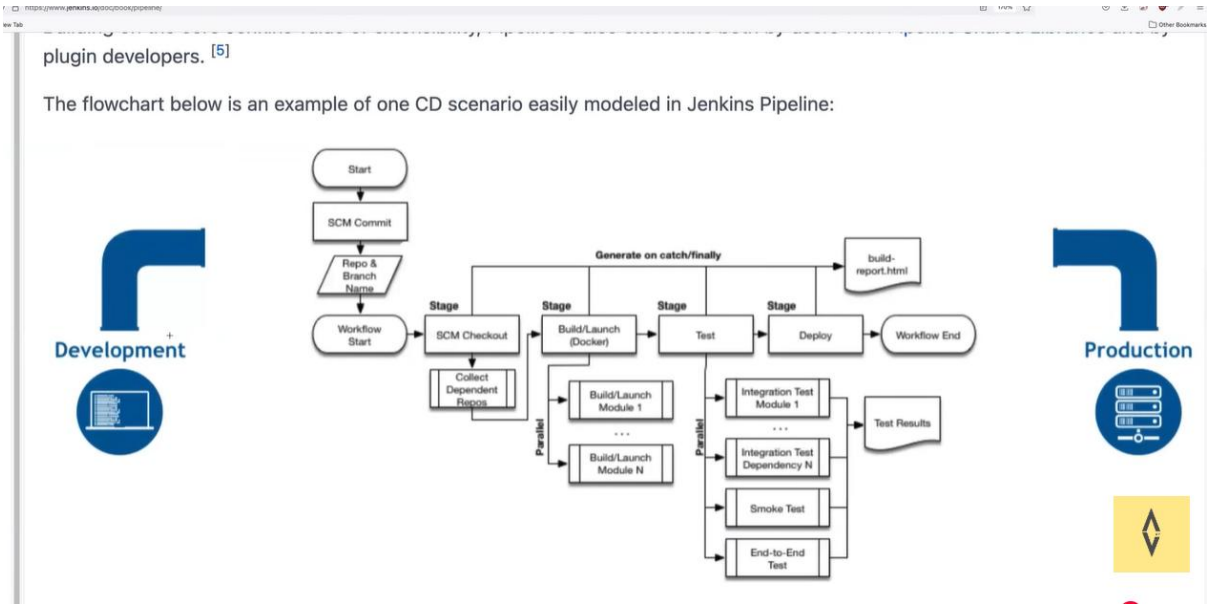
Devops

JENKINS - CI/CD TOOL

JENKINS PIPELINE FLOWCHART EXPLANATION



The flowchart shows a typical CD scenario in Jenkins Pipeline:

Flow:

Start ↓ SCM Commit (Code pushed to version control) ↓ Repo & Branch Name (Identify which repository and branch) ↓ Workflow Start ↓ SCM Checkout (Get code from repository) ↓ Collect Dependent Repos (Get dependencies if needed) ↓

Parallel execution of stages:

Stage 1: Build/Launch (Docker)

- Build/Launch Module 1
- Build/Launch Module 2
- Build/Launch Module N

Stage 2: Test

- Integration Test Module 1
- Integration Test Dependency N
- Smoke Test

- End-to-End Test

Stage 3: Deploy (to staging/production)

Stage 4: Workflow End

Generate on catch/finally: build-report.html

Development environment on left, Production environment on right.

This shows the complete pipeline from code commit to production deployment with multiple parallel stages.

PRACTICAL JENKINS INSTALLATION

STEP 1: CREATE EC2 INSTANCE

On AWS console:

1. Go to EC2 Dashboard
2. Click Launch Instance
3. Name: Jenkins
4. OS: Ubuntu
5. Instance type: t2.medium (Jenkins needs more resources)
6. Create key pair or use existing
7. Launch instance

STEP 2: CONNECT TO INSTANCE

Using MobaXterm:

1. Open MobaXterm
2. Create new SSH session
3. Paste public IP of Jenkins instance
4. Use private key (.pem file)
5. Username: ubuntu
6. Connect

Connection established to the instance.

STEP 3: INSTALL JAVA

Why Java needed: Jenkins is a Java-based application so we need to install Java first.

Update packages:

sudo apt update

Install Java:

sudo apt install openjdk-17-jre -y

Explanation:

- sudo: Run with administrator privileges
- apt install: Package installation command
- openjdk-17-jre: OpenJDK Java Runtime Environment version 17
- -y: Automatically answer yes to prompts

Verify Java installation:

java --version

Expected output:

openjdk version "17.0.8" 2023-07-18

OpenJDK Runtime Environment (build 17.0.8+7-Ubuntu-122.04)

Java is now installed successfully.

STEP 4: INSTALL JENKINS

Run the following commands:

Command 1:

curl -fsSL https://pkg.jenkins.io/debian/jenkins.io-2023.key | sudo tee \

/usr/share/keyrings/jenkins-keyring.asc > /dev/null

Explanation:

- curl: Download content from URL
- -fsSL: Flags for silent, show errors, follow redirects, secure

- https://pkg.jenkins.io/debian/jenkins.io-2023.key: Jenkins GPG key URL
- | (pipe): Send output to next command
- sudo tee: Write to file with sudo permissions
- /usr/share/keyrings/jenkins-keyring.asc: Where to save the key
- /dev/null: Suppress output to terminal

Purpose: Download and save Jenkins repository GPG key for package verification.

Command 2:

```
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
 https://pkg.jenkins.io/debian binary/ | sudo tee \
 /etc/apt/sources.list.d/jenkins.list > /dev/null
```

Explanation:

- echo: Print text
- deb: Debian package repository
- [signed-by=...]: Use this key to verify packages
- https://pkg.jenkins.io/debian binary/: Jenkins package repository URL
- | sudo tee: Write to file with sudo
- /etc/apt/sources.list.d/jenkins.list: Add Jenkins to package sources
- /dev/null: Suppress output

Purpose: Add Jenkins repository to system's package sources list.

Command 3:

```
sudo apt-get update
```

Explanation: Update package lists including newly added Jenkins repository.

Purpose: Refresh package information so Jenkins package is available for installation.

Command 4:

```
sudo apt-get install jenkins -y
```

Explanation:

- sudo apt-get install: Install package
- jenkins: Package name
- -y: Automatically answer yes

Purpose: Install Jenkins from the repository we just added.

Installation output:

Reading package lists... Done

Building dependency tree... Done

...

Setting up jenkins (2.426.1) ...

Created symlink /etc/systemd/system/multi-user.target.wants/jenkins.service

Jenkins is now installed and running.

Jenkins will be installed on your device.

STEP 5: CONFIGURE SECURITY GROUP FOR JENKINS

Understanding AWS EC2 security:

By default, AWS EC2 instances will not accept traffic from external world. Inbound traffic rules are blocked by default.

Jenkins details: Jenkins is an application which has its port running on 8080.

Problem: AWS is not yet accepting traffic on port 8080 because of its default behavior.

Solution: Open port 8080 in security group.

Opening port 8080:

Step 1: Go to EC2 instance page

Step 2: Select your Jenkins instance

Step 3: Click on Security tab (bottom panel)

Step 4: Click on the existing security group link

Step 5: Click on "Inbound rules" tab

Step 6: Click "Edit inbound rules" button

Step 7: Click "Add rule"

Step 8: Configure rule:

- Type: Custom TCP
- Port range: 8080
- Source type: Anywhere-IPv4 (0.0.0.0/0)
- Description: Jenkins web interface

Step 9: Click "Save rules"

Inbound rules now allow traffic on port 8080.

STEP 6: ACCESS JENKINS WEB INTERFACE

Copy public IPv4 address: Go to EC2 instance details and copy public IPv4 address.

Example: 54.123.45.67

Open browser: Type in address bar:

http://54.123.45.67:8080

Press Enter.

STEP 7: UNLOCK JENKINS

Jenkins page will appear with message:

"Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

/var/lib/jenkins/secrets/initialAdminPassword

Please copy the password from either location and paste it below.

Administrator password: [input field]"

Get the password:

Go to terminal and execute command:

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Expected output:

a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6

Copy this password.

Paste in Jenkins page: Paste the password in the "Administrator password" field.

Click "Continue".

STEP 8: INSTALL PLUGINS

Page appears with two options:

Option 1: Install suggested plugins Option 2: Select plugins to install

Our choice: Install suggested plugins

What happens: Plugins will get automatically installed.

Plugins being installed:

- Git plugin
- Pipeline plugin
- GitHub plugin
- Credentials plugin
- And many more essential plugins

This takes 2-5 minutes.

STEP 9: CREATE FIRST ADMIN USER

Page appears to create first admin user.

Fill in details:

- Username: admin (or your choice)
- Password: (strong password)
- Confirm password: (same password)
- Full name: Your Name
- E-mail address: your.email@example.com

Click "Save and Continue".

STEP 10: INSTANCE CONFIGURATION

Page shows Jenkins URL.

Default URL:

http://54.123.45.67:8080/

Click "Save and Finish".

STEP 11: JENKINS IS READY

Page shows: "Jenkins is ready!"

Click "Start using Jenkins".

You are now on Jenkins Dashboard.

Jenkins is successfully installed and configured.

JENKINS ARCHITECTURE

HOW JENKINS WORKS IN REAL TIME

JENKINS MASTER

What is Jenkins Master: The EC2 instance where Jenkins is installed can be called Jenkins Master.

Role of Jenkins Master:

- Schedules jobs
- Manages configuration
- Coordinates workers
- Stores pipeline definitions
- Handles UI and API

THE LOAD PROBLEM

In our organization:

- Multiple developers
- Multiple development teams

Result: Jenkins gets a lot of load.

Solution to offload: You should always use Jenkins Master only for scheduling purpose.

Why not run everything on master: All the work cannot be run on a single Jenkins Master.

PROBLEM WITH RUNNING EVERYTHING ON MASTER

Problem: Package conflicts

Example conflicts:

- Team 1 requires Java version 17
- Team 2 requires Java version 22
- Team 3 requires Node.js version 14
- Team 4 requires Node.js version 18
- Cannot have multiple versions on same machine

Result: Conflicts occur, jobs fail.

TRADITIONAL SOLUTION: WORKER NODES

What people do:

1. Create Jenkins Master
2. Create Jenkins Worker Nodes (separate EC2 instances)

Worker nodes will be used by the applications.

Architecture:

Jenkins Master (Scheduler)

|

+--- Worker Node 1 (Ubuntu, Java 17)

+--- Worker Node 2 (Ubuntu, Node.js 18)

+--- Worker Node 3 (Windows, .NET)

How it works:

- Job requires Java 17: Runs on Worker Node 1
- Job requires Node.js: Runs on Worker Node 2
- Job requires Windows: Runs on Worker Node 3

This architecture was good 2-3 years ago.

PROBLEMS WITH WORKER NODE APPROACH

Problem 1: Idle resources

Scenario: You have 3 worker nodes:

- Worker Node 1: Ubuntu
- Worker Node 2: Ubuntu
- Worker Node 3: Windows

Usage pattern:

- Worker Node 1 and 2: Used frequently
- Worker Node 3: Very rarely used (no Windows applications)

Result: Jenkins Worker Node 3 is always sitting idle.

Issue: You are wasting resources on your AWS instance.

Suggested solution: Stop the instance when not in use and turn it on when required.

Counter-problem: We never know when we may require this instance.

Additional issues:

- Takes time to start instance
- Jenkins needs to reconnect
- First job delayed
- Manual intervention needed

Problem 2: Version conflicts

With advancement of microservices: You have different types of applications.

Example: Worker Node 1 is used by two teams:

- Team A needs Node.js version 15
- Team B needs Node.js version 18

Conflict: Cannot have both versions on same machine.

Manual solution: Log in to instance and upgrade/downgrade Node.js version.

Problems:

- Time consuming
- Manual work
- Risk of breaking other jobs
- Not scalable

Problem 3: Resource waste

Scenario: Application is not being used or no changes are being made.

Result: EC2 instance sitting idle but still:

- Running 24/7
- Consuming resources
- Costing money
- Not doing any work

THE MODERN SOLUTION: DOCKER AS AGENTS

To solve these problems: We are going to use latest approach: using Jenkins with Docker as agents.

What this means: We will try to run the pipeline on Docker containers instead of EC2 worker nodes.

WHY DOCKER CONTAINERS ARE BETTER

Advantage 1: Lightweight

Docker containers are very light in weight compared to virtual machines.

Comparison:

- EC2 instance (VM): Takes minutes to start, GBs of size
- Docker container: Takes seconds to start, MBs of size

Advantage 2: No version conflicts

First scenario (Worker Nodes):

- Want to deploy app with Node.js 15
- Worker node has older version installed
- Manually log in to instance
- Upgrade Node.js version
- Risk breaking other applications

With Docker:

- Specify Node.js 15 in Dockerfile
- Container created with Node.js 15
- Other containers can use different versions
- No conflicts

Conflict scenario with worker nodes:

- Team 1 using Worker Node 1 needs Node.js 15
- Team 2 using same Worker Node 1 needs Node.js lower version
- Conflict occurs

With Docker:

- Team 1 job: Creates container with Node.js 15
- Team 2 job: Creates container with Node.js 12
- Both run simultaneously
- No conflicts

Advantage 3: Easy updates

With containers:

- Anytime created
- Anytime destroyed
- Anytime updated

Process: You just have to modify the Dockerfile and then you can use any container that you want.

Advantage 4: Cost efficiency

How it saves cost:

- Container created only when job runs
- Container deleted after job completes
- No idle resources
- Pay only for actual usage

Example:

- Job runs for 5 minutes
- Container exists for 5 minutes
- Rest of the day: 0 containers
- Cost: Only 5 minutes of compute

IMPLEMENTING DOCKER AS JENKINS AGENT

STEP 1: INSTALL DOCKER ON JENKINS MASTER

Install Docker on the same machine where Jenkins is installed.

Command:

sudo apt install docker.io -y

Explanation:

- sudo apt install: Install package
- docker.io: Docker package
- -y: Auto-confirm

Verify installation:

docker --version

Expected output:

Docker version 24.0.5, build 24.0.5-0ubuntu1~22.04.1

Docker is now installed.

STEP 2: UNDERSTANDING DOCKER DAEMON

What is Docker daemon: Docker typically runs on a single process called daemon process.

Security feature: This daemon process is by default not accessible to other users.

Current situation: Since I have installed Docker using root, only root user will have access to this Docker daemon and can grant access to this Docker daemon.

STEP 3: GRANT ACCESS TO JENKINS USER

Jenkins user: Jenkins installation creates a user called jenkins user.

Requirement: I will grant access to jenkins user and also to ubuntu user.

Commands to execute:

sudo su -

usermod -aG docker jenkins

usermod -aG docker ubuntu

systemctl restart docker

Command explanation:

Command 1: sudo su -

- Switch to root user
- Needed for user management

Command 2: usermod -aG docker jenkins

- usermod: Modify user
- -aG: Add to group (append)

- docker: Group name
- jenkins: User to add

Purpose: Make jenkins user part of docker group

Command 3: usermod -aG docker ubuntu

- Same as above
- Makes ubuntu user part of docker group

Why this is needed: Docker installation creates a group called docker. Whoever wants to create containers or whoever wants to access Docker, they should be part of the docker group.

Command 4: systemctl restart docker

- Restart Docker daemon
- Apply changes
- Required for permissions to take effect

After doing this, we are restarting the Docker daemon.

STEP 4: TEST DOCKER ACCESS AS JENKINS USER

Switch to jenkins user:

su - jenkins

Test Docker access:

docker run hello-world

Expected result: Container runs successfully.

If permission denied:

Steps to fix:

1. Logout from jenkins user:

exit

2. Re-add jenkins to docker group:

usermod -aG docker jenkins

3. Switch back to jenkins user:

su - jenkins

4. Run command again:

docker run hello-world

This command will be executed successfully.

Output:

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

...

Hello from Docker!

This message shows that your installation appears to be working correctly.

STEP 5: RESTART JENKINS

Why restart: Sometimes our Jenkins might not pick up these changes immediately.

How to restart Jenkins:

Go to browser where Jenkins was running.

Change the URL to:

http://54.123.45.67:8080/restart

Press Enter.

Confirmation page appears.

Click "Yes" to restart.

Jenkins restarts (takes 30-60 seconds).

We will see login page again. Log in with credentials created earlier.

Jenkins restarted successfully.

STEP 6: INSTALL DOCKER PIPELINE PLUGIN

Why do we need this plugin: This is done in order to run Docker as agent.

Explanation: Jenkins should understand that whenever I am executing a job, I have to make sure that if user provides in the Jenkins file to run this specific job on Docker, I need to have the configuration to do so.

This is the reason why you will install Docker Pipeline plugin so that Jenkins will execute your pipeline on Docker agent if the required information is provided in the Jenkins file.

Installing the plugin:

Step 1: Click on "Manage Jenkins" (left sidebar)

Step 2: Click on "Plugins"

Step 3: Click on "Available plugins" tab

Step 4: Search for "Docker Pipeline"

Step 5: Check the checkbox next to "Docker Pipeline"

Step 6: Click "Install" button at bottom

Installation starts.

Wait for installation to complete.

STEP 7: RESTART JENKINS AGAIN

After plugin installation:

Go to browser and change URL to:

http://54.123.45.67:8080/restart

Press Enter and confirm restart.

Jenkins restarts with new plugin.

Log in again.

UNDERSTANDING JENKINS JOB TYPES

Click on "New Item" on Jenkins dashboard.

You will see multiple ways of creating Jenkins pipeline:

- Freestyle project
- Pipeline
- Multi-configuration project
- Folder

- Multibranch pipeline
- Organization folder

Why so many options?

FREESTYLE PROJECT (OLD APPROACH)

What it was: When we go back 5 years, we only had Freestyle projects.

How it worked:

- Provide all details in input fields
- Click radio buttons
- Select dropdowns
- Configure through UI
- No need for coding

Example Freestyle project:

1. Project name: MyApp
2. Source Code Management: Git
3. Repository URL: (paste URL)
4. Build Triggers: Poll SCM
5. Build Steps: Execute shell
6. Post-build Actions: Archive artifacts

Everything configured through UI.

PROBLEM WITH FREESTYLE

The major drawback: This is not a declarative approach. This approach cannot be shared with a team member.

Explanation:

What we do traditionally: If you want to modify project and send for peer review, we raise a pull request on GitHub and some of our peers will review it.

Problem with Freestyle:

- Configuration stored in Jenkins database

- Not in version control
- Cannot create pull request for Jenkins config
- Cannot review changes easily
- Cannot track who changed what
- Cannot rollback changes

The workflow that is followed by most tools:

- Code in version control (Git)
- Peer review via pull requests
- Track changes
- Rollback capability

Freestyle cannot go through all of this workflow.

PIPELINE APPROACH (MODERN SOLUTION)

To overcome this problem and many other problems: Jenkins has come up with the pipeline approach.

What is pipeline approach: The biggest advantage is you can write a declarative or you can write a scripted pipeline.

What this means: We can write our pipeline as code here and we can put this entire thing in Git repository.

Code language: The code is written in Groovy scripting language.

Benefits:

- Pipeline as code
- Version controlled
- Peer review possible
- Track changes
- Rollback capability
- Reusable
- Shareable

STEP 8: CREATE FIRST PIPELINE

Click "New Item".

Enter name:

my-first-pipeline

Select: Pipeline

Click "OK".

PIPELINE CONFIGURATION PAGE

Description: Jenkins is all about picking up the code from your VCS (Version Control System) and delivering it to production or some staging environment with automation of all the steps that are involved in between.

Example: If there are 10 stages and your job is to automate 10 stages in Jenkins, Jenkins acts as an orchestrator.

PIPELINE SCRIPT SECTION

In the Pipeline section:

Definition dropdown: Pipeline script

You can see sample code in the script section.

If you choose "GitHub + Maven": A code will appear in code block.

Example code shown:

groovy

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                git 'https://github.com/example/repo.git'
                sh 'mvn clean install'
            }
```

```
    }
    stage('Test') {
      steps {
        sh 'mvn test'
      }
    }
  }
}
```

Understanding sample code:

This code tells: There are multiple stages in this Jenkins pipeline.

First stage: Build stage In build stage it is:

- Getting the code from GitHub (checking out code)
- Executing a Maven target
- Verifying if JUnit coverage is proper
- Archiving the artifact (JAR or WAR) which is created at the end

There may be many stages or fewer stages depending on organization.

Inside each stage: We have steps to be followed.

PIPELINE SYNTAX HELPER

Feature available: Pipeline Syntax link at bottom of script section.

What it does: Check the Groovy syntax for a particular thing.

How to use:

1. Click "Pipeline Syntax"
2. Select action from dropdown
3. Fill in required information
4. Click "Generate Pipeline Script"
5. Code will be generated

6. Copy and paste in your pipeline

Benefit: Don't need to memorize Groovy syntax.

STEP 9: CREATE JENKINSFILE IN GIT REPOSITORY

For this project: We will select Definition as "Pipeline script from SCM".

Why: Store pipeline code in Git repository for version control.

Create folder in your Git repository:

Folder name: my-first-pipeline

Create file: Jenkinsfile

File content:

```
groovy
pipeline {
  agent {
    docker { image 'node:16-alpine' }
  }
  stages {
    stage('Test') {
      steps {
        sh 'node --version'
      }
    }
  }
}
```

Commit this file to your repository.

Jenkinsfile explanation:

Line 1: pipeline {

Start of pipeline definition

Lines 2-4: agent { docker { image 'node:16-alpine' } }

- agent: Where to run pipeline

- docker: Use Docker container

- image: Use Node.js version 16 Alpine Linux image

Lines 5-11: stages { stage('Test') { steps { ... } } }

- stages: Collection of stages

- stage('Test'): Stage named Test

- steps: Actions to perform

- sh 'node --version': Run shell command to check Node.js version

Purpose:

This pipeline will create a Docker container with Node.js 16 and verify the version.

STEP 10: CONFIGURE JENKINS JOB

Back in Jenkins (http://54.123.45.67:8080):

We selected Definition as "Pipeline script from SCM".

Now configure:

SCM dropdown: Git

Repository URL:

Paste your Git repository URL

```
https://github.com/yourusername/devops.git
```

Credentials:

We need not have to give any credentials because it is our public repository.

If private repository:

Click "Add" and provide GitHub username and password/token.

Branches to build:

Branch Specifier: */main

Script Path:

```
my-first-pipeline/Jenkinsfile
```

This tells Jenkins where to find the Jenkinsfile in the repository.

Click "Save".

STEP 11: RUN THE PIPELINE

Current situation:

Right now we just have one EC2 instance and have no worker nodes.

Question:

Where is this pipeline going to be executed?

Answer:

On Docker container on the Jenkins Master itself.

Execute pipeline:

Click "Build Now" button on left sidebar.

Build starts:

You will see build number appearing (e.g., #1).

Click on build number: #1

Click "Console Output" to see logs.

CONSOLE OUTPUT EXPLANATION

Output shows:

Step 1: Fetch code from GitHub

```
Started by user admin
Checking out git https://github.com/yourusername/devops.git
Commit: abc123def456
```

First, it will fetch the code from GitHub.

Step 2: Check for Docker image

```
Pulling Docker image node:16-alpine
```

After that, it is pulling an image called node:16-alpine.

What happened:

It checked whether there is any Docker container or Docker image on that node or on that Jenkins Master.

Result:

It didn't find any image.

Action:

So it tried to create a Docker image.

Step 3: Create container and run pipeline

```
Creating container with image node:16-alpine

Running on container abc123

[Test] $ sh node --version

v16.20.0
```

Executed the entire pipeline and verified the Node version.

Step 4: Build successful

```
Finished: SUCCESS
```

WHAT HAPPENS AFTER BUILD

Once the build is successful:

Jenkins will delete this Docker container.

Verify:

Go to terminal and run:

```

docker ps

```

Output:

```

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Empty output means no containers running.

What basically happened:

1. Jenkins requested Docker to create one container
2. Used Docker Pipeline plugin that we configured
3. Jenkins spoke to Docker
4. Asked for a container to run a pipeline which is Node.js-related application
5. Container was created
6. Pipeline executed
7. As soon as the process is completed, the container got destroyed

Benefit: Only when there is a request, a container is created. This is how we save a lot of cost.

---

WHY THIS APPROACH IS BENEFICIAL

Comparison with worker nodes approach:

Worker Nodes (Old approach):

- 3 EC2 instances always running
- Want to change Node.js version: Manual login, manual upgrade
- Risk of conflicts
- Always consuming resources
- High cost

Docker Containers (New approach):
```

- Containers created on demand
- Want to change Node.js version: Modify Jenkinsfile, change image version
- No conflicts (isolated containers)
- Resources used only when needed
- Low cost

If we want to change the version of any dependency: We just have to make changes in the Jenkins pipeline code.

Example: Change from Node.js 16 to Node.js 18:

groovy

```
agent {
  docker { image 'node:18-alpine' }
}
```

Save, commit, next run uses Node.js 18. Simple.

MULTI-STAGE MULTI-AGENT PIPELINE

SCENARIO: 3-TIER APPLICATION

You are working in an organization with current application with database.

Application architecture: 3-tier application:

- Frontend
- Backend
- Database

Requirements:

Database CI/CD: Should be executed on VM which has CentOS.

Frontend CI/CD: Should be run on VMs that have Ubuntu or Node.js installed.

Backend CI/CD: Should be run on VMs that have Ubuntu or Java installed.

Problem: How to solve this? Different stages need different environments.

SOLUTION: MULTI-AGENT PIPELINE

In the pipeline code:

Top-level agent: none

Why agent none: Because we will specify different agents for different stages.

Backend stage agent:

groovy

docker { image 'maven:3.8.1-adoptopenjdk-11' }

Frontend stage agent:

groovy

docker { image 'node:16-alpine' }

Database stage agent:

groovy

docker { image 'mysql:latest' }

COMPLETE PIPELINE CODE

Create new Jenkinsfile with this code:

groovy

```
pipeline {
 agent none
 stages {
  stage('Back-end') {
   agent {
    docker { image 'maven:3.8.1-adoptopenjdk-11' }
   }
   steps {
    sh 'mvn --version'
   }
  }
```

```
    stage('Front-end') {
      agent {
        docker { image 'node:16-alpine' }
      }
      steps {
        sh 'node --version'
      }
    }
    stage('DB') {
      agent {
        docker { image 'mysql:latest' }
      }
      steps {
        sh 'echo "DB stage executed successfully."'
      }
    }
  }
}
```

Code explanation:

agent none:

No default agent. Each stage specifies its own agent.

Stage 1: Back-end

- Uses Docker image: maven:3.8.1-adoptopenjdk-11

- Has Java 11 and Maven 3.8.1

- Runs: mvn --version

- Verifies Maven installed

Stage 2: Front-end

- Uses Docker image: node:16-alpine

- Has Node.js 16

- Runs: node --version

- Verifies Node.js installed

Stage 3: DB

- Uses Docker image: mysql:latest

- Has MySQL database

- Runs: echo command

- Simulates database operations

STEP 12: CREATE NEW JENKINS JOB FOR MULTI-STAGE

Click "New Item".

Name: multi-stage-pipeline

Select: Pipeline

Click "OK".

Configuration:

Definition: Pipeline script from SCM

SCM: Git

Repository URL: (your repo URL)

Branch: */main

Script Path: path/to/multi-stage/Jenkinsfile

Click "Save".

STEP 13: RUN MULTI-STAGE PIPELINE

Click "Build Now".

What should happen:

3 containers should be created (one for each stage).

Verify containers running:

During build, go to MobaXterm terminal.

Run command:

```
docker ps
```

Output example:

```
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS     NAMES
f9be63276a5f   node:16-alpine   "docker-entrypoint.s..."   1 second ago   Up Less than a second             objective_sanderson
```

This shows Docker container running for Frontend stage.

As pipeline progresses:

- Backend stage: Creates Maven container
- Frontend stage: Creates Node container
- DB stage: Creates MySQL container

Each container:

- Created when stage starts
- Runs stage steps
- Deleted when stage completes

Console output shows:

```
[Back-end] Running on container abc123
[Back-end] $ sh mvn --version
Apache Maven 3.8.1
...

[Front-end] Running on container def456
[Front-end] $ sh node --version
v16.20.0
...

[DB] Running on container ghi789
[DB] $ sh echo "DB stage executed successfully."
DB stage executed successfully.
...

Finished: SUCCESS
```

All stages executed successfully on different Docker containers.

After completion:

All containers deleted.

Verify:

```
docker ps
```

```

Output: Empty (no containers running)

---

JENKINS INTERVIEW QUESTIONS

---

QUESTION 1: EXPLAIN THE CI/CD WORKFLOW IN YOUR ORGANIZATION

Answer (refer to flowchart):

Our CI/CD workflow:

Step 1: SCM Commit

Developer commits code to Git repository (GitHub/GitLab/Bitbucket).

Step 2: Repository and Branch Name

Jenkins identifies which repository and which branch received the commit.

Step 3: Workflow Start

Jenkins pipeline triggers automatically.

Step 4: SCM Checkout

Jenkins checks out the code from repository.
```

Step 5: Collect Dependent Repos

If application has dependencies, Jenkins fetches them.

Step 6: Parallel Stages

Stage 1: Build/Launch (Docker)

- Build Docker images
- Launch containers
- Build modules in parallel
  - Build/Launch Module 1
  - Build/Launch Module 2
  - Build/Launch Module N

Stage 2: Test

- Integration Test Module 1
- Integration Test Dependency N
- Smoke Test
- End-to-End Test

All tests run in parallel for speed.

Stage 3: Deploy

Deploy application to target environment (Dev/Staging/Production).

Step 7: Workflow End

Pipeline completes.

Step 8: Generate Report

On catch/finally: build-report.html generated with all logs and test results.

Environments:

- Left side: Development environment

- Right side: Production environment

Pipeline promotes application from Development to Production through these stages.

QUESTION 2: HOW DO YOU HANDLE ISSUES IN WORKER NODES?

Examples of issues:

- Node not responding

- Node running out of disk space

- Node running out of memory

- Jenkins job stuck on worker node

Answer:

Approach 1: Manual investigation

I will log into this worker node and try to understand what is the problem by looking at its health.

Commands to check:

```

df -h (check disk space)

free -g (check memory)

top (check CPU and processes)
```

systemctl status jenkins (check Jenkins agent status)

Based on findings, take corrective action.

Approach 2: Automated monitoring I can write a simple application or a script to monitor the node's health and send alerts when:

- Memory out of space
- Disk space low
- CPU usage high
- Node disconnected

Example monitoring script:

bash

```
#!/bin/bash
DISK_USAGE=$(df -h / | tail -1 | awk '{print $5}' | sed 's/%//')
if [ $DISK_USAGE -gt 80 ]; then
   echo "Disk usage is ${DISK_USAGE}%!" | mail -s "Alert" admin@example.com
fi
```

Run this script via cron job every 10 minutes.

Approach 3: Auto-scaling We can implement auto-scaling on the EC2 instances.

How it works:

- Monitor Jenkins queue
- If jobs waiting: Launch new worker node
- If nodes idle: Terminate excess nodes

Tools:

- AWS Auto Scaling Groups
- Jenkins EC2 Plugin

Approach 4: Best approach - Docker agents The best way is what we have implemented in the above two projects.

Why Docker agents are better:

Problem with worker nodes:

- Node crashes: Jobs fail
- Node full: Cannot run more jobs
- Node slow: All jobs slow
- Manual maintenance required

With Docker agents:

- Container crashes: Create new container
- Master node full: Use cloud-based runners
- One container slow: Others unaffected
- Automatic cleanup

Benefits:

- No worker node management
- Containers automatically created/destroyed
- Failures isolated
- Automatic recovery
- No idle resources

Implementation: Use Docker containers as agents instead of EC2 worker nodes. This eliminates most worker node issues.

ADDITIONAL INTERVIEW QUESTIONS

QUESTION 3: What is Jenkins?

Answer: Jenkins is an open-source automation server used for CI/CD. It automates building, testing, and deploying applications.

QUESTION 4: What is a Jenkins pipeline?

Answer: Jenkins pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines. It's defined in a Jenkinsfile using Groovy syntax.

QUESTION 5: Difference between Declarative and Scripted pipeline?

Answer:

Declarative pipeline:

- Simpler syntax
- Pre-defined structure
- Easier to learn
- Recommended for most use cases
- Example:

```
groovy
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'make'
      }
    }
  }
}
```

Scripted pipeline:

- More flexible
- Full Groovy programming
- Complex logic possible
- Harder to learn
- Example:

```
groovy
node {
  stage('Build') {
    sh 'make'
  }
}
```

QUESTION 6: What are Jenkins agents?

Answer: Jenkins agents (or nodes) are machines that execute jobs sent by Jenkins Master. Can be physical machines, VMs, or Docker containers.

QUESTION 7: What is Jenkinsfile?

Answer: Jenkinsfile is a text file containing Jenkins Pipeline definition. Stored in source control repository, enabling version control and code review.

QUESTION 8: How to secure Jenkins?

Answer:

- Enable authentication
- Use role-based access control
- Keep Jenkins updated
- Use credentials plugin for secrets
- Enable CSRF protection
- Use HTTPS
- Limit exposed ports

QUESTION 9: What are Jenkins plugins?

Answer: Extensions that add functionality to Jenkins. Examples: Git plugin, Docker plugin, AWS plugin, Email plugin.

QUESTION 10: Difference between agent any and agent none?

Answer:

agent any:

- Use any available agent
- Jenkins chooses agent
- Same agent for all stages

agent none:

- No default agent
- Each stage must specify own agent
- Different agents for different stages

SUMMARY

Jenkins installation:

1. Install Java
2. Add Jenkins repository
3. Install Jenkins
4. Open port 8080
5. Access web interface
6. Install plugins
7. Create admin user

Jenkins with Docker:

1. Install Docker
2. Add jenkins user to docker group
3. Restart Docker
4. Install Docker Pipeline plugin
5. Use Docker images as agents

Benefits of Docker agents:

- No version conflicts

- Lightweight
- Created on demand
- Deleted after use
- Cost-effective
- No idle resources

Multi-stage pipeline:

- Different agents for different stages
- agent none at pipeline level
- Specify agent per stage
- Maximum flexibility

Handling worker node issues:

- Manual: SSH and investigate
- Automated: Monitoring scripts
- Auto-scaling: Dynamic capacity
- Best: Use Docker agents (eliminates most issues)

This is modern Jenkins CI/CD setup with Docker integration.