pipeline and waits for manual approval before proceeding.
* **Example:**



1. **Options:**

* **Definition**: Customizes pipeline behavior (e.g., **timeout**, **retries**).
* **Common Options**:
  + **Timeout**: Stops pipeline if it takes too long.
  + **Retry**: Retries steps on failure.
* **Example:**



1. **Multi-Stage Example:**

Below is a complete multi-stage example combining all key components:



