JENKINS

What is Jenkins?

Jenkins is an **open-source automation server** used for **Continuous Integration (CI) and Continuous Deployment (CD)** in DevOps. It helps automate software development workflows by **building, testing, and deploying** applications efficiently.

🔹 **Key Features**

- Automates CI/CD pipelines
- Supports plugins (e.g., Git, Docker, Kubernetes, Ansible)
- Works with multiple programming languages
- Distributed architecture for scalability
- Web-based UI & CLI support

🔹 **How Jenkins Works?**

1. Developers push code to a repository (GitHub, GitLab, Bitbucket).
2. Jenkins pulls the latest code and starts a pipeline.
3. It builds & tests the application using tools like Maven, Gradle, or NPM.
4. If successful, Jenkins deploys the code to a staging/production server.

🔹 **Why Use Jenkins?**

✅ Reduces manual work
✅ Speeds up software releases
✅ Detects bugs early
✅ Integrates with DevOps tools

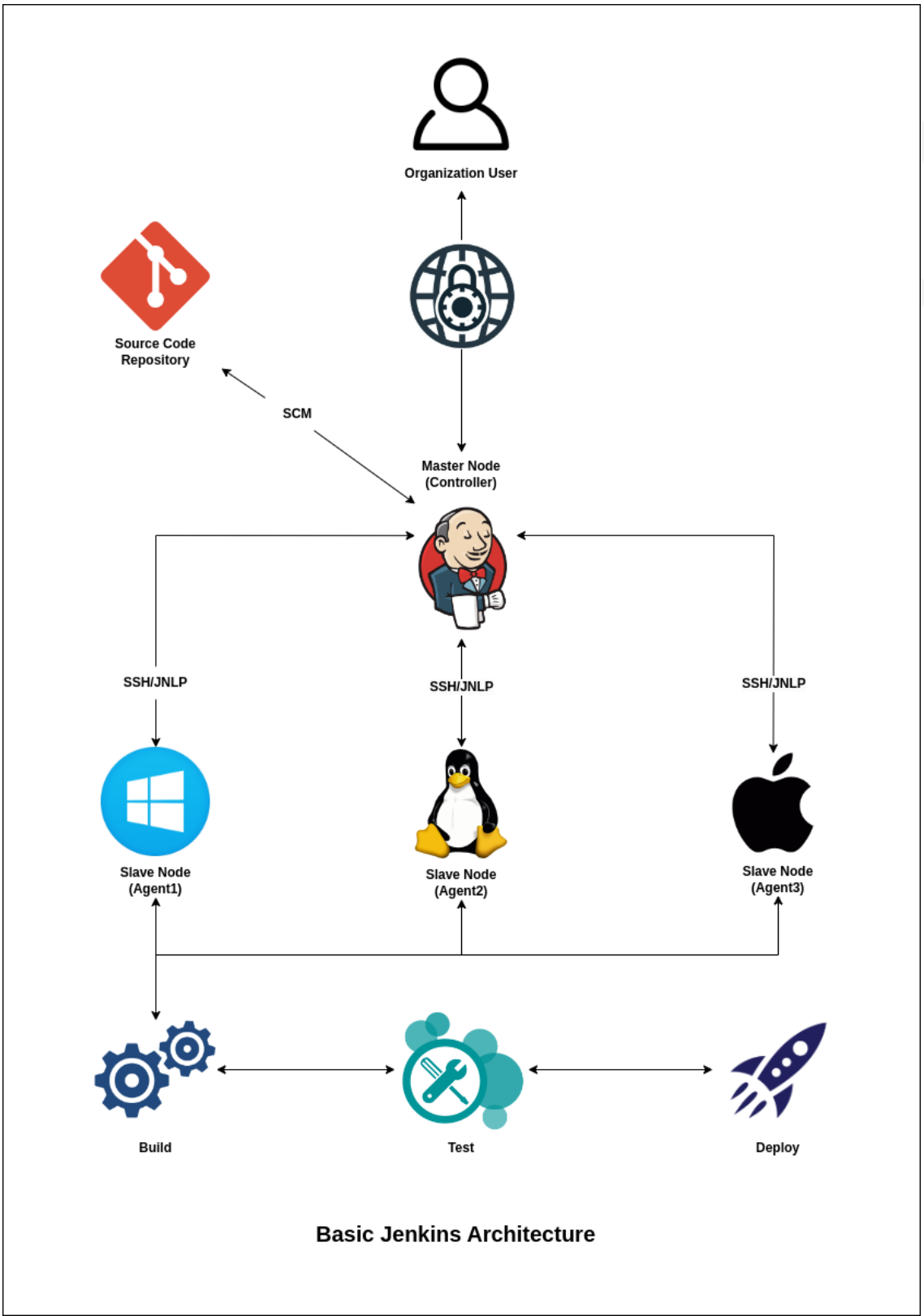
Jenkins Architecture

1. Jenkins Master (Controller)

- Provides **Web UI & CLI** for job configuration.
- The central node that manages **job scheduling and monitoring**.
- Manages configuration
- Loads and manages plugins
- Schedules builds
- Dispatches builds to agents/slaves
- Monitors agents and records build results
- Stores build artifacts and build history

1. Jenkins Agents (Slaves)

- Execute jobs dispatched by master
- Can run on different operating systems
- Support different environments and tools
- Handle actual build execution
- Report status back to master



Basic Jenkins Architecture

JENKINS ARCHITECTURE TYPES

---

1. Standalone Architecture (Single Node)

- **Use Case:** Small teams, simple projects
- **Setup:** Jenkins is installed on a single server that handles both build & execution.
- **Pros:** Easy setup, minimal maintenance
- **Cons:** Performance bottlenecks, no high availability

📌 **Example:**
Jenkins running on a **single Fedora server** executing all jobs.

---

2. Master-Agent Architecture (Distributed)

- **Use Case:** Large teams, heavy workloads
- **Setup:**
    - **Master Node:** Handles UI, job scheduling, and monitoring.
    - **Agent Nodes:** Execute build/test jobs.
- **Pros:** Scales well, reduces master load
- **Cons:** Requires extra setup & maintenance

📌 **Example:**
Jenkins **master** on Fedora, agents running on multiple AWS EC2 instances.

---

3. Multi-Master High Availability (HA)

- **Use Case:** Critical production systems
- **Setup:**
    - Multiple **master nodes** behind a load balancer
    - Shared **database** (MySQL/PostgreSQL)
    - Shared **file system** (NFS/S3)
- **Pros:** High availability, fault tolerance
- **Cons:** Complex setup, requires external DB & storage

📌 **Example:**
Jenkins deployed on **Kubernetes** with multiple replicas of the master.

---

4. Containerized Jenkins (Docker & Kubernetes)

- **Use Case:** Cloud-native & microservices projects
- **Setup:**
  - Jenkins runs inside a **Docker container**
  - Uses **Kubernetes agents** for dynamic scaling
- **Pros:** Fast provisioning, scalability, environment isolation
- **Cons:** Learning curve for container orchestration

📌 **Example:**
Jenkins in a **Docker container** with dynamic **Kubernetes agents** in AWS EKS.

---

5. Serverless Jenkins (Cloud-Native)

- **Use Case:** Fully managed CI/CD
- **Setup:**
  - Jenkins configured to trigger **serverless builds** (AWS Lambda, GitHub Actions)
- **Pros:** No infra management, scales automatically
- **Cons:** Limited control, dependent on cloud services

📌 **Example:**
Jenkins using **AWS CodeBuild** or **GitHub Actions** to execute builds.

Which One to Choose?

Architecture	Best For
Standalone	Small projects, quick setup
Master-Agent	Large teams, distributed workloads
HA (Multi-Master)	Enterprise-level reliability
Containerized (Docker/K8s)	Cloud-native, microservices
Serverless	Fully managed, no infra maintenance

TYPES OF JENKINS PROJECTS

Jenkins supports various types of projects (also called job types) to cater to different automation needs in the CI/CD pipeline. Each project type is designed for specific use cases, such as building code, running tests, or deploying applications.

1. Freestyle Project

The most basic and flexible type of Jenkins project. It allows you to configure build steps, triggers, and post-build actions using a graphical user interface (GUI).

Use Case : Simple projects where you need to execute a sequence of steps (e.g., build, test, deploy)

Pros:

- Easy to set up and configure.
- No scripting knowledge required.

Cons:

- Limited flexibility for complex workflows.

2. **Pipeline Project**

A more advanced project type that uses **Jenkins Pipeline** (declarative or scripted) to define the entire CI/CD workflow as code.

- **Use Case**: Complex workflows involving multiple stages (e.g., build, test, deploy).
- **Features:**
  - Defined using a `Jenkinsfile` (stored in SCM).
  - Supports parallel execution, error handling, and conditional logic.
  - Integrates with Docker, Kubernetes, and other tools.
- **Pros:**
  - Highly customizable and reusable.
  - Better visibility into the pipeline stages.
- **Cons:**
  - Requires knowledge of Groovy scripting (for scripted pipelines).

3. Multi-Configuration Project (Matrix Project)

Description : A project type that allows you to run the same job on multiple configurations (e.g., different operating systems, environments, or versions).

Use Case: Testing applications across multiple environments or platforms.

- Pros:

- Reduces manual effort for multi-environment testing.

- Cons:

- Can be resource-intensive.

- Example: Testing a web application on Windows, Linux, and macOS with different browser versions.

---

4. Multibranch Pipeline Project

- Description: Automatically creates a pipeline for each branch in a repository. It scans the repository for branches and executes the corresponding `Jenkinsfile`.

- Use Case: Projects with multiple branches (e.g., feature branches, release branches).

- Features:

- Automatically detects new branches and pull requests.

- Executes pipelines based on branch-specific `Jenkinsfile`.

- Pros:

- Simplifies CI/CD for Git workflows.

- Reduces manual configuration.

- Cons:

- Requires a `Jenkinsfile` in every branch.

- Example: Automatically building and testing feature branches in a Git repository.

---

5. Folder Project

Description : A container for organizing other Jenkins projects (e.g., Freestyle, Pipeline) into folders. It helps manage large numbers of jobs.

- **Use Case:** Organizing projects by team, application, or environment.

- **Features:**

  - Supports nested folders.

  - Can apply shared configurations (e.g., credentials, environment variables) to all jobs within the folder.

- **Example**: Grouping all microservices-related jobs under a "Microservices" folder.

6. GitHub Organization Project

- **Description:** Automatically scans a GitHub organization or user account for repositories and creates pipelines for each repository with a `Jenkinsfile`.

- **Use Case**: Organizations using GitHub for source code management.

- **Features:**

  - Automatically detects new repositories and branches.

  - Integrates with GitHub webhooks for automatic triggering.

- **Pros:**

  - Simplifies CI/CD for GitHub-based projects.

- **Cons:**

  - Limited to GitHub repositories.

- Example: Automatically building and testing all repositories in a GitHub organization.

7. External Job Project

- **Description:** A project type that monitors and reports on external processes or jobs (e.g., scripts running outside Jenkins).

- **Use Case**: Integrating non-Jenkins jobs into the Jenkins dashboard.

FEATURES :

- Does not execute the job itself but monitors its status.

- Requires the external job to send updates to Jenkins.

- **Pros**:

- Centralized monitoring of external processes.

- **Cons:**

- Limited functionality compared to other project types.

- **Example:** Monitoring a cron job or a script running on a remote server.

8. Maven Project

- **Description :** A specialized project type for building Maven-based Java projects. It automatically configures Maven build steps and goals.

- **Use Case**: Java projects using Maven as the build tool.

- **Features**:

- Automatically resolves dependencies and executes Maven goals.

- Integrates with Maven repositories (e.g., Artifactory, Nexus).

- **Pros:**

- Simplifies Maven project configuration.

- **Cons:**

- Limited to Maven-based projects.

- **Example**: Building and deploying a Java application using Maven.

10. Library Project

- **Description:** A project type used to define shared libraries for Jenkins pipelines. These libraries can be reused across multiple pipelines.

- **Use Case**: Organizations with multiple pipelines sharing common logic or functions.

- **Features:**

- Stores reusable Groovy code in a version-controlled repository.

- Can be imported into Jenkins pipelines.

- **Pros:**

- Promotes code reuse and consistency.

- **Cons:**

- Requires knowledge of Groovy scripting.

- **Example:** Creating a shared library for common deployment steps.

Jenkins Pipeline (Declarative & Scripted Pipelines)

What is a Jenkins Pipeline?

A **Jenkins Pipeline** is a sequence of steps that automate software development processes like **building, testing, and deploying code**. It is written using **Groovy-based syntax** inside a file called **Jenkinsfile**.

Why Use Jenkins Pipelines?

✔️ **Automation** – Reduces manual effort in CI/CD.
✔️ **Code as Pipeline** – Pipelines are stored as code, making them version-controlled.
✔️ **Scalability** – Can handle simple to complex workflows.
✔️ **Plugins & Extensibility** – Supports integrations with tools like Docker, Kubernetes, AWS, etc.

Types of Jenkins Pipelines

Jenkins provides two types of pipelines:
1. **Declarative Pipeline** (Easier & Recommended)
2. **Scripted Pipeline** (More Flexible but Complex)

---

1. Declarative Pipeline

🔹 **What is it?**

- A structured and **simpler way** to write pipelines.
- Uses a `pipeline {}` block with predefined syntax.
- Easier to read and maintain.

🔹 **Basic Structure of a Declarative Pipeline:**

```
pipeline {
    agent any  // Runs on any available agent
    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
            }
        }
        stage('Test') {
            steps {
                echo 'Running tests...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application...'
            }
        }
    }
}
```

🔹 **Key Elements:**

- `pipeline {}` → Defines the pipeline.
- `agent any` → Runs on any available Jenkins agent.
- `stages {}` → Contains multiple **stages** (e.g., Build, Test, Deploy).
- `stage('StageName') {}` → Defines a step in the pipeline.
- `steps {}` → Commands to execute in each stage.

Why use Declarative Pipeline?

- Easy to understand/
- Structured format

- Best for most CI/CD use cases

2. SCRIPTED PIPELINE

- A **flexible but complex** way to write pipelines.
- Uses Groovy scripting without predefined structure.
- Gives more control over flow logic.

🔹 **Basic Structure of a Scripted Pipeline:**

```
node {
    stage('Build') {
        echo 'Building the application...'
    }
    stage('Test') {
        echo 'Running tests...'
    }
    stage('Deploy') {
        echo 'Deploying the application...'
    }
}
```

🔹 **Key Elements:**

- `node {}` → Defines where the pipeline runs.
- `stage('StageName') {}` → Defines each step in the pipeline.
- `echo 'Message'` → Prints a message in the Jenkins console.

Why use Scripted Pipeline?

- More control over flow logic
- Can use Groovy functions & conditions
- Useful for advanced users

Declarative vs. Scripted Pipeline (Comparison Table)

Feature	Declarative Pipeline	Scripted Pipeline
Ease of Use	✅ Easy to learn	❌ Complex
Structure	✅ Predefined syntax	❌ Flexible but unstructured
Best For	✅ Simple workflows	✅ Advanced workflows
Groovy Knowledge	❌ Not required	✅ Required
Flexibility	❌ Limited	✅ High

Example: Full Declarative Pipeline with Git, Maven, and Docker

```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/example/repo.git'
            }
        }
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Docker Build & Push') {
            steps {
                sh '''
                    docker build -t myapp:latest .
                    docker tag myapp:latest myrepo/myapp:latest
                    docker push myrepo/myapp:latest
                '''
            }
        }
    }
}
```

- **Pulls code from GitHub**
- **Builds a Java project using Maven**
- **Runs tests**
- **Builds and pushes a Docker image**

Plugins & Integrations in Jenkins

What are Jenkins Plugins?

- Plugins are **extensions** that add extra features to Jenkins.
- They help **integrate** Jenkins with tools like Git, Docker, Kubernetes, and cloud services (AWS, Azure, etc.).
- Jenkins has **over 1,800 plugins**, making it highly customizable.

Why Are Plugins Important?

- Extend Jenkins functionality **without modifying core code**.
- Automate tasks like testing, deployments, and notifications.
- Improve efficiency by integrating with external tools.

Popular Jenkins Plugins & Their Uses

Plugin Name	Purpose
Pipeline Plugin	Enables **Declarative & Scripted Pipelines** for CI/CD workflows.
Git Plugin	Allows Jenkins to **pull code** from Git repositories (GitHub, GitLab, Bitbucket).
GitHub Integration Plugin	Triggers builds when code is pushed to **GitHub** repositories.
Docker Plugin	Helps Jenkins interact with **Docker containers**, allowing builds inside containers.
Kubernetes Plugin	Deploys Jenkins **agents dynamically** in a Kubernetes cluster.
Slack Plugin	Sends **notifications** to Slack about build results.
Email Extension Plugin	Sends **custom email notifications** after build success or failure.
SonarQube Plugin	Integrates **SonarQube** for code quality analysis.
JIRA Plugin	Links Jenkins builds with **JIRA tickets** for tracking issues.
Artifactory Plugin	Uploads build artifacts to **JFrog Artifactory**.

How to Install a Plugin in Jenkins?

1. Open **Jenkins Dashboard** → Go to **Manage Jenkins**
2. Click **Manage Plugins**
3. Go to the **Available Plugins** tab
4. Search for the plugin name
5. Click **Install** and restart Jenkins

Jenkinsfile

A **Jenkinsfile** is a script that defines a **Jenkins Pipeline** and is written in **Groovy-based syntax**. It allows developers to **automate builds, tests, and deployments** in a structured way.

Types of Jenkins Pipelines

Jenkins provides two ways to write pipelines:

1. **Declarative Pipeline** (Recommended) – Easier to read and write.
2. **Scripted Pipeline** – More flexible but complex.

What is CI/CD?

- **CI (Continuous Integration)** → Developers **push code** frequently; Jenkins **builds and tests** it automatically.
- **CD (Continuous Deployment/Delivery)** → If the code **passes tests**, Jenkins **deploys** it to production/staging.

Step-by-Step CI/CD Workflow in Jenkins

Step 1: Developer Pushes Code to GitHub/GitLab

- A developer **writes code** and pushes it to a repository like **GitHub, GitLab, or Bitbucket**.

Step 2: Jenkins Detects the Change (Webhooks or Polling)

- Jenkins is configured to **monitor the repository**.
- When a change is detected (via **webhook** or **polling**), Jenkins **automatically triggers a build**.

Step 3: Code is Pulled and Built

- Jenkins **clones the latest code** from the repo.

- It **compiles** the code (if required, e.g., Java, C++ projects).

Step 4: Run Automated Tests

- Jenkins runs **unit tests, integration tests, and functional tests** to verify the code.

Step 5: Generate Artifacts (Optional)

- If the build is successful, Jenkins **packages** the code into deployable artifacts (e.g., `.jar`, `.war`, Docker image).

Step 6: Deploy to Staging or Production

- **For Web Apps**: Jenkins deploys to a **Tomcat server, AWS, or Kubernetes**.
- **For Docker Apps**: Jenkins builds and pushes a **Docker image** to a registry (Docker Hub, AWS ECR).

```
docker build -t myapp:latest .
docker push myrepo/myapp:latest
```

- **For Kubernetes**: Jenkins applies the deployment manifest.

```
kubectl apply -f deployment.yaml
```

Step 7: Notify Team

- Jenkins sends a **notification** (Slack, Email) if the build passes or fails.

Summary of the CI/CD Workflow in Jenkins

Step	Action
1	Developer pushes code to GitHub
2	Jenkins detects the change
3	Jenkins pulls and builds the code
4	Runs automated tests
5	Generates artifacts (JAR, Docker image, etc.)
6	Deploys to staging/production
7	Sends notification to the team

User Management & Security in Jenkins

Security is a crucial part of Jenkins to **control access** and **protect the build process**. Jenkins provides **user authentication, authorization, and role-based access control** to manage users effectively.

1. Authentication (Who can log in?)

Authentication ensures that **only authorized users can access Jenkins**.

Methods of Authentication:

Jenkins supports multiple authentication methods:

- **Jenkins' Built-in User Database**
    - Default option where users are manually created.
    - Can set passwords and assign roles.
- **LDAP (Lightweight Directory Access Protocol)**
    - Connects Jenkins with **Active Directory** or other LDAP-based authentication systems.
- **GitHub / Google / Other OAuth Providers**
    - Allows users to log in using **GitHub, Google, or other OAuth** services.
- **Custom Security Realm (SAML, CAS, etc.)**
    - Used in enterprise setups where **external authentication services** handle login.

How to Enable Authentication in Jenkins?

1. Go to **Manage Jenkins > Configure Global Security**
2. Enable **Security Realm** and choose authentication method
3. For built-in database, select **Jenkins' own user database**
4. Create users via **Manage Jenkins > Manage Users**

---

2. Authorization (What can users do?)

Authorization defines **who can perform which actions** in Jenkins.

Types of Authorization in Jenkins:

1. **Anyone can do anything** (⚠️ Not Recommended)
    - No restrictions; all users have full control.
2. **Legacy Mode** (⚠️ Deprecated)
    - Only allows simple access control, not recommended.
3. **Logged-in users can do anything**

    - Only logged-in users have admin rights.
4. **Matrix-based Security** (✅ Recommended)
    - Fine-grained control using **permissions for users & groups**.
    - Example:
        - Admins → Full control
        - Developers → Can build, configure jobs
        - Viewers → Can only see jobs
5. **Role-Based Strategy** (Best for Enterprises ✅)
    - Allows defining **roles with specific permissions**.
    - Example:
        - "Developers" → Access to pipelines
        - "Testers" → Only view build results
        - "Admins" → Full control

How to Enable Authorization?

1. Go to **Manage Jenkins > Configure Global Security**
2. Select **Authorization Strategy** (Matrix-based or Role-based)
3. Assign **permissions** to users or groups

---

3. Securing Jenkins (Best Practices)

1. **Disable Anonymous Access** (Only allow authenticated users)
2. **Use Role-Based Access Control (RBAC)**
3. **Enable CSRF Protection** (`Manage Jenkins > Configure Global Security`)
4. **Restrict who can configure jobs**
5. **Use HTTPS for Secure Connections**
6. **Limit Plugin Installation (Only from trusted sources)**

Jenkins Best Practices (Scaling, Optimization, Troubleshooting)

1. Scaling Jenkins

Scaling ensures Jenkins can handle increased workloads (e.g., more jobs, users, or builds) without performance degradation.

Best Practices for Scaling:

Use Distributed Builds (Agents/Nodes)

- Distribute workloads across multiple agents (also called nodes) to avoid overloading the Jenkins controller.

- Example: Use cloud-based agents (e.g., Kubernetes, AWS EC2) for dynamic scaling.

Master-Agent Architecture

- Keep the Jenkins controller (master) lightweight by offloading builds to agents.

- Avoid running builds directly on the controller.

Use Cloud Plugins

- Plugins like Kubernetes Plugin, AWS EC2 Plugin, or Azure VM Agents can dynamically provision agents as needed.

Monitor Resource Usage

- Use monitoring tools (e.g., Prometheus, Grafana) to track CPU, memory, and disk usage.

- Scale up resources (e.g., RAM, CPU) for the Jenkins controller if needed.

2. Optimizing Jenkins

Optimization ensures Jenkins runs efficiently, reducing build times and resource usage.

Best Practices for Optimization:

Limit Concurrent Builds

- Configure the number of concurrent builds on the controller and agents to avoid resource exhaustion.

- Go to Manage Jenkins> Configure System > # of Executors

Use Lightweight Plugins:

- Only install necessary plugins to reduce overhead.

- Regularly review and remove unused plugins.

Optimize Job Configuration

- Use Pipeline jobs instead of Freestyle jobs for better performance and flexibility.

- Avoid long-running or blocking operations in jobs.

Enable Caching

- Cache dependencies (e.g., Maven, npm) to speed up builds.
- Use tools like Artifactory or Nexus for dependency management.

Clean Up Workspace:

- Regularly clean up old build artifacts and workspaces to free up disk space.
- Use the Discard Old Builds option in job configurations.

Use Parallelism

- Split large jobs into smaller, parallel tasks to reduce build times.
- Example: Use the `parallel` step in Jenkins Pipeline.

3.Troubleshooting Jenkins

Troubleshooting helps identify and resolve issues quickly to minimize downtime.

Common Issues and Solutions:

- **Jenkins is Slow:**
  - Check resource usage (CPU, memory, disk) on the controller and agents.
  - Reduce the number of concurrent builds or scale up resources.
  - Optimize job configurations and pipelines.
- **Builds Fail Randomly:**
  - Check agent connectivity and stability.
  - Look for flaky tests or race conditions in the build process.
  - Review logs in the **Console Output** of failed builds.
- **Plugins Cause Errors:**
  - Disable or uninstall problematic plugins.
  - Update plugins to the latest version.
  - Check the Jenkins Logs (Manage Jenkins > System Log) for plugin-related errors.
- **Out of Memory Errors:**
  - Increase the Java heap size for Jenkins by editing the `JAVA_OPTS` environment variable.
- **Agent Connection Issues:**
  - Ensure the agent is online and properly configured.

- Check network connectivity between the controller and agent.
- Restart the agent if necessary.

Tools for Troubleshooting:

- **Jenkins Logs:**
  - Access logs via **Manage Jenkins** > **System Log**.
  - Check for errors, warnings, or unusual activity.
- **Monitoring Tools:**
  - Use tools like **Prometheus, Grafana, or ELK Stack to monitor Jenkins performance.
- Pipeline Visualization:
  - Use the Blue Ocean plugin to visualize and debug pipeline runs.