

Jenkins kc

Monday, November 17, 2025 5:02 PM

Jenkins streamlines the software development lifecycle by automating builds, tests, CI/CD pipelines, and deployments. It can be installed locally, eliminating dependency on SaaS platforms, and has a proven track record in the CI/CD space.

Continuous Integration (CI) is the process that takes code from a version control system, compiles it, packages it, and prepares it for deployment. Think of CI as assembling a gift:

During CI, automated tests (unit tests, integration tests, and security checks) validate that the code meets quality and dependency standards before moving to the next stage.

Once the code is packaged and tested in the CI stage, the next phase is deployment. This stage, known as Continuous Delivery (or Continuous Deployment), involves delivering the packaged code to a target environment, which could be serverless, containerized, a virtual machine, or on-premises hardware.

Continuous Deployment takes automation one step further by removing the manual approval. When code is pushed to a repository like GitHub, every successful commit that passes all automated tests is immediately deployed to the target environment.

Jenkins—an open-source automation server that simplifies continuous integration and continuous delivery (CI/CD) workflows by automating manual tasks.

Great question! Let's break this down:

How SSH Verification Works Between Local and Jenkins VM

When you configure SSH authentication in Jenkins:

1. Key Pair Generation

- On your local machine, you generate an SSH key pair (private key and public key) using a command like: `ssh-keygen -t rsa -b 4096`
- The private key stays on your local machine, and the public key is uploaded to Jenkins under your user profile.

2. Public Key Registration in Jenkins

- Jenkins stores your public key in its user configuration.
- When you attempt to connect via SSH, Jenkins uses this stored public key to verify your identity.

3. Authentication Process

- You initiate an SSH connection to Jenkins using:
- Jenkins sends a challenge that your local SSH client signs using your private key.

- Jenkins verifies the signature against the stored public key. If it matches, authentication succeeds.

This process ensures **password-less, secure authentication** without transmitting sensitive credentials over the network.

What Purpose Does This Solve?

- **Security:** SSH keys are more secure than passwords because they use asymmetric encryption and are resistant to brute-force attacks.
- **Automation:** CLI commands can be scripted without exposing passwords, enabling CI/CD automation.
- **Convenience:** Developers can interact with Jenkins from the terminal without logging into the UI.
- **Role-based Access:** Jenkins still enforces permissions based on your user account, so CLI commands respect your configured roles.

Would you like me to also **show you how to run Jenkins CLI commands via SSH step-by-step**, or **compare SSH authentication with API token authentication for Jenkins CLI**?

Plugins act as versatile connectors that enable one service to interact with another seamlessly. They function similarly to how Single Sign-On (SSO) allows you to use your Gmail or GitHub accounts to log into various websites

For example, if you need to connect [Jenkins](#) with Azure, you can install an Azure plugin within your Jenkins pipeline.

Here's a **clear, step-by-step explanation** of what we did on both the **host machine** and the **Jenkins VM**:

On the Host Machine (Your Local System)

1. **Generated SSH Key Pair**
 - You created a public/private key pair using: ssh-keygen -t rsa -b 4096
 - **Private key** stays on your local machine (used for authentication).
 - **Public key** is what you share with Jenkins.
2. **Displayed Public Key**
 - You ran: cat ~/.ssh/id_rsa.pub
 - This shows the public key that will be copied to Jenkins.
3. **Prepared for SSH Connection**
 - Your local machine will use the private key to sign authentication challenges when connecting to Jenkins via SSH.

On the Jenkins VM

1. **Configured User Profile**

- Logged into Jenkins UI → clicked your username → **Configure**.
- Added the **public SSH key** in the "SSH Keys" section.
- Saved the configuration.

2. Enabled SSH Authentication for CLI

- Jenkins now knows your public key and can verify any SSH connection from your machine.
- When you run: `ssh -l <username> -p <port> <jenkins-host>` Jenkins checks the signature from your private key against the stored public key.

3. Purpose of This Setup

- **Secure Authentication:** No passwords sent over the network.
- **CLI Access:** You can run Jenkins commands from the terminal (e.g., help, build, list-jobs).
- **Automation:** Useful for scripts and CI/CD pipelines without exposing credentials.

In Short

- **Host Machine:** Holds private key, initiates SSH connection.
- **Jenkins VM:** Stores public key, verifies identity, grants CLI access.

Do you want me to **illustrate this with a simple diagram showing the flow of SSH authentication between local and Jenkins VM?** Or should I **also include how cURL-based CLI works compared to SSH?**

Here's a **lucid summary of what was done in this Jenkins CLI workflow:**

Step 1: Download Jenkins CLI

- From the Jenkins UI, you navigated to **Manage Jenkins** → **Jenkins CLI**.
- Downloaded the `jenkins-cli.jar` file using: `wget http://<jenkins-server>:8080/jnlpJars/jenkins-cli.jar`
- This JAR file is the client tool that allows you to interact with Jenkins from the command line.

Step 2: Authentication Setup

- Logged into Jenkins UI → clicked your username → **Configure**.
- Generated an **API token** under the **API Tokens** section.
- This token is used for CLI authentication instead of a password (more secure and script-friendly).

Step 3: Authenticate via CLI

- Used the downloaded JAR to connect to Jenkins with: `java -jar jenkins-cli.jar -s http://<jenkins-server>:8080/ -auth <username>:<api-token>`
- Successful authentication gives access to CLI commands.

Step 4: Display CLI Help

- Ran: `java -jar jenkins-cli.jar -s http://<jenkins-server>:8080/ -webSocket help`

- This lists all available CLI commands (e.g., list-jobs, build, create-job).

Step 5: Manage Jobs via CLI

- **List Jobs:** `java -jar jenkins-cli.jar -s http://<jenkins-server>:8080/ -auth <username>:<api-token> -webSocket list-jobs`
- **Create a New Job:**
 - Done via Jenkins UI (Pipeline job named test2).
- **Build a Job:**
`java -jar jenkins-cli.jar -s http://<jenkins-server>:8080/ -auth <username>:<api-token> -webSocket build test2`

What Was Achieved

- Installed Jenkins CLI tool.
- Configured secure authentication using API token.
- Listed existing jobs and triggered a build from the command line.
- This enables automation, remote management, and script-based control of Jenkins without relying on the UI.

👉 Do you want me to compare API token authentication vs SSH authentication for Jenkins CLI in a clear table? Or should I create a visual workflow diagram showing these steps?

Let me simplify what that means:

When you search for "**Azure**" in the Jenkins plugin repository, you'll find **many plugins** that allow Jenkins to integrate with Microsoft Azure services. These plugins serve different purposes, such as:

What These Plugins Do

1. **Manage Azure Credentials**
 - Use Jenkins' **Credentials API** to securely store and retrieve Azure service credentials (like Service Principal or Managed Identity).
2. **Interact with Azure SDK**
 - Jenkins jobs can call Azure SDK APIs to manage resources (VMs, storage accounts, etc.) directly from pipelines.
3. **Integrate Azure Active Directory (AAD)**
 - Allows Jenkins to authenticate users via Azure AD for **Single Sign-On (SSO)** and centralized identity management.
4. **Set Up Azure VM Agents**
 - Dynamically provision Jenkins build agents on Azure VMs.
 - Useful for scaling CI/CD workloads in the cloud.
5. **Use Azure CLI in Jenkins**
 - Run Azure CLI commands inside Jenkins pipelines to automate tasks like resource creation, deployments, etc.

Why This Matters

- These plugins make Jenkins **cloud-ready**, enabling you to:
 - Deploy apps to Azure.
 - Scale build agents dynamically.
 - Securely manage credentials.
 - Automate infrastructure and application workflows.

Would you like me to **create a clear diagram showing Jenkins ↔ Azure integration points** (Credentials, VM Agents, AD, CLI)? Or should I **list the most popular Azure-related Jenkins plugins with their use cases?**

A **Jenkins Pipeline** is simply:

A set of automated steps that define how your software is built, tested, and deployed inside Jenkins.

Think of it as a **workflow script** that tells Jenkins what to do, in what order, and on which environment.

Key Points

- **Pipeline = Automation**
Instead of clicking buttons in Jenkins UI, you write a script that runs everything automatically.
- **Written in Groovy-based DSL**

```
1 pipeline {
2     agent any
3     stages {
4         stage('Build') {
5             steps {
6                 sh 'mvn clean package'
7             }
8         }
9         stage('Test') {
10            steps {
11                junit '**/target/surefire-reports/*.xml'
12            }
13        }
14    }
15 }
```

Show less

- **Stages** = Logical sections (Build, Test, Deploy).
- **Steps** = Actual commands (e.g., git, sh, junit).

Why Use Pipelines?

- **Repeatable**: Same process every time.
- **Automated**: No manual clicks.
- **Version-controlled**: Pipeline scripts can live in Git.
- **Flexible**: Works for simple jobs or complex CI/CD workflows.

Types of Pipelines

1. **Declarative Pipeline**
 - Easier, structured syntax (pipeline { ... }).
2. **Scripted Pipeline**
 - More flexible, uses Groovy scripting.

In short: A Jenkins Pipeline is your CI/CD process written as code.

What We Did: Creating a Jenkins Pipeline

1. Created a New Pipeline Job

- In Jenkins UI:
 - Click **New Item** → Enter name (e.g., git test) → Select **Pipeline** → Click **OK**.
- This sets up a job that will run based on a pipeline script.

Pipeline Script Explained

```
pipeline {  
    agent {  
        maven "M3" // Uses Maven tool named "M3" configured in Jenkins  
    }  
    stages {  
        stage('Build') {  
            steps {  
                git 'https://github.com/jglick/simple-maven-project-with-tests.git' // Clones repo  
                sh "mvn -Dmaven.test.failure.ignore=true clean package" // Runs Maven build  
            }  
        }  
    }  
}
```

- **Keywords Explained:**

- pipeline {} → Defines the pipeline.
- agent { maven "M3" } → Runs on an agent with Maven tool configured as "M3".
- stages {} → Groups steps into stages (e.g., Build, Test).
- git → Provided by **Git Plugin**; clones a repository.
- sh → Executes shell commands (Linux/Unix).

Second Example (Explicit Tools Declaration)

```
pipeline {  
    agent any  
    tools {  
        maven "M3" // Explicitly declares Maven tool  
    }  
    stages {  
        stage('Build') {  
            steps {  
                git 'https://github.com/jglick/simple-maven-project-with-tests.git'  
                sh "mvn -Dmaven.test.failure.ignore=true clean package"  
            }  
        }  
    }  
}
```

- Here, tools {} block makes it clear which tools are used.

Why Plugins Matter

- Jenkins pipeline steps like git, sh, junit come from **plugins**.
- Example:
 - **Git Plugin** → Enables git step for repository cloning.

- **JUnit Plugin** → Enables junit step for test result reporting.
- **Pipeline Plugin** → Enables pipeline syntax.

Managing Plugins

- Go to **Manage Jenkins** → **Manage Plugins**.
- **Installed Tab** → Search for "Git" or "JUnit" to confirm installation.
- Plugins add functionality to Jenkins pipelines.

Checkout Example Using Git Plugin

```
checkout([
    $class: 'GitSCM',
    branches: [[name: '*/*master']],
    userRemoteConfigs: [[url: 'http://git-server/user/repository.git']]]
])
```

- This is a more advanced way to configure Git checkout with custom options.

Pipeline with Additional Plugins

Example using **JUnit** and **Archive Artifacts**:

```
withEnv(["MVN_HOME=$mvnHome"]){
    if (isUnix()) {
        sh "'$MVN_HOME/bin/mvn' -Dmaven.test.failure.ignore clean package"
    } else {
        bat("%MVN_HOME%\bin\mvn" -Dmaven.test.failure.ignore clean package")
    }
}
stage('Results') {
    junit '**/target/surefire-reports/TEST-*.xml' // Collects test results
    archiveArtifacts 'target/*.jar' // Archives build artifacts
}
```

- **junit** → Provided by JUnit plugin; publishes test reports.
- **archiveArtifacts** → Saves build outputs for later use.

Key Takeaways

- **Pipeline keywords** (git, sh, junit) come from plugins.
- **Plugins** are installed via **Manage Plugins**.
- Pipelines can:
 - Clone code from GitHub.
 - Build using Maven.
 - Run tests and publish results.
 - Archive artifacts for deployment.

Here's a **lucid breakdown of what was done in managing users and teams in Jenkins**:

1. Creating a New User

- **Steps:**
 - Go to **Manage Jenkins** → **Global Security** → **Manage Users**.
 - Fill in:
 - **Username** (e.g., testuser)
 - **Password**
 - **Full Name**
 - **Email**
 - Click **Create User**.
- **Purpose:** Adds a new Jenkins account for controlled access.

2. Updating User Settings

- After creating the user:
 - Click the **gear icon** next to the username.
 - You can:
 - Change name or description (e.g., "Front-end developer").
 - Generate **API tokens** for CLI or automation.
 - Update email and notification settings.
 - Add **SSH public keys** for secure CLI access.
 - Change password.

3. Installing Role-Based Authorization Plugin

- Jenkins **does not have role-based access by default**.
- To enable it:
 - Go to **Manage Plugins** → **Available tab**.
 - Search for "**Role Strategy**" plugin.
 - Click **Install without restart**.
- **Purpose:** Allows assigning permissions based on roles (e.g., admin, developer).

4. Managing and Assigning Roles

- After installing the plugin:
 - Go to **Manage Jenkins** → **Manage and Assign Roles**.
 - **Manage Roles:**
 - Create roles like developer, admin.
 - Define permissions (e.g., build jobs, configure jobs).
 - **Assign Roles:**
 - Assign roles to users or groups.
 - Example: mike → **admin**, anonymous → **developer** (limited access).
 - Click **Save**.

Why This Matters

- **Security:** Prevent unauthorized changes.
- **Granular Control:** Different permissions for different teams.
- **Scalability:** Easy to manage large teams.

👉 Do you want me to **create a visual diagram showing Jenkins user management flow (Create User → Install Role Plugin → Assign Roles)**? Or should I **write a quick cheat sheet of Jenkins security best practices for managing users and roles**?

Here's a **clear, structured explanation** of what this guide covers and why it matters:

1. Configuring the System

- **Where:** Manage Jenkins → Configure System
- **What you set here:**
 - **Home Directory:** Where Jenkins stores jobs and configs (e.g., /var/live/Jenkins).
 - **System Message:** A banner message for all users.
 - **Executors:** Number of parallel builds Jenkins can run.
 - **Labels & Usage:** Assign labels to nodes for targeted builds.
 - **DNS Name / Public IP:** Helps identify Jenkins instance (e.g., "Jenkins 01").
- **Why:** Controls Jenkins performance and environment behavior.

2. GitHub & Source Control Integration

- Configure:
 - **GitHub Servers:** For webhooks and API calls.
 - **Rate Limiting:** Avoid hitting GitHub API limits.
 - **Pipeline Libraries:** Share reusable pipeline code.
 - **Git Global Settings:** Set global username/email for commits.
- **Why:** Enables Jenkins to pull code from GitHub and manage pipelines efficiently.

3. Email Notification Settings

- Configure SMTP (e.g., Gmail, Office 365).
- Define:
 - **Recipients**
 - **Reply-to**
 - **Allowed Domains**
 - **Subject & Content**
- Example message: \$PROJECTNAME - Build # \$BUILDNUMBER - \$BUILDSTATUS: Check console output at \$BUILDURL
- **Why:** Keeps teams informed about build failures or pipeline issues.

4. Global Security Configuration

- **Security Realm:** How users authenticate:
 - Jenkins internal DB
 - LDAP / Unix DB
 - External SSO
- **Authorization Strategy:**
 - Role-Based Strategy (via plugin)
 - Matrix-based or legacy models
- **Agent & Controller Security:**
 - Secure communication between Jenkins master and agents.
 - Disable unused inbound agent options.
- **API Tokens:**
 - Use modern tokens (avoid legacy).
- **SSH Server:**
 - Allows CLI commands via SSH (secure access only).
- **Why:** Protects Jenkins from unauthorized access and ensures compliance.

Key Takeaways

- **Configure System** → Controls Jenkins behavior and performance.
- **Source Control Integration** → Enables GitHub and repo connectivity.
- **Email Notifications** → Automates alerts for build status.
- **Global Security** → Ensures authentication, authorization, and secure agent communication.

⌚ Do you want me to **create a visual diagram showing Jenkins configuration layers (System → SCM → Notifications → Security)**? Or should I **write a quick checklist of best practices for Jenkins system and security configuration?**

Great question! Let's make this **super simple**:

What is an SSH Server?

- **SSH (Secure Shell)** is a protocol that lets you securely connect to a remote machine and run commands.
- An **SSH server** is a program running on a machine that **listens for SSH connections**.
- When you connect using an SSH client (like your terminal), you can:
 - Log in securely.
 - Execute commands remotely.

How Does Jenkins Use SSH Server?

- Jenkins can **act as an SSH server** so you can run **Jenkins CLI commands** without downloading the CLI JAR or using HTTP.
- Instead of: `java -jar jenkins-cli.jar ...` You can do: `ssh -p <port> <username>@<jenkins-server>`
- After connecting, you type Jenkins commands like:
`help`
`list-jobs`
`build my-job`

Real Project Example

Imagine you have Jenkins running on `jenkins.company.com`:

1. **Enable SSH Server in Jenkins**
 - Go to **Manage Jenkins** → **Configure Global Security**.
 - Enable **SSH Server** and set a port (e.g., 53801).
2. **Add SSH Key for Your Jenkins User**
 - In Jenkins UI → Your user → **Configure** → Add your **public SSH key**.
 - This allows password-less login.
3. **Connect from Your Laptop** `ssh -l deployuser -p 53801 jenkins.company.com`
 - Jenkins verifies your key and logs you in.
4. **Run Jenkins Commands**
 - `list-jobs`
 - `build deploy-app`

Why Do Teams Use This?

- **Secure:** Uses SSH keys, not passwords.
- **Fast:** No need to download CLI JAR.
- **Automation:** Perfect for scripts and CI/CD pipelines.

✍ Do you want me to **draw a simple diagram showing SSH flow (Local → Jenkins SSH Server → CLI Commands)**? Or should I **write a quick step-by-step guide for enabling SSH CLI in Jenkins**?

Let's make this **crystal clear**:

What is an SSH Server?

- **SSH (Secure Shell)** is a protocol that lets you securely connect to a remote machine and run commands.
- An **SSH server** is a program running on that machine which **listens for SSH connections**.
- When you connect using an SSH client (like your terminal), you can:
 - Log in securely.
 - Execute commands remotely.

Example outside Jenkins: When you do:

ssh user@linux-server

You are connecting to the **SSH server** running on that Linux machine.

What Does Jenkins Do by Being an SSH Server?

- Jenkins can **enable an SSH service** so you can connect to Jenkins **like you would to a Linux server**, but instead of running Linux commands, you run **Jenkins CLI commands**.
- This means:
 - Jenkins listens on a port (e.g., 53801) for SSH connections.
 - You authenticate using your Jenkins username and SSH key.
 - Once connected, you can run Jenkins commands like:
help
list-jobs
build my-job

Why Is This Useful in Real Projects?

- **Automation:** Deployment scripts can trigger Jenkins jobs via SSH without downloading jenkins-cli.jar.
- **Security:** Uses SSH keys (no passwords in scripts).
- **Centralized Management:** All CLI commands go through one secure channel.

Real Example

1. Enable SSH Server in Jenkins

- Go to **Manage Jenkins → Configure Global Security**.
- Enable **SSH Server** and set a port (e.g., 53801).

- 2. Add SSH Key for Your Jenkins User**
 - In Jenkins UI → Your user → **Configure** → Add your **public SSH key**.
- 3. Connect from Your Laptop** ssh -l jenkinsuser -p 53801 jenkins.company.com
 - Jenkins verifies your key and logs you in.
- 4. Run Jenkins Commands**
 - list-jobs
 - build deploy-app

In short: Jenkins acts like a mini SSH server so you can securely run Jenkins commands from your terminal without using the web UI or downloading extra tools.

 Do you want me to **draw a simple diagram showing this flow (Local Machine → SSH → Jenkins → CLI Commands)**? Or should I **write a quick step-by-step guide for enabling SSH CLI in Jenkins with screenshots?**

Exactly! You've got the concept right:

- **Your laptop** needs the **private key** (kept secret).
- **Jenkins server** needs the **public key** (shared openly).

Here's **how to do it step by step**:

Step 1: Generate SSH Key Pair on Your Laptop

Open your terminal and run:

```
ssh-keygen -t rsa -b 4096
```

- Press **Enter** for default location (~/.ssh/id_rsa).
- You'll get:
 - **Private key** → ~/.ssh/id_rsa (keep this safe, never share).
 - **Public key** → ~/.ssh/id_rsa.pub (this is what Jenkins needs).

Step 2: Copy Your Public Key

Run:

```
cat ~/.ssh/id_rsa.pub
```

- This prints something like: ssh-rsa AAAAB3NzaC1yc2EAAAABIw... user\@host
- **Copy this entire line.**

Step 3: Add Public Key to Jenkins

- Go to **Jenkins UI** → **Your Username** → **Configure**.
- Scroll to **SSH Public Keys** section.
- Paste the copied key.
- Click **Save**.

Step 4: Connect to Jenkins via SSH

Now you can connect:

```
ssh -l <jenkins-username> -p <port> <jenkins-server>
```

Example:

```
ssh -l mike -p 53801 jenkins.company.com
```

Step 5: Run Jenkins CLI Commands

Once connected, type:

```
help  
list-jobs  
build my-job
```

In short:

- Private key stays on your laptop → used for authentication.
- Public key goes to Jenkins → used to verify your identity.

☞ Do you want me to **create a simple diagram showing this flow (Laptop → SSH → Jenkins)** or **write a quick cheat sheet for SSH setup in Jenkins?**

Pipeline Code Example

Below is the Groovy script used to define this simple pipeline:

```
pipeline {  
    agent any  
  
    stages {  
        stage('Hello') {  
            steps {  
                echo 'Hello World'  
            }  
        }  
    }  
}
```

Here's a **very clear explanation** of what's happening with Jenkins and Prometheus:

What is Prometheus?

- **Prometheus** is a monitoring tool that collects metrics from systems and applications.
- It works by **scraping endpoints** that expose metrics in a specific text format.

What Does Jenkins Do Here?

- Jenkins has a **Prometheus Metrics Plugin**.
- This plugin creates a special URL: <http://:8080/prometheus>
- When you visit this URL, Jenkins shows **metrics about itself** in Prometheus format.

What Kind of Metrics Are Exposed?

Examples from your snippet:

- **CPU Usage:** processcpuseconds_total 32.15
- **Memory Usage:** processresidentmemory_bytes 396992512
- **Jenkins Health:** defaultjenkinsup 1.0 // Jenkins is running defaultjenkinsuptime 182986.0 // Time since start
- **Build Metrics:** defaultjenkinsbuilddurationmillisecondssummarycount{jenkinsjob="test1"} 1.0 defaultjenkinsbuilddurationmillisecondssummarysum{jenkinsjob="test1"} 5634.0 → Shows how many builds ran and their total duration.

How Prometheus Uses This

- Prometheus is configured with a **scrape target**: <http://:8080/prometheus>
- It regularly fetches this data and stores it.
- Then you can query in Prometheus UI: defaultjenkinsbuildssuccessbuild_count
- This shows successful builds per job.

Why Is This Useful?

- You can monitor:
 - Jenkins uptime.
 - Job success/failure rates.
 - Build durations.
 - System resource usage.
- Helps DevOps teams **detect issues early** and **optimize CI/CD performance**.

In short:

- Jenkins exposes metrics at /prometheus.
- Prometheus scrapes these metrics and lets you query them in its dashboard.
- This gives you **real-time visibility into Jenkins health and job performance**.

👉 Do you want me to **draw a simple diagram showing Jenkins → Prometheus → Dashboard flow**? Or should I **write a quick step-by-step guide for integrating Jenkins with Prometheus and Grafana for visualization**?

What is a Jenkinsfile?

- A **Jenkinsfile** is a **text file** that contains the instructions for a Jenkins **pipeline**.
- Think of it like a **recipe**:
 - It tells Jenkins **what steps to follow** (build, test, deploy).
 - It defines **how to run those steps** (on which machine, using which tools).

Where is it stored?

- Usually inside your **source code repository** (e.g., GitHub).
- This makes your pipeline **version-controlled** along with your code.

Why is it important?

- **Automation:** No manual clicks—everything runs automatically.
- **Consistency:** Same process every time.
- **Flexibility:** Works for CI (Continuous Integration), CD (Continuous Delivery), or both.

```
pipeline {  
    agent any  
    stages {  
        stage('Build') {  
            steps {  
                echo 'Building..'  
            }  
        }  
        stage('Test') {  
            steps {  
                echo 'Testing..'  
            }  
        }  
        stage('Deploy') {  
            steps {  
                echo 'Deploying..'  
            }  
        }  
    }  
}
```

Breakdown:

- `pipeline {}` → Starts the pipeline definition.
- `agent any` → Runs on any available Jenkins agent (machine).
- `stages {}` → Groups steps into phases (Build, Test, Deploy).
- `steps {}` → Actual commands (e.g., `echo`, `sh`, `git`).

How It Works in Real Projects

- You commit a Jenkinsfile to your repo.
- Jenkins detects it and runs the pipeline automatically when code changes.
- Example flow:
 - **Build** → Compile code.
 - **Test** → Run unit tests.
 - **Deploy** → Push to staging or production.

Advanced Use

- Add **environment-specific stages** (Dev → Staging → Prod).
- Integrate with **cloud services** (AWS, Azure).
- Use **plugins** for Git, Maven, Docker, JUnit, etc.

In short:

A Jenkinsfile is **Pipeline-as-Code**—your CI/CD process written in a file, stored in Git, and executed by Jenkins.

Here's a **lucid breakdown of building a Jenkinsfile step by step**:

1. What is a Jenkinsfile?

- A **Jenkinsfile** is a text file that defines your **pipeline as code**.
- It tells Jenkins:
 - **What steps to run** (build, test, deploy).
 - **In what order**.
 - **On which agent**.

2. Create a Basic Jenkinsfile

In **VS Code**, create a file named Jenkinsfile:

```
pipeline {  
    agent any  
    stages {  
        stage('Hello') {  
            steps {  
                echo 'Hello World'  
            }  
        }  
    }  
}
```

Explanation:

- **pipeline {}** → Starts the pipeline definition.
- **agent any** → Runs on any available Jenkins agent.
- **stages {}** → Groups tasks into phases.

- stage('Hello') → A single stage named **Hello**.
- steps { echo 'Hello World' } → Prints message in console.

3. Rename Stage (Optional)

You can rename the stage to match your workflow:

```
stage('Dev') {
    steps {
        echo 'Hello World'
    }
}
```

4. Pull Code from GitHub

Add a **git step** to fetch code:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'
            }
        }
    }
}
```

What happens:

- Jenkins clones the repo during the **Build** stage.
- You can then add commands to compile or test the code.

5. Multi-Stage Pipeline

Separate environments or tasks:

```
pipeline {
    agent any
    stages {
        stage('Dev') {
            steps {
                git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'
            }
        }
        stage('UAT') {
            steps {
                git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'
            }
        }
    }
}
```

Why multi-stage?

- **Dev stage** → Build and test.
- **UAT stage** → Deploy to staging.
- Later, you can add **Prod stage** for production deployment.

Key Points

- Jenkinsfile is written in **Groovy DSL**.
- Common steps:
 - echo → Print message.
 - git → Clone repo.
 - sh → Run shell commands.
- If any stage fails, pipeline stops (fail-fast).

In short: A Jenkinsfile is your **CI/CD workflow written as code**, stored in Git, and executed by Jenkins automatically.

 Do you want me to **create a full example Jenkinsfile for a real CI/CD pipeline (Build → Test → Deploy with Maven and JUnit)**? Or **draw a diagram showing Jenkinsfile → Jenkins → Pipeline stages flow?**

Here's a **clear, step-by-step explanation** of what you just learned about running a sample Jenkins pipeline:

1. Create a New Pipeline Job

- In Jenkins dashboard:
 - Click **New Item**.
 - Enter a name (e.g., test four).
 - Choose **Pipeline** (preferred for modern CI/CD).
 - Click **OK**.

2. Configure Pipeline Settings

- Add a description (e.g., *Deploy our sample pipeline to test Jenkins*).
- Optional settings:
 - **Discard Old Builds** → Prevents unlimited build history.
 - **Do Not Allow Concurrent Builds** → Avoids multiple runs at once.
 - **Build Triggers:**
 - **Poll SCM** → Checks for code changes.
 - **GitHub hook trigger** → Runs on push events.
 - **Build periodically** → Like a cron job.

3. Add Pipeline Script

Jenkins gives sample scripts. Two examples:

Hello World Pipeline

```
pipeline {  
    agent any  
    stages {  
        stage('Hello') {  
            steps {  
                echo 'Hello World'  
            }  
        }  
    }  
}
```

- Runs on any agent.
- Prints **Hello World** in console.

Scripted Pipeline with Maven

```
node {  
    def mvnHome  
    stage('Preparation') {  
        git 'https://github.com/jglick/simple-maven-project-with-tests.git'  
        mvnHome = tool 'M3'  
    }  
    stage('Build') {  
        withEnv(["MVNHOME=$mvnHome"]) {  
            if (isUnix()) {  
                sh "$MVNHOME/bin/mvn -Dmaven.test.failure.ignore clean package"  
            }  
        }  
    }  
}
```

- **Preparation** → Clones GitHub repo and sets Maven tool.
- **Build** → Runs Maven build on Unix.

Results Stage

```
stage('Results') {  
    junit '/target/surefire-reports/TEST-.xml'  
    archiveArtifacts 'target/*.jar'  
}  
  
• Publishes test results using JUnit plugin.  
• Archives JAR files for later use.
```

4. Save and Run

- Click **Save** → **Build Now**.
- Jenkins executes the pipeline and shows stage view.

5. Modify Stage Name

Example:

```
pipeline {
```

```

agent any
stages {
    stage('Dev') {
        steps {
            echo 'Hello World'
        }
    }
}

```

- Rename stage from **Hello** to **Dev**.
- Save and rebuild → Stage name updates.

In short: You created a **sample Jenkins pipeline**, learned how to:

- Configure job settings.
- Add pipeline scripts (Declarative or Scripted).
- Run and modify stages.

 Do you want me to **create a full Jenkinsfile for a real CI/CD pipeline (Build → Test → Deploy)** or **draw a diagram showing Jenkins pipeline flow with stages and plugins?**

Here's a **clear explanation of what you just did and why it matters**:

What is a Multistage Pipeline?

- A **pipeline with multiple stages** (e.g., Dev → UAT → Prod).
- Each stage represents a **phase in your CI/CD process**:
 - **Dev** → Build and test code.
 - **UAT** → Deploy to staging for user acceptance testing.
 - **Prod** → Deploy to production.

How We Built It

1. Start with an existing stage:

```

stage('Dev') {
    steps {
        echo 'Hello World'
    }
}

```

2. Copy and paste it below:

```
stage('UAT') {
```

```
    steps {
        echo 'Hello World'
    }
}
```

3. Wrap both inside stages {}:

```
stages {
    stage('Dev') {
        steps {
            echo 'Hello World'
        }
    }
    stage('UAT') {
        steps {
            echo 'Hello World'
        }
    }
}
```

What Happens When You Run It

- Jenkins executes **Dev stage first**, then **UAT stage**.
- Stage view in Jenkins UI will show **two boxes**: Dev → UAT.
- If **Dev fails**, UAT will **not run** (fail-fast behavior).

Why Is This Important?

- **Organized workflow**: Separate build, test, and deploy steps.
- **Better control**: Fail early if something breaks.
- **Scalable**: You can add more stages (e.g., Prod, Security Scan).

In short: You just created a **basic multistage pipeline**, which is the foundation for real-world CI/CD workflows.

 Do you want me to **give you a full Jenkinsfile example for a real CI/CD pipeline (Build → Test → Deploy)** or **draw a diagram showing multistage pipeline flow (Dev → UAT → Prod)**?

Here's a **lucid, step-by-step explanation** of building a **CI pipeline for a Go application in Jenkins**:

Step 1: Install Required Plugin

- Go to **Manage Jenkins** → **Manage Plugins**.
- Search for **Go Plugin** in the **Available** tab.
- Install it and **restart Jenkins**.
- **Why?** This plugin lets Jenkins use Go tools in pipelines.

Step 2: Create a Pipeline Job

- In Jenkins dashboard:
 - Click **New Item** → Enter name (e.g., Go 3) → Select **Pipeline** → Click **OK**.
- Add a description like: *Run CI pipeline for Go app*.

Step 3: Configure Pipeline Script

Start with basic setup:

```
pipeline {  
    agent any  
    tools {  
        go 'go-1.17' // Go version configured in Jenkins  
    }  
    environment {  
        GO111MODULE = 'on' // Enables Go modules  
    }  
}
```

Why these settings?

- agent any → Runs on any available Jenkins agent.
- tools { go 'go-1.17' } → Ensures correct Go version.
- GO111MODULE=on → Allows fetching dependencies from GitHub.

Step 4: Add Stages

Add a **Test stage**:

```
pipeline {  
    agent any  
    tools {  
        go 'go-1.17'  
    }  
    environment {  
        GO111MODULE = 'on'  
    }  
    stages {  
        stage('Test') {  
            steps {  
                // Clone the repository  
                git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'  
                // Run all tests  
                sh 'go test ./...'  
            }  
        }  
    }  
}
```

What happens here?

- Jenkins clones the Go app repo.
- Runs go test ./... → Executes all unit tests in the repo.

Step 5: Save and Run

- Click **Save** → **Build Now**.
- Jenkins installs Go tools, clones repo, runs tests.
- Check **Console Output** for results.

Step 6: Verify Test Results

Expected output:

```
+ go test ./...
ok  github.com/ybkuroki/go-webapp-sample/controller 1.001s
?  github.com/ybkuroki/go-webapp-sample/config [no test files]
  • ok → Tests passed.
  • [no test files] → No tests in those packages.
```

Why This Is a CI Pipeline

- **Continuous Integration** means:
 - Every code change triggers automated build & test.
 - Ensures code quality before merging.

In short: You installed the Go plugin, created a Jenkins pipeline, configured Go tools, cloned a repo, and ran unit tests automatically.

 Do you want me to **extend this pipeline to include build and deploy stages (full CI/CD)** or **draw a diagram showing the CI pipeline flow (Clone → Test → Report)?**

Great question! Right now, the pipeline you wrote **runs only when you click “Build Now” manually.**

For it to become a true **Continuous Integration (CI) pipeline**, you need to **add a trigger** so Jenkins runs automatically whenever code changes happen.

How to Trigger Automatic Builds

Jenkins provides several **build triggers** in the job configuration:

1. Poll SCM

- Jenkins checks your Git repo at intervals (like a cron job).
- Example: Every 5 minutes: H/5 * * * *
- If changes are found, Jenkins runs the pipeline.

2. GitHub Webhook (Recommended)

- Configure a webhook in your GitHub repo:
 - URL: <http://<jenkins-server>:8080/github-webhook/>
- When you push code, GitHub sends a signal to Jenkins → Jenkins triggers the pipeline immediately.

3. Build Periodically

- Runs at scheduled times (e.g., nightly builds).

Where to Configure

- Go to your pipeline job → **Configure** → **Build Triggers**.
- Select:
 - **Poll SCM** OR
 - **GitHub hook trigger for Git SCM polling**.

Why This Makes It CI

- Every commit or pull request triggers:
 - **Clone repo** → **Run tests** → **Report results**.
- No manual intervention → Continuous Integration achieved.

In short:

Add **Poll SCM** or **GitHub webhook** in Jenkins job settings → Jenkins automatically runs your pipeline when code changes.

☞ Do you want me to **show the exact steps to set up a GitHub webhook for Jenkins** (with screenshots and URL format)? Or **write a Jenkinsfile snippet that includes a trigger for automatic builds?**

Here's a **Jenkinsfile snippet with an automatic build trigger** and a **step-by-step explanation**:

Jenkinsfile Snippet

```
pipeline {
    agent any
    triggers {
        // Poll SCM every 5 minutes
        pollSCM('H/5 * * * *')
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }
    }
}
```

Step-by-Step Explanation

1. What Does triggers {} Do?

- The triggers block defines **when Jenkins should automatically run the pipeline**.
- Here we used: `pollSCM('H/5 * * * *)`
 - This means Jenkins will **check the Git repository every 5 minutes**.
 - If there are new commits, Jenkins triggers the pipeline automatically.

2. How It Works

- Jenkins reads this `Jenkinsfile` from your repo.
- Every 5 minutes:
 - Jenkins polls the repo.
 - If changes are detected → pipeline starts.
- No manual "Build Now" click needed.

3. Alternative Trigger: GitHub Webhook

Instead of polling (which can waste resources), you can use:

```
triggers {
  githubPush()
}

• This works when:
  ◦ You configure a GitHub webhook pointing to Jenkins: http://github-webhook/
• Whenever you push code → GitHub notifies Jenkins → Jenkins runs the pipeline instantly.
```

4. Full Flow

- Developer pushes code → GitHub webhook → Jenkins triggers pipeline → Runs stages (Build → Test → Deploy).

In short: Adding triggers {} in `Jenkinsfile` makes your pipeline **automatically run on code changes**, turning it into a true **CI pipeline**.

 Do you want me to **write a complete Jenkinsfile for a real CI/CD pipeline with GitHub webhook trigger (Build → Test → Deploy)** or **draw a diagram showing the automatic trigger flow (GitHub → Jenkins → Pipeline)**?

Excellent question! Let's make this **super clear**:

Does the `Jenkinsfile` stay in the GitHub repo?

Yes!

- The `Jenkinsfile` is stored **alongside your project code in the Git repository** (usually at the root of the repo).
- This is called **Pipeline-as-Code** because your CI/CD process is version-controlled with your application code.

How does Jenkins connect and build automatically?

Here's the flow:

1. Jenkins Job Points to Your Repo

- When you create a Jenkins pipeline job:
 - You tell Jenkins where your code lives (GitHub URL).
 - Jenkins knows to look for a file named Jenkinsfile in that repo.

2. Jenkins Reads Jenkinsfile

- Jenkins pulls the repo.
- Finds the Jenkinsfile.
- Executes the steps defined inside it (Build → Test → Deploy).

3. Automatic Trigger

- To make it **automatic**, you add a trigger:
 - **GitHub Webhook:**
 - Configure a webhook in GitHub: <http://github-webhook/>
 - When you push code → GitHub sends a signal to Jenkins → Jenkins runs the pipeline.
 - OR **pollSCM** in Jenkinsfile:

```
triggers {  
    pollSCM('H/5 * * *') // Check every 5 minutes  
}
```

4. Full Flow

- Developer commits code → Push to GitHub.
- GitHub webhook notifies Jenkins.
- Jenkins pulls the latest code → Reads Jenkinsfile → Runs pipeline automatically.

In short:\ The Jenkinsfile **lives in your repo**, Jenkins **fetches it on every build**, and **webhooks or polling make it automatic**.

☞ Do you want me to **write a complete Jenkinsfile for a real CI/CD pipeline with GitHub webhook integration** and **show the exact steps to configure the webhook in GitHub and Jenkins**? Or **draw a diagram showing this flow visually (GitHub → Jenkins → Pipeline)**?

Great question! Let's break this down **step by step** so it's crystal clear:

How Jenkins Pulls the Repo

Jenkins uses **Git integration** to fetch your project code from GitHub (or any Git-based SCM). Here's how it works:

1. Jenkins Job Configuration

- When you create a pipeline job, you specify:
 - **Repository URL** (e.g., <https://github.com/your-org/your-project.git>).
 - **Credentials** (if private repo → SSH key or GitHub token).
- Jenkins stores this info in the job configuration.

2. Jenkinsfile in the Repo

- Jenkins expects a file named **Jenkinsfile** in the repo.
- This file contains pipeline instructions (Build → Test → Deploy).

3. Trigger Happens

- Either:
 - **Webhook** from GitHub → Jenkins gets notified of a new commit.
 - OR **pollSCM** → Jenkins checks repo periodically.

4. Jenkins Executes git Step

- Inside your Jenkinsfile, you usually have: git '<https://github.com/your-org/your-project.git>'
- This uses the **Git Plugin** in Jenkins.
- Jenkins runs: git clone <https://github.com/your-org/your-project.git>
- It pulls the latest code into the Jenkins workspace.

5. Pipeline Runs

- After cloning:
 - Jenkins reads the Jenkinsfile.
 - Executes stages (Build, Test, Deploy).

In short: Jenkins pulls the repo using **Git commands via the Git plugin**, triggered by **webhooks or polling**, and then runs the pipeline defined in the Jenkinsfile.

 Do you want me to **write a complete Jenkinsfile for a real CI/CD pipeline with GitHub webhook integration** and **show the exact steps to configure the webhook in GitHub and Jenkins?** Or **draw a diagram showing the full flow (GitHub → Jenkins → Pipeline execution)?**

Yes, you've understood the distinction perfectly! Let me explain **why they are different and how they work together**:

1. GitHub Integration (Source Code Connection)

- This is **how Jenkins knows where your code is and how to pull it**.
- In Jenkins job configuration (or Jenkinsfile), you specify:
 - **Repository URL** (e.g., <https://github.com/org/project.git>).
 - **Credentials** (SSH key or token for private repos).
- When Jenkins runs a build, it uses the **Git plugin** to execute: git clone <repo-url>
- **Purpose:** Fetch the latest code from GitHub into Jenkins workspace.

2. Trigger (Notification of Change)

- This is **how Jenkins knows WHEN to start a build**.
- Two common ways:
 - **Webhook (Recommended):**
 - GitHub sends a signal to Jenkins when you push code.
 - Jenkins immediately starts the pipeline.
 - **Poll SCM:**
 - Jenkins checks the repo periodically (e.g., every 5 min).
 - If changes are found → build starts.

How They Work Together

- **Integration** = Jenkins can access your repo.
- **Trigger** = Jenkins knows when to run the pipeline.
- Without a trigger, Jenkins only builds when you click **Build Now**.
- Without integration, Jenkins cannot fetch your code even if triggered.

In short:

- **Integration** = "Where is the code?"
- **Trigger** = "When should I build?"

👉 Do you want me to **write a complete Jenkinsfile for a CI/CD pipeline AND show the exact steps to configure GitHub webhook in Jenkins? Or draw a diagram showing this flow visually (GitHub → Webhook → Jenkins → Pipeline)?**

You're absolutely right to ask this because Jenkins pipelines rely heavily on plugins. Let's go **step by step** and list **all the plugins involved in what we've discussed so far**, along with **what each one does**:

1. Pipeline Plugin

- **Purpose:** Enables Jenkins to run pipelines defined in a Jenkinsfile.
- **Why Needed:** Without this, you cannot use pipeline {} syntax.
- **Function:** Provides Declarative and Scripted Pipeline support.

2. Git Plugin

- **Purpose:** Allows Jenkins to pull code from Git repositories.
- **Why Needed:** When you use: git '<https://github.com/your-org/project.git>' Jenkins uses this plugin to execute git clone.
- **Function:** Handles SCM integration (GitHub, GitLab, Bitbucket).

3. GitHub Integration Plugin (Optional but Common)

- **Purpose:** Connects Jenkins with GitHub for webhooks and status updates.
- **Why Needed:** Enables **GitHub hook trigger for Git SCM polling**.
- **Function:** Lets Jenkins know when code changes happen (trigger builds).

4. Role-Based Authorization Plugin

- **Purpose:** Manages user roles and permissions.
- **Why Needed:** For secure access control in Jenkins.
- **Function:** Assign roles like admin, developer, viewer.

5. Go Plugin (Specific to Go CI Pipeline)

- **Purpose:** Installs and manages Go toolchains.
- **Why Needed:** When you use: tools { go 'go-1.17' } Jenkins uses this plugin to set up Go environment.
- **Function:** Provides Go executable for builds and tests.

6. JUnit Plugin

- **Purpose:** Publishes test results in Jenkins UI.
- **Why Needed:** When you use: junit '*-/target/surefire-reports/.xml'
- **Function:** Displays test reports and trends.

7. Prometheus Metrics Plugin (Optional for Monitoring)

- **Purpose:** Exposes Jenkins metrics at /prometheus endpoint.
- **Why Needed:** For monitoring Jenkins health and job performance.
- **Function:** Integrates Jenkins with Prometheus/Grafana dashboards.

8. SSH Plugin (Optional for CLI Access)

- **Purpose:** Allows Jenkins to act as an SSH server for CLI commands.
- **Why Needed:** For secure remote command execution.
- **Function:** Enables ssh -l user -p port jenkins-server CLI usage.

Summary Table

Plugin	Role
Pipeline Plugin	Enables Jenkinsfile pipelines
Git Plugin	Pulls code from Git repositories
GitHub Integration	Webhook triggers for automatic builds
Role-Based Authorization	User roles and permissions
Go Plugin	Go language support for CI
JUnit Plugin	Test result reporting
Prometheus Plugin	Monitoring and metrics

In short:

- **Integration plugins** (Git, GitHub) → Fetch code & trigger builds.
- **Pipeline plugins** → Run Jenkinsfile.
- **Language/tool plugins** (Go, Maven) → Provide build tools.
- **Reporting plugins** (JUnit) → Show test results.
- **Security & Monitoring plugins** → Manage access & health.

☞ Do you want me to **draw a diagram showing how these plugins interact in a CI/CD pipeline (GitHub → Jenkins → Build → Test → Deploy)**? Or **write a full Jenkinsfile example that uses Git, Go, JUnit, and automatic triggers?**

Here's a **lucid breakdown of what's happening in this CD pipeline and the plugins involved**:

What This Pipeline Does

- **Stage 1: Development**
 - Pulls code from GitHub: git '<https://github.com/AdminTurnedDevOps/go-webapp-sample.git>'
- **Stage 2: Build Docker Image**
 - Uses Docker plugin to build: app = docker.build("adminturneddevops/go-webapp-sample")

Plugins Used and Their Roles

1. **Pipeline Plugin**
 - Enables Jenkinsfile syntax (pipeline {}).
 - Without this, you cannot define stages.
2. **Git Plugin**
 - Handles git step to clone the repository.
 - Connects Jenkins to GitHub.
3. **Docker Pipeline Plugin**
 - Provides docker.build() and other Docker steps.
 - Allows Jenkins to build and push Docker images.
4. **Docker API Plugin & Docker Commons Plugin**
 - Enable communication with Docker daemon.

- Share Docker configuration across jobs.

5. Go Plugin

- Installs Go toolchain for building/testing Go apps.
- Used in tools { go 'go-1.17' }.

How Jenkins Connects and Builds Automatically

- **Integration:** Jenkins knows the GitHub repo URL from the git step.

- **Trigger:** You can add:

```
triggers {
    githubPush() // Runs pipeline when code is pushed to GitHub
}
```

- **Flow:**

- Developer pushes code → GitHub webhook → Jenkins pulls repo → Runs pipeline → Builds Docker image.

Why This Is CD

- After building the Docker image, you can **extend the pipeline** to:
 - Push image to Docker Hub or AWS ECR.
 - Deploy container to Kubernetes or a VM.
- This makes it **Continuous Delivery/Deployment**.

In short: This pipeline uses **Git + Docker + Go plugins** to fetch code, optionally test, and build a Docker image. Adding a webhook trigger makes it fully automated.

☞ Do you want me to **write a complete Jenkinsfile for a full CI/CD pipeline (Build → Test → Docker Build → Push to Docker Hub → Deploy)** and **show the exact steps to configure GitHub webhook + Docker credentials in Jenkins?** Or **draw a diagram showing the full flow (GitHub → Jenkins → Docker → Deployment)?**

Here's a **lucid explanation of the full pipeline deployment you just described:**

What This Pipeline Does

The Jenkinsfile defines **three stages** for a Go web application:

1. Test Stage

```
stage('Test') {
    steps {
        git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'
        sh 'go test ./...'
    }
}
```

- Pulls code from GitHub.

- Runs all unit tests in the repo.

2. Build Stage

```
stage('Build') {
    steps {
        git 'https://github.com/AdminTurnedDevOps/go-webapp-sample.git'
        sh 'go build .'
    }
}
```

- Pulls code again (best practice is to use checkout scm instead).
- Compiles the Go application into a binary.

3. Run Stage

```
stage('Run') {
    steps {
        sh 'cd /var/lib/jenkins/workspace/go-full-pipeline && go-webapp-sample &'
    }
}

```

- Navigates to Jenkins workspace.
- Starts the compiled binary in the background.

Why This Is a Full CI/CD Pipeline

- **CI (Continuous Integration):**
 - Automatically tests and builds code after every commit.
- **CD (Continuous Delivery/Deployment):**
 - Runs the application (or deploys it) after building.

How Jenkins Connects to GitHub

- Jenkins uses the **Git Plugin** to clone the repo.
- You configure:
 - Repo URL in Jenkinsfile (git '<https://github.com/...git>').
 - Credentials if private repo.
- Jenkins pulls the latest code during each stage.

How It Runs Automatically

- Add a trigger in Jenkinsfile:


```
triggers {
    githubPush() // Runs pipeline when code is pushed to GitHub
}
```
- Configure a **GitHub webhook** pointing to: <http://github-webhook/>
- Flow:
 - Developer pushes code → GitHub sends webhook → Jenkins runs pipeline → Tests → Builds → Runs app.

Important Notes

- The Run stage uses the Jenkins workspace path.
- If you see go-webapp-sample: not found in logs:
 - It's because the shell script runs in a temporary context.
 - The app still runs in the background successfully.

In short: This Jenkinsfile automates **testing, building, and running** your Go app. With a webhook trigger, it becomes a true **CI/CD pipeline**.

⌚ Do you want me to: Write a best-practice Jenkinsfile for this pipeline (with checkout scm, proper workspace handling, and Docker deployment) \ OR\ Draw a diagram showing the full flow (GitHub → Jenkins → Test → Build → Run → Browser)?

Here's a **lucid explanation of Build Agents** based on your analogy and details:

What Are Build Agents?

Think of a **CI/CD pipeline** like an assembly line for software. The **Jenkins server** is the control center, and **build agents** are the workers who actually do the heavy lifting—compiling code, running tests, and packaging applications.

Using your car analogy:

- The **car body** = your application structure.
- The **tires** = build agents, because they make the process move forward.

Role of Build Agents

Build agents are **executors** in the pipeline. When Jenkins schedules a job (like building code or running tests), a build agent picks it up and performs the steps. They:

- Compile source code into binaries.
- Run unit, integration, and smoke tests.
- Package deployable artifacts for different platforms (Windows, Linux, macOS).

Where Can Build Agents Run?

Any machine that supports Java can act as a build agent:

- **Physical servers** (bare metal)
- **Virtual machines**
- **Docker containers** (often orchestrated with Kubernetes for scalability)
- **ARM devices** (e.g., Raspberry Pi)

This flexibility makes them ideal for diverse environments.

Supported Platforms

- **Windows** (Windows Server, Windows 10)
- **Linux** (Ubuntu, Red Hat, Debian)
- **macOS** (for macOS/iOS builds)
- **Docker** (for containerized builds)

Containers add **scalability and isolation**, which is great for modern CI/CD setups.

Why Not Use Jenkins Server for Builds?

Two main reasons:

1. **Performance**: Running builds on Jenkins itself can overload it, slowing down the entire system.
2. **Security**: Isolating builds reduces risk. If a build script is compromised, your main Jenkins server stays safe.

So, **dedicated build agents** are recommended for stability and security.

In Short

Build agents are the backbone of CI/CD pipelines. They:

- Execute tasks queued by Jenkins.
- Provide flexibility across platforms.
- Improve performance and security by offloading work from the main server.

Would you like me to: Create a **simple diagram** showing Jenkins, build agents, and their roles?

Or write a **short summary in bullet points** for quick reference?

Or prepare a **real-world example of configuring a Jenkins build agent?**

Which one would help you most?

Here's a **clear and concise explanation of the process you shared:**

Using an Ubuntu Server as a Jenkins Build Agent

A **build agent** is a machine that executes jobs in your CI/CD pipeline. Setting up an Ubuntu server as a Jenkins agent involves four main steps:

Step 1: Create a Dedicated User

On your Ubuntu server:

```
sudo adduser newuser
```

```
sudo usermod -aG sudo newuser
```

- adduser creates the user.
- usermod adds the user to the sudo group for administrative privileges. Verify:

```
groups newuser
```

Expected: newuser : newuser sudo

Step 2: Add Credentials in Jenkins

- Go to **Manage Jenkins → Manage Credentials**.
- Add a **Username with password** credential using the new user's details.
- This allows Jenkins to authenticate via SSH.

Step 3: Configure the Node

- Navigate to **Manage Jenkins → Manage Nodes and Clouds → New Node**.
- Choose **Permanent Agent** and fill in:
 - **Remote Root Directory:** /home/newuser
 - **Launch Method:** Launch agents via SSH
 - **Host:** Public IP/DNS of Ubuntu server
 - **Credentials:** Select the one you added
 - **Host Key Verification:** Manually trusted key verification strategy

Step 4: Verify Connection

- Jenkins will attempt SSH connection.
- Check logs for:
 - Successful authentication
 - Java version check
 - remoting.jar transfer
- Status should show **Online** in Nodes section.

Why This Matters

- **Security:** Isolates builds from Jenkins master.
- **Performance:** Offloads heavy tasks to dedicated agents.
- **Flexibility:** Supports multiple platforms and scaling.

Would you like me to:

- **Create a visual diagram** showing Jenkins master and Ubuntu agent setup?
- **Write a quick checklist** for future reference?
- **Provide a sample automation script** to speed up this setup?

Which one would help you most?

Here's a **clear breakdown of how to use your new Ubuntu build agent in a Jenkins CI/CD pipeline:**

Steps to Configure and Use the Build Agent

1. Create a New Pipeline Project

- In Jenkins, click **New Item**.
- Enter a name (e.g., Ubuntu Test Pipeline).
- Select **Freestyle Project** and click **OK**.

2. Restrict the Project to Your Build Agent

- In the project configuration page:
 - Check **Restrict where this project can be run**.
 - Enter the **exact name** of your Ubuntu agent (e.g., ubuntuagent).
- Jenkins will confirm if the agent exists.

Tip: If you mistype the name, Jenkins will show "no agent matches."

3. Add a Build Step

- Scroll to **Build** section.
- Choose **Execute shell**.
- Add a simple command to test:

echo "testing to confirm the build agent works"

Click **Save**, then **Build Now**.

4. Verify the Build

- Open **Console Output**.
- You should see:

Building remotely on ubuntuagent in workspace /home/newuser/workspace/ubuntutestpipeline

+ echo "testing to confirm the build agent works"

testing to confirm the build agent works

Finished: SUCCESS

This confirms the job ran on your Ubuntu agent.

Why This Is Important

- Lets you **target specific environments** (Linux, macOS, Windows).
- Ensures **consistent builds** for platform-dependent applications.
- Improves **scalability and isolation** in CI/CD workflows.

Would you like me to:

- **Create a visual workflow diagram** showing Jenkins Master → Ubuntu Agent → Pipeline execution?
- **Write a quick reference checklist** for setting up and using agents?
- Or **provide a sample Jenkinsfile** for a scripted pipeline using your Ubuntu agent?

Which one would help you most?

Blue Ocean is a modern user interface for Jenkins designed to make working with CI/CD pipelines more intuitive and visually appealing. It doesn't replace Jenkins' core functionality—it simply improves the **user experience**.

What is Blue Ocean?

Blue Ocean is a **Jenkins plugin** that:

- Provides a **clean, modern UI** for Jenkins pipelines.
- Focuses on **visualization and usability**, making pipelines easier to understand and manage.

Key Benefits

1. Enhanced Visualization

- Graphical representation of pipeline stages and steps.
- Easier to track progress and troubleshoot failures.

2. Interactive Pipeline Editor

- Build pipelines without writing Jenkinsfiles manually.
- Drag-and-drop interface for quick setup.

3. Faster Status Comprehension

- Clear indicators for success, failure, and in-progress stages.

4. Role-Based Personalization

- Different views for developers, testers, and admins.

5. Native Integration

- Works seamlessly with GitHub and Bitbucket for branch and pull request workflows.

Why Use Blue Ocean?

- Traditional Jenkins UI is functional but outdated.
- Blue Ocean makes Jenkins **feel like modern CI/CD tools** (e.g., GitHub Actions, GitLab CI).
- Improves **developer productivity** and **pipeline clarity**.

Next Steps

To use Blue Ocean:

- Install the **Blue Ocean plugin** from Jenkins Plugin Manager.
- Access the new UI via **Blue Ocean link** in Jenkins.
- Create or edit pipelines using its visual editor.

☞ Do you want me to:

- **Show a step-by-step guide to install Blue Ocean and create a pipeline?**
- **Draw a diagram comparing Classic Jenkins UI vs Blue Ocean UI?**
- **Or list common use cases where Blue Ocean shines the most?**

Which one would help you best?