**🔹 Jenkins Architecture Deep Dive**

**🧠 Core Components of Jenkins**

Jenkins is an open-source automation server used for building, testing, and deploying software. Its architecture follows a **modular and scalable master-agent** model.

lua

CopyEdit

+-------------------+ +-------------------+ +-------------------+

| Developer Git | ---> | Jenkins Master | ---> | Agent Nodes |

| (GitHub/GitLab) | | (Main controller) | | Worker slaves|

+-------------------+ +-------------------+ +-------------------+

| ^

| |

+--------------------+

| Web UI / CLI |

+--------------------+

**🔧 Jenkins Master Responsibilities**

* Hosts the **Jenkins UI** and core engine.
* Stores:
  + Job configurations
  + Plugin data
  + Pipeline logs and build history
* Schedules jobs, monitors agent status.
* Can run builds, but not recommended in scalable environments.

**🏃 Agent (Slave) Responsibilities**

* Executes build jobs delegated by the master.
* Can be:
  + **Static** (always connected)
  + **Ephemeral** (spun up on demand – e.g., Kubernetes pods, cloud agents)

**🔄 Jenkins Lifecycle**

1. **Start-up:** Jenkins loads configuration files, plugins.
2. **Idle:** Awaits job triggers (via webhook, timer, manual).
3. **Trigger:** Job starts, Jenkins selects an agent.
4. **Execution:** Pipeline steps run on the agent.
5. **Archival:** Logs and artifacts stored on master (or artifact repository).
6. **Notification:** Sends build status (email, Slack, etc.).

**🧩 Master-Agent Model**

**📍 Communication Protocols**

* **SSH**: Traditional way of connecting Linux-based agents.
* **JNLP (Java Network Launch Protocol)**:
  + Agent pulls connection from master (useful for firewall/NAT setups).
  + Java must be installed on the agent.

**🛠️ How Agents Are Connected**

* Static agent setup (long-running node).
* Dynamic/ephemeral agent setup (cloud-based, e.g., AWS EC2, Kubernetes).

**🧪 Declarative Pipeline – Core Structure**

**🆚 Freestyle vs. Pipeline Jobs**

| **Feature** | **Freestyle Jobs** | **Declarative Pipelines** |
| --- | --- | --- |
| UI-based | ✅ Yes | ❌ (code-based) |
| Flexibility | ❌ Limited | ✅ Full control |
| Version control | ❌ Hard to manage | ✅ Can be stored in Git |
| Scalability | ❌ Low | ✅ High |

**📐 Pipeline Syntax Overview**

Declarative pipelines are structured and easy to read.

pipeline {

agent any

environment {

MY\_VAR = 'value'

}

tools {

// Optional: define JDK, Maven, etc.

}

stages {

stage('Build') {

steps {

echo "Building the project..."

}

}

stage('Test') {

steps {

echo "Running tests..."

}

}

stage('Deploy') {

steps {

echo "Deploying application..."

}

}

}

}

**📌 Key Blocks Explained**

| **Block** | **Purpose** |
| --- | --- |
| pipeline | Root block of the script |
| agent | Where to run the pipeline or stage |
| environment | Set environment variables |
| tools | Toolchain like JDK, Maven (not used in this lab) |
| stages | Sequence of high-level tasks |
| steps | Actual commands/scripts inside each stage |

**⚙️ Setting Up Jenkins Agent (Lab)**

**Step-by-step: Add SSH Agent Node**

**1. Install SSH Server on the Agent (Linux)**

bash

CopyEdit

sudo apt update

sudo apt install openssh-server

**2. Add Jenkins Master’s SSH Key to Agent**

On the Jenkins master:

bash

CopyEdit

ssh-keygen -t rsa

ssh-copy-id jenkins@<agent\_ip>

**3. Configure Agent on Jenkins**

* Go to: Manage Jenkins → Manage Nodes and Clouds → New Node
* Enter:
  + Node name: agent-1
  + Type: Permanent Agent
  + Remote root directory: /home/jenkins
  + Launch method: Launch agents via SSH
  + Host: agent\_ip
  + Credentials: Add SSH credentials

**4. Verify Connection**

Click "Save" and then "Launch Agent". Should say **Agent connected**.

**🧪 Lab – Sample Declarative Pipeline**

**Jenkinsfile Example:**

pipeline {

agent { label 'agent-1' }

environment {

PROJECT\_NAME = 'MyDemoApp'

DEMO\_MESSAGE = 'Welcome to Jenkins Pipeline Training!'

BUILD\_DIR = 'build'

}

options {

timestamps()

}

stages {

stage('Initialize') {

steps {

echo 'Starting the pipeline execution...'

echo "Running on agent: ${env.NODE\_NAME}"

}

}

stage('Preparation') {

steps {

echo "Preparing workspace for ${env.PROJECT\_NAME}"

sh '''

echo "Cleaning previous builds..."

rm -rf $BUILD\_DIR

mkdir -p $BUILD\_DIR

echo "Source code simulated." > $BUILD\_DIR/source.txt

'''

}

}

stage('Environment Check') {

steps {

sh '''

echo "Hostname: $(hostname)"

echo "Current User: $(whoami)"

echo "Current Directory: $(pwd)"

echo "Disk Space:"

df -h

'''

}

}

stage('Use Environment Variable') {

steps {

echo "The demo message is: ${env.DEMO\_MESSAGE}"

}

}

stage('Build Simulation') {

steps {

sh '''

echo "Compiling application..."

sleep 2

echo "Build successful!"

'''

}

}

stage('Cleanup') {

steps {

echo 'Cleaning up workspace...'

deleteDir()

}

}

}

post {

always {

echo 'Pipeline execution completed!'

}

success {

echo '✅ The pipeline executed successfully!'

}

failure {

echo '❌ The pipeline failed!'

}

}

}

**🔎 Explanation of Key Elements**

| **Block** | **Explanation** |
| --- | --- |
| agent { label 'agent-1' } | Ensures the pipeline runs on your configured SSH agent. Use your agent label. |
| environment {} | Demonstrates setting an environment variable. |
| echo & sh | Combines scripted output with shell commands. |
| post block | Useful for wrapping up – shows success/failure clearly. |
| deleteDir() | Cleans workspace after execution – useful to prevent clutter. |

**Instructions:**

1. Create a **new pipeline job** in Jenkins.
2. In the configuration:
   * Choose Pipeline script
   * Paste the above code into the text area.
3. Assign label agent-1 to your configured agent.
4. Save and **Build Now**.

**✅ Learning Outcomes Recap**

* ✅ You now understand **Jenkins internal architecture** and lifecycle.
* ✅ You can set up a **master-agent configuration** (via SSH).
* ✅ You can write a basic **declarative pipeline** with stages and steps.

**Debugging Existing Jobs**

When debugging existing jobs in CI/CD pipelines, especially Jenkins pipelines, there are several techniques and best practices to effectively identify and resolve issues. Below are detailed explanations for each aspect:

**1. Techniques to Debug Failing Stages or Steps**

When a stage or step fails, the goal is to locate the root cause quickly. Here are common techniques:

* **Analyze Console Output:**  
  The console output provides detailed logs of each step executed. Look for:
  + The exact command that failed.
  + Error messages, stack traces, or exceptions.
  + Exit codes that indicate failure reasons (e.g., 1 for general errors, 127 for command not found).
* **Stage-Level Isolation:**  
  If a pipeline has multiple stages (e.g., Build, Test, Deploy), you can isolate issues by:
  + Identifying the exact stage where failure occurred.
  + Re-running only that stage if your pipeline supports it (Restart from Stage in Blue Ocean).
* **Post-Build Actions:**  
  Some failures might only be evident in post-build actions like:
  + Artifact uploads
  + Test result parsing
  + Notifications  
    Check logs for issues in these final steps.
* **Retry Logic:**  
  Implement retry logic for flaky tests or external dependencies:

groovy

CopyEdit

retry(3) {

sh "some-command"

}

* **Parallelism Debugging:**  
  If you are running parallel stages, remember that race conditions or environment conflicts can occur. Isolate each parallel path for testing.

**2. Use of echo, sh, and script Blocks for Troubleshooting**

Jenkins provides several options to add debugging information during execution:

* **echo Command:**  
  Use echo to print out information during the build:

groovy

CopyEdit

echo "Starting the build process..."

echo "Current working directory: ${pwd()}"

* **sh Command:**  
  Execute shell commands directly and display output:

groovy

CopyEdit

sh 'ls -la'

sh 'echo "Debugging inside shell command"'

* **script Block:**  
  When more logic is needed (like conditionals), use script:

groovy

CopyEdit

script {

if (isUnix()) {

echo "Unix system detected"

} else {

echo "Windows system detected"

}

}

* **Print Environment Variables:**  
  During debugging, it's often helpful to see environment variables:

groovy

CopyEdit

sh 'env'

**3. Console Output, Blue Ocean View, and Logs Analysis**

* **Console Output:**
  + Available directly in Jenkins for every build.
  + Provides real-time output of running stages and steps.
  + Use the **Search** feature to quickly find errors.
* **Blue Ocean View:**
  + A more visual representation of the pipeline with clickable stages.
  + You can easily spot which stage failed and expand it to see logs.
* **Logs Analysis:**
  + Check not just the Jenkins logs, but also:
    - Application logs (from your build).
    - System logs (if there are system-level issues).
    - Docker logs (docker logs <container-id> if Docker is used).

**4. Re-running Failed Stages and Checking Environment Variables at Runtime**

* **Re-run Failed Stages:**
  + Jenkins allows re-running only the failed stages in Blue Ocean, which saves time.
  + Use the Restart from Stage option to avoid re-running the entire pipeline.
* **Check Environment Variables at Runtime:**  
  You can dynamically inspect environment variables during the build:

groovy

CopyEdit

sh '''

echo "JAVA\_HOME: $JAVA\_HOME"

echo "BUILD\_NUMBER: $BUILD\_NUMBER"

'''

* **Declarative Pipeline Options:**  
  If you are using declarative syntax, you can define environment variables:

groovy

CopyEdit

pipeline {

environment {

JAVA\_HOME = "/usr/lib/jvm/java-11"

}

stages {

stage('Check Java Version') {

steps {

sh 'java -version'

}

}

}

}

**Summary of Best Practices:**

1. Use echo and sh to display debugging information.
2. Isolate issues to specific stages.
3. Leverage Blue Ocean for visual clarity.
4. Re-run only failed stages to save time.
5. Always inspect logs for deeper insights.
6. Print and verify environment variables during execution.