yatlas

JENKINS INTRO

(RESOURCES USED)
-youtube:
- DEVOPS JOURNEY

-deepseek:

The Core Concept: What is Jenkins?

At its heart, Jenkins is an open-source automation server. It is the most popular tool for implementing CI/CD (Continuous Integration and Continuous Delivery/Deployment).

Think of it as a robotic team member whose only job is to:

1. Watch your code repository (e.g., on GitHub, GitLab).
2. Get the latest code whenever a change is made.
3. Build that code (compile, test, package).
4. Report back if anything broke.
5. Deploy the code to servers if everything is okay.

This automates the repetitive parts of software development, leading to higher quality and faster releases.

Jenkins Architecture: The Master-Agent Model

Jenkins uses a distributed architecture, often called a **master-agent (or controller-agent)** model. This is designed to handle large projects and distribute the workload.

1. Jenkins Server (Master/Controller)

This is the main Jenkins server that you install first. It's the brain of the operation.

- What it does:
    - Serves the web UI that you interact with.
    - Stores all configuration data (credentials, job definitions, etc.).
    - Schedules and manages build jobs.
    - Tracks the status of agents.
    - It can also execute jobs itself, but this is not recommended for production workloads.

2. Agents (Formerly "Slaves")

These are separate machines (which can be physical servers, virtual machines, or containers) that are connected to the master.

- What they do:
  - They do the actual work. The master delegates build jobs to agents.
  - They listen for instructions from the master.
  - They have the necessary environments to run builds (e.g., specific JDK versions, Node.js, build tools like Maven or Gradle).

Why use agents?

- Distribute Load: You can run multiple builds in parallel on different agents.
- Different Environments: You can have agents with different operating systems (Windows, Linux, macOS) and toolchains to test your software on various platforms.
- Security: Isolate build environments from the main Jenkins controller.

How they connect: Agents connect to the master using a Java network protocol (JNLP) or via SSH. The master is always in control.

Key Jenkins Components You Will Interact With

1. Dashboard

- **The home page you see after logging in. It gives you an overview of all your jobs, build queues, and agent status.**

2. Jobs (or "Projects")

- **This is the heart of your work. A job is a configuration that tells Jenkins what to do. There are different types:**

Freestyle Project: The most flexible and common type for beginners. You configure build steps through a GUI.

Pipeline: The modern, recommended way to define jobs. You write a script (a Jenkinsfile) in code, which is stored in your repository. This is known "Pipeline-as-Code" and is much more powerful and maintainable.

(WE WILL GET INTO MORE DETAILS FOR THE PIPELINE PART)

Multibranch Pipeline: A special type of Pipeline that automatically creates a job for each branch in your repository (e.g., main, dev, feature-x).

3. Build

- **A single execution of a job. If your job is a recipe, a build is you following that recipe once. Each build has a history, logs, and artifacts.**

4. Workspace

- **A directory on the agent's machine where a build happens. Jenkins checks out your source code into the workspace and performs all its steps there.**

5. Plugins

- **These are extensions that give Jenkins superpowers. Jenkins' core is very simple; almost all its functionality (integrating with GitHub, building Java code, deploying to AWS, etc.) comes from plugins.**
- Examples: Git plugin, Docker plugin, Pipeline plugin, Blue Ocean plugin (for a nicer UI), Email notification plugin.

6. Node

- **A general term for any machine that is part of the Jenkins architecture, be it the master or an agent.**

7. Executor

- **A slot for running a build on a node. A node can have multiple executors, meaning it can run multiple builds concurrently. An executor is essentially a thread on the agent machine.**

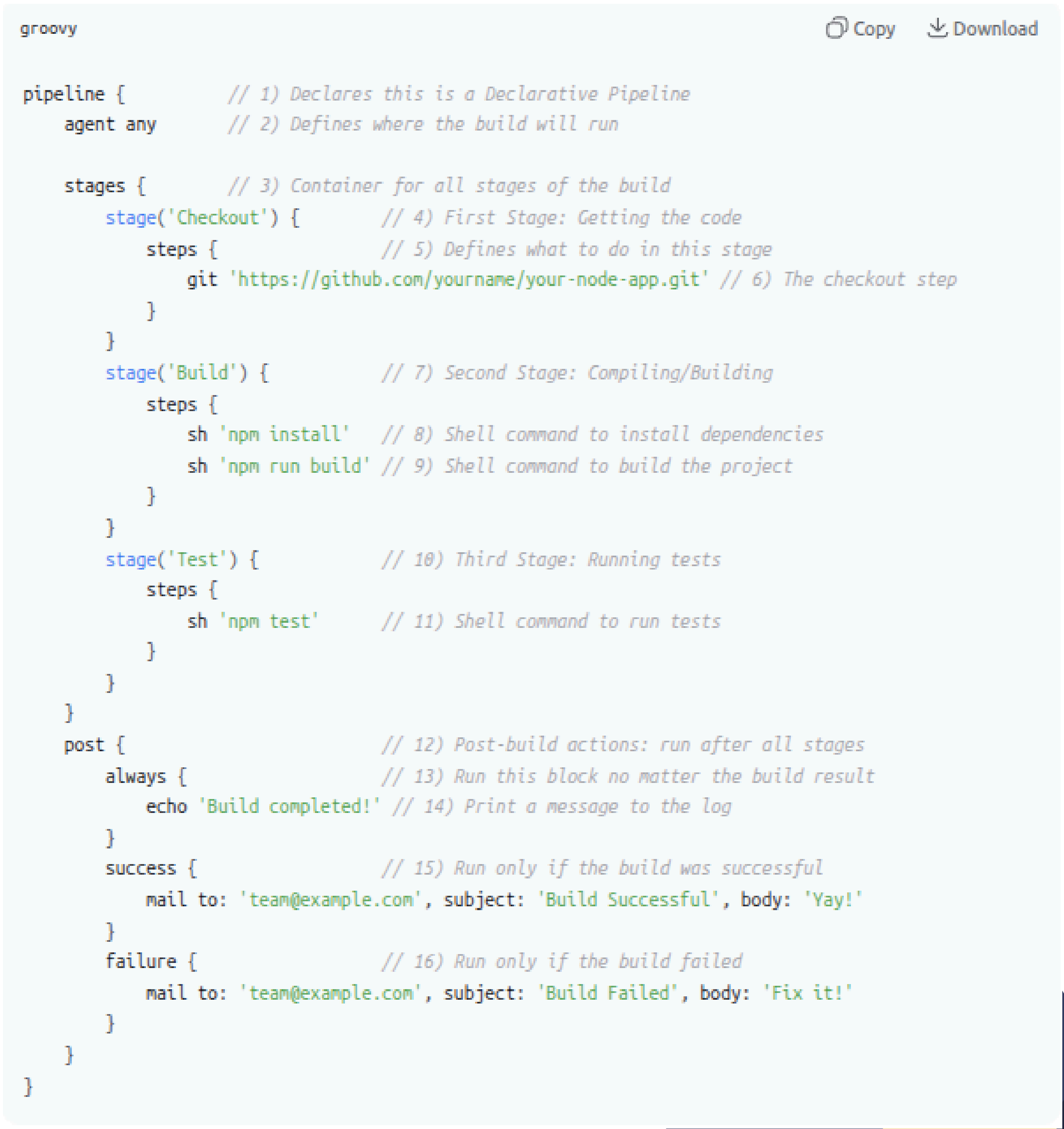
Leveling Up: Using a Pipeline (The Better Way)

Instead of configuring everything in the UI, you create a Jenkinsfile in the root of your project repository.

Example Jenkinsfile (Declarative Pipeline):

The example Jenkinsfile I provided is a classic and complete example of Continuous Integration (CI).

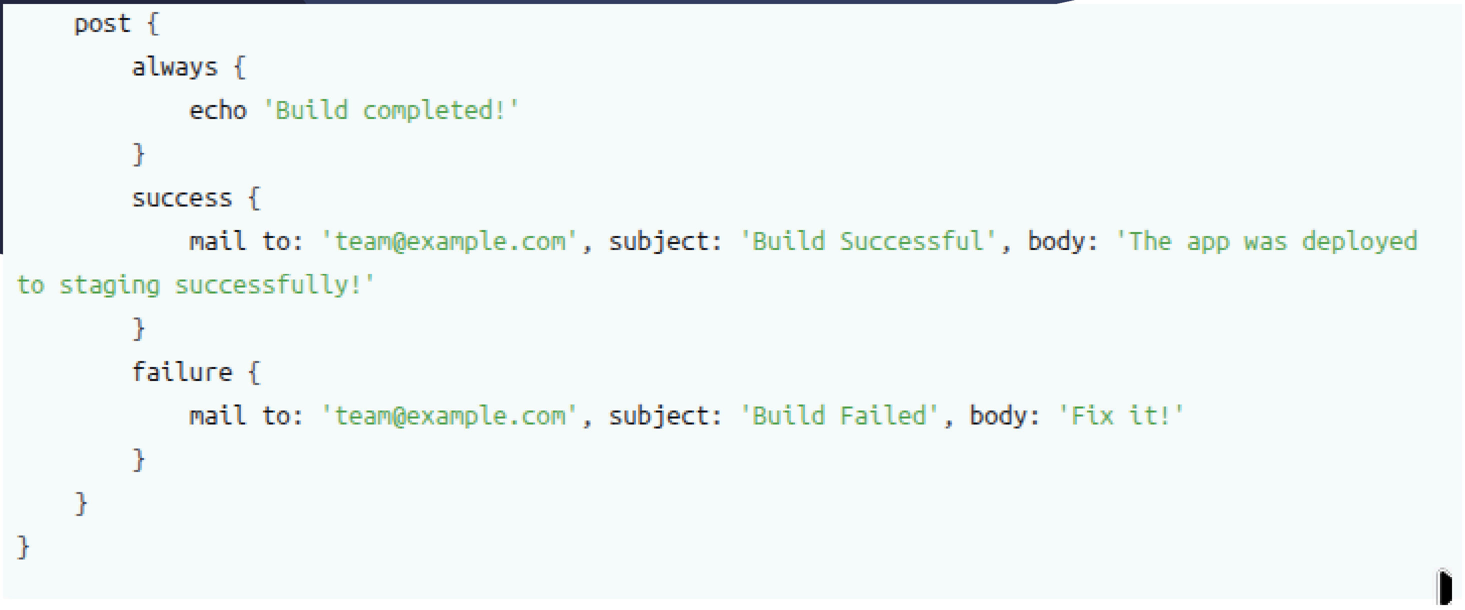
- CI (Continuous Integration): The practice of automatically building and testing code every time a developer pushes a change. The goal is to find bugs early and ensure the codebase is always in a working state.
    - Our Example Does CI: It Checkouts, Builds, and Tests the code. This validates that a new code change integrates successfully with the existing code.



```groovy
pipeline {          // 1) Declares this is a Declarative Pipeline
    agent any       // 2) Defines where the build will run

    stages {        // 3) Container for all stages of the build
        stage('Checkout') {       // 4) First Stage: Getting the code
            steps {               // 5) Defines what to do in this stage
                git 'https://github.com/yourname/your-node-app.git' // 6) The checkout step
            }
        }
        stage('Build') {          // 7) Second Stage: Compiling/Building
            steps {
                sh 'npm install'   // 8) Shell command to install dependencies
                sh 'npm run build' // 9) Shell command to build the project
            }
        }
        stage('Test') {           // 10) Third Stage: Running tests
            steps {
                sh 'npm test'      // 11) Shell command to run tests
            }
        }
    }
    post {                        // 12) Post-build actions: run after all stages
        always {                  // 13) Run this block no matter the build result
            echo 'Build completed!' // 14) Print a message to the log
        }
        success {                 // 15) Run only if the build was successful
            mail to: 'team@example.com', subject: 'Build Successful', body: 'Yay!'
        }
        failure {                 // 16) Run only if the build failed
            mail to: 'team@example.com', subject: 'Build Failed', body: 'Fix it!'
        }
    }
}
```

Let's add a CD (Delivery) stage to our example. We'll assume we are packaging our app and deploying it to a staging environment for further testing (a common step before production).

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/yourname/your-node-app.git'
            }
        }
        stage('Build') {
            steps {
                sh 'npm install'
                sh 'npm run build'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
        // --- NEW CD STAGES ADDED BELOW ---
        stage('Package') {
            steps {
                sh 'npm run package' // Or use a tool like 'tar' to create a deployable artifact
                archiveArtifacts artifacts: 'dist/**/*' // Archives the build results for download
            }
        }
        stage('Deploy to Staging') {
            steps {
                // This step is highly dependent on your environment
                // Example 1: Deploy to a cloud platform (e.g., AWS S3, Elastic Beanstalk)
                sh 'aws s3 sync ./dist s3://my-staging-bucket/'

                // Example 2: Deploy via SSH (requires SSH Agent plugin)
                // sshagent(['staging-server-credentials']) {
                //     sh 'scp -r ./dist user@staging-server:/path/to/app'
                // }

                // Example 3: Use a Docker plugin to build and push an image
                // sh 'docker build -t my-app:staging .'
                // sh 'docker push my-app:staging'
                // sh 'kubectl set image deployment/my-app my-app=my-app:staging'
            }
        }
    }
```



```
    post {
        always {
            echo 'Build completed!'
        }
        success {
            mail to: 'team@example.com', subject: 'Build Successful', body: 'The app was deployed
to staging successfully!'
        }
        failure {
            mail to: 'team@example.com', subject: 'Build Failed', body: 'Fix it!'
        }
    }
}
```

Now this pipeline is a full CI/CD pipeline. It not only builds and tests but also packages the application and deploys it to a staging environment.

Who Writes the Jenkinsfile?

This is a collaborative effort, and the lines can blur, but here's the typical breakdown:

DevOps Engineers / Platform Engineers:

- **Responsibility**: They design the foundation.
- **They create:** The initial Jenkins infrastructure (master, agents, security, plugins). They often write the first version of the Jenkinsfile or a shared pipeline library that defines reusable patterns for building, testing, and deploying. They ensure the pipeline has access to secrets (passwords, keys) securely.

Developers:

- **Responsibility**: They own the application-specific logic inside the pipeline.
- **They modify:** The Jenkinsfile that lives in their application's repository. They are the experts on how their app is built and tested. They might change the Node.js version, add a new test command, or define a new packaging step. The DevOps team gives them the tools and templates; the developers use them.

The Modern "You Build It, You Run It" Philosophy: The trend is for development teams to own their entire pipeline. The DevOps team provides the platform and guardrails, and the developers write and maintain the Jenkinsfile for their service. This empowers developers and reduces bottlenecks.

yatlas

Is the Jenkinsfile All There Is To It?

- **No, absolutely not. The Jenkinsfile is the most visible part, but it's just the tip of the iceberg. Here's what else you need to know and set up:**

Component	Description	Why It's Important
Plugins	The heart of Jenkins' functionality. Need to integrate with GitHub? Docker? AWS? Slack? There's a plugin for that.	You must install and configure the right plugins (e.g., Git, Pipeline, Docker, Blue Ocean, SSH Agent).
Security	Configuring user roles, permissions, and credentials.	**Crucial.** You must set up authentication (e.g., with GitHub SSO), authorize users, and securely store secrets (API keys, passwords) that the pipeline uses.
Agents/Nodes	Setting up the worker machines.	You need to configure agents with the right tools (Java, Node, Python, Docker) and connect them to the master. Often done dynamically with Docker or Kubernetes.
Shared Libraries	Central repositories of Groovy code that multiple pipelines can use.	Lets you avoid copy-pasting the same Jenkinsfile logic across 100 projects. Define a standard deployment method once and reuse it.
Triggers	How a pipeline starts.	Beyond manual triggers, you need to configure **webhooks** so that GitHub can notify Jenkins to start a build immediately on a git push.
Pipeline Syntax	Knowing more than just the basics.	Learn about parallel stages, input steps(to pause for manual approval before deploying to production), timeout, retries, and handling environment variables.

Secrets in jenkins

This is a critical security concept. Your pipeline needs to interact with external systems (e.g., GitHub to checkout code, AWS to deploy, a database to run tests). These interactions require credentials. You should never hardcode usernames, passwords, or API keys directly in your Jenkinsfile.

The Wrong Way (NEVER DO THIS):

```
stage('Deploy to AWS') {
    steps {
        sh 'aws configure set aws_access_key_id AKIAIOSFODNN7EXAMPLE' // Hardcoded key!
        sh 'aws configure set aws_secret_access_key wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY' // Hardcoded secret!
        sh 'aws s3 sync ./dist s3://my-bucket/'
    }
}
```

The Right Way: Using Jenkins' Credentials Binding

The DevOps/Platform team sets up a secure store for secrets within Jenkins. Developers can then safely reference these secrets in the pipeline without ever seeing the actual value.

- A DevOps admin stores the secret in Jenkins:
- They go to **Manage Jenkins** > **Manage Credentials**.
- They add a **new credential** of type "Secret text" or "Username with password".
- They give it a **unique ID** (e.g., aws-prod-access-key).
- The Developer references the secure credential in the Jenkinsfile:
- Jenkins provides a **withCredentials** step that temporarily makes the secret available as an environment variable during the step.

```
stage('Deploy to AWS') {
    steps {
        // This block securely injects the credentials
        withCredentials([string(credentialsId: 'aws-prod-access-key', variable: 'AWS_ACCESS_KEY_ID'),
                         string(credentialsId: 'aws-prod-secret-key', variable: 'AWS_SECRET_ACCESS_KEY')]) {
            // Inside this block, the environment variables are set
            sh 'aws s3 sync ./dist s3://my-bucket/'
        }
        // Outside this block, the environment variables are gone
    }
}
```

How it works?
When the pipeline runs, Jenkins fetches the actual secret values from its secure store and places them in the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables only for the duration of the withCredentials block. The pipeline log will automatically mask these values, so if a command prints them, it will show ** instead of the actual key.**
This is what is meant by "providing access to secrets securely." The DevOps team provides the mechanism, and the developer uses it without ever handling the raw secret.

Using Specific Agents (Beyond agent any)

Using “agent any” is fine for getting started, but in a real-world setup like this one**(5 VMs across 2 physical hosts)**, you need precision. You use labels to assign jobs to specific agents.

The Concept of Labels:

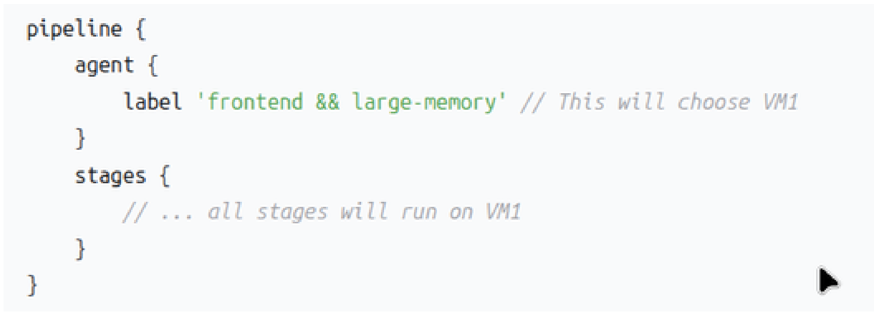
You assign one or more labels to each agent (VM). Labels are simple tags like linux, windows, docker, large-memory, frontend-builder.
In this pipeline, you tell Jenkins to run on an agent that has a specific label.
Our Example Scenario: 2 physical servers with 5 VMs:
Physical Server 1 (Big Machine):
VM1: Label it linux && frontend && large-memory (for heavy Node.js builds)
VM2: Label it linux && backend && java (for Java/Maven builds)
VM3: Label it linux && integration-tests (runs long test suites)
Physical Server 2 (Smaller Machine):
VM4: Label it windows && frontend (for testing Internet Explorer/Edge)
VM5: Label it linux && deployment (has all the CLI tools for AWS/Azure)

How to Assign Labels:

A DevOps admin configures this when adding the agent in **Manage Jenkins** > **Manage Nodes and Clouds**. They simply type the labels into the field.

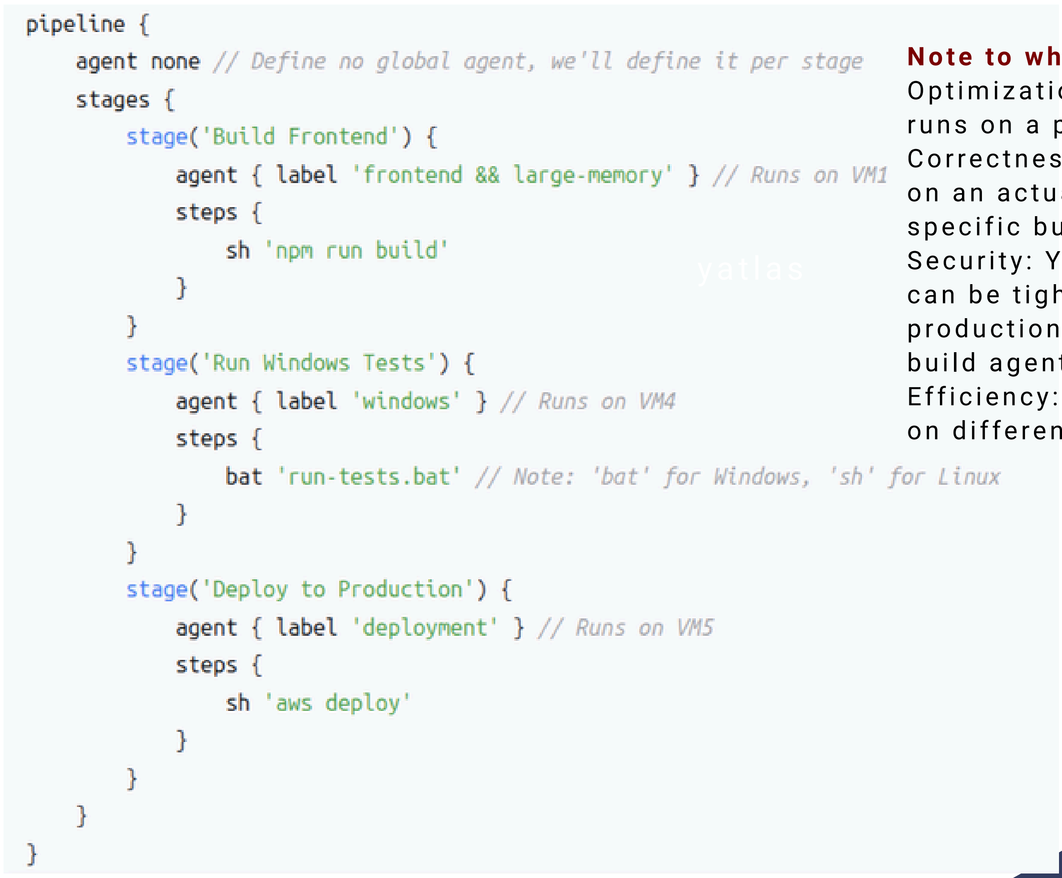
How to Use Labels in Your Jenkinsfile:

Example 1: Run the entire pipeline on a specific type of agent.



```
pipeline {
    agent {
        label 'frontend && large-memory' // This will choose VM1
    }
    stages {
        // ... all stages will run on VM1
    }
}
```

Example 2: Run different stages on different agents (Powerful feature!). This is useful if one stage has special requirements.



```
pipeline {
    agent none // Define no global agent, we'll define it per stage
    stages {
        stage('Build Frontend') {
            agent { label 'frontend && large-memory' } // Runs on VM1
            steps {
                sh 'npm run build'
            }
        }
        stage('Run Windows Tests') {
            agent { label 'windows' } // Runs on VM4
            steps {
                bat 'run-tests.bat' // Note: 'bat' for Windows, 'sh' for Linux
            }
        }
        stage('Deploy to Production') {
            agent { label 'deployment' } // Runs on VM5
            steps {
                sh 'aws deploy'
            }
        }
    }
}
```

Note to why this is powerful:
Optimization: You ensure your heavy build runs on a powerful machine.
Correctness: You run your Windows tests on an actual Windows agent, catching OS-specific bugs.
Security: Your deployment agent (VM5) can be tightly locked down and have production access credentials, while your build agents do not.
Efficiency: You can run stages in parallel on different agents.

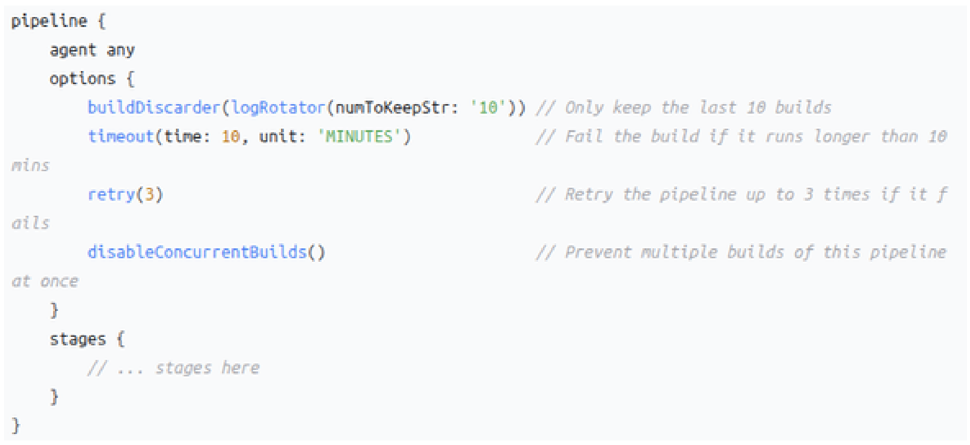
Breakdown of Other Key Directives & Steps

```
pipeline/
├── agent/
│   └── [Label, Docker, Kubernetes, or other parameters here]
├── options/
│   └── [buildDiscarder, timeout, retry, etc.]
├── environment/
│   └── [KEY = 'value', KEY = credentials('id')]
├── tools/
│   └── [Tool name and version for automatic installation]
├── parameters/
│   └── [string, choice, boolean, text parameters]
├── triggers/
│   └── [cron, pollSCM, upstream]
├── stages/                        # The core sequence of operations
│   └── stage('Stage Name A')/
│       ├── agent/                 # (Optional) Override the global agent for this stage
│       ├── environment/           # (Optional) Override the global environment for this stage
│       ├── when/                  # (Optional) Condition to execute this stage
│       ├── steps/                 # MANDATORY: The workhorse of the stage
│       │   └── [sh, bat, git, echo, script, etc.]
│       └── post/                  # (Optional) Stage-specific post actions
│           ├── always/
│           ├── success/
│           ├── failure/
│           └── changed/
│   └── stage('Stage Name B')/
│       └── ... (same structure as Stage A)
└── post/                          # Post-build actions for the entire pipeline
    ├── always/
    ├── success/
    ├── failure/
    ├── unstable/
    └── changed/
```

Let's explore the other "directories" (options, environment, tools, etc.) and some of the most important "files" (steps) you can put inside them.

1. options (Sibling of agent and stages)

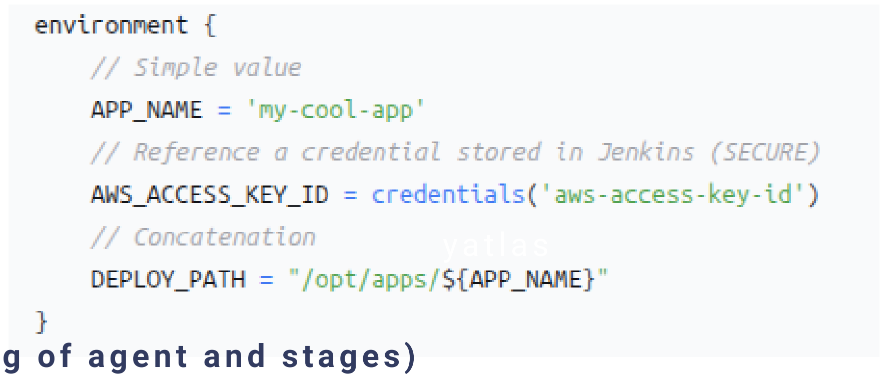
This directive allows you to configure pipeline-specific options from the Jenkins root.



```
pipeline {
    agent any
    options {
        buildDiscarder(logRotator(numToKeepStr: '10')) // Only keep the last 10 builds
        timeout(time: 10, unit: 'MINUTES')             // Fail the build if it runs longer than 10
mins
        retry(3)                                       // Retry the pipeline up to 3 times if it f
ails
        disableConcurrentBuilds()                      // Prevent multiple builds of this pipeline
at once
    }
    stages {
        // ... stages here
    }
}
```

2. environment (Sibling of agent and stages)

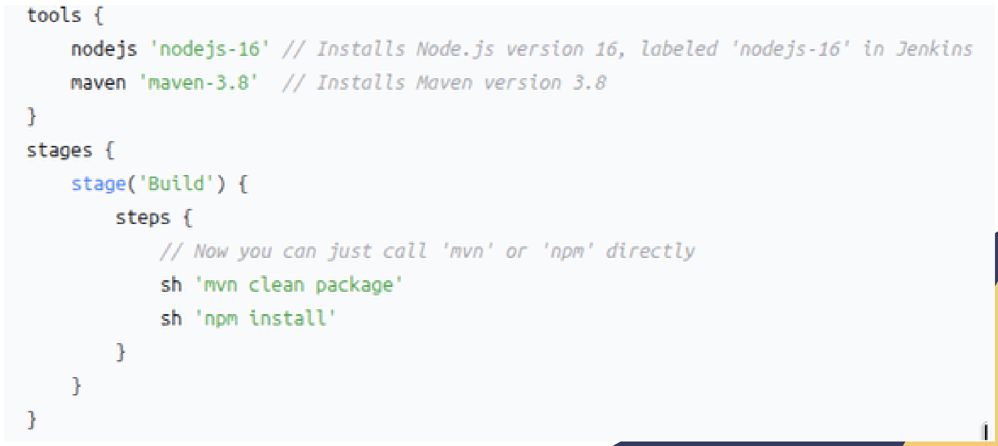
Defines environment variables that are available to all steps. This is where you put non-secret configuration



```
environment {
    // Simple value
    APP_NAME = 'my-cool-app'
    // Reference a credential stored in Jenkins (SECURE)
    AWS_ACCESS_KEY_ID = credentials('aws-access-key-id')
    // Concatenation
    DEPLOY_PATH = "/opt/apps/${APP_NAME}"
}
```

3. tools (Sibling of agent and stages)

Automatically installs tools on the agent and adds them to the PATH. Requires the tool to be pre-configured in **Manage Jenkins** > **Global Tool Configuration**.



```
tools {
    nodejs 'nodejs-16' // Installs Node.js version 16, labeled 'nodejs-16' in Jenkins
    maven 'maven-3.8'  // Installs Maven version 3.8
}
stages {
    stage('Build') {
        steps {
            // Now you can just call 'mvn' or 'npm' directly
            sh 'mvn clean package'
            sh 'npm install'
        }
    }
}
```

4. parameters (Sibling of agent and stages)

Defines user-provided parameters that trigger a new run. This creates the "Build with Parameters" button.

```
parameters {
    string(name: 'DEPLOY_ENV', defaultValue: 'staging', description: 'Environment to deploy to')
    choice(name: 'LOG_LEVEL', choices: ['INFO', 'DEBUG', 'WARN'], description: 'Set log level')
    booleanParam(name: 'SKIP_TESTS', defaultValue: false, description: 'Skip test execution?')
}
stages {
    stage('Deploy') {
        steps {
            // Parameters are available as params.DEPLOY_ENV
            echo "Deploying to environment: ${params.DEPLOY_ENV}"
            sh "deploy.sh --env ${params.DEPLOY_ENV}"
        }
    }
}
```

5. triggers (Sibling of agent and stages)

```
triggers {
    cron('H 4 * * 1-5') // Run at 4 AM every weekday (H for hash to spread load)
    pollSCM('H */4 * * *') // Poll the SCM every 4 hours
    // For webhooks (e.g., from GitHub), you typically configure the hook on the repo side,
    // not here. This is for scheduled triggers.
}
```

6. when (Child of a stage)

```
stage('Deploy to Staging') {
    when {
        // Only run this stage if we are on the main branch
        branch 'main'
        // You can also use expressions for more complex logic
        expression { return params.DEPLOY_ENV == 'staging' }
        // Or any of the built-in conditions
        environment name: 'DEPLOY_TARGET', value: 'staging'
    }
    steps {
        sh 'deploy-to-staging.sh'
    }
}
```

7. Important steps (Files inside the steps/ directory)

- sh / bat: Execute a shell (Unix) or batch (Windows) command.
- script: A necessary escape hatch. This block allows you to run arbitrary Groovy code for complex logic that the Declarative syntax doesn't support.

```groovy
steps {
    script {
        // Groovy code here
        if (env.BRANCH_NAME == 'main') {
            env.DEPLOY_TARGET = 'production'
        } else {
            env.DEPLOY_TARGET = 'staging'
        }
        // You can use loops, try-catch, etc.
    }
}
```

- echo: Print a message to the log.
- git: Check out source code from a Git repository.
- input: Pause the pipeline and wait for human input (e.g., approval to deploy to production).

```groovy
steps {
    input message: 'Deploy to production?', ok: 'Yes, deploy!'
}
```

- timeout / retry: Wrap other steps with a timeout or automatic retry logic.

```groovy
steps {
    retry(3) {
        sh './flaky-network-script.sh'
    }
    timeout(time: 5, unit: 'MINUTES') {
        sh './long-running-test-suite.sh'
    }
}
```

Full example of a pipeline (real case scenario)

SETUP

1- Agents (VMs) & Labels: Based on our previous discussion, we have:

- builder-frontend (VM1): Has Node.js, npm, and plenty of RAM. Label: frontend-builder
- builder-backend (VM2): Has JDK, Maven. Label: backend-builder
- integration-tester (VM3): Has Docker, Docker Compose. Label: integration-tester
- deployer (VM5): Has AWS CLI, Kubernetes CLI (kubectl), or other deployment tools. Label: deployer
- windows-tester (VM4): Windows VM for cross-browser testing (optional for this example). We won't use it here.

2- Credentials: The following are stored securely in Jenkins:

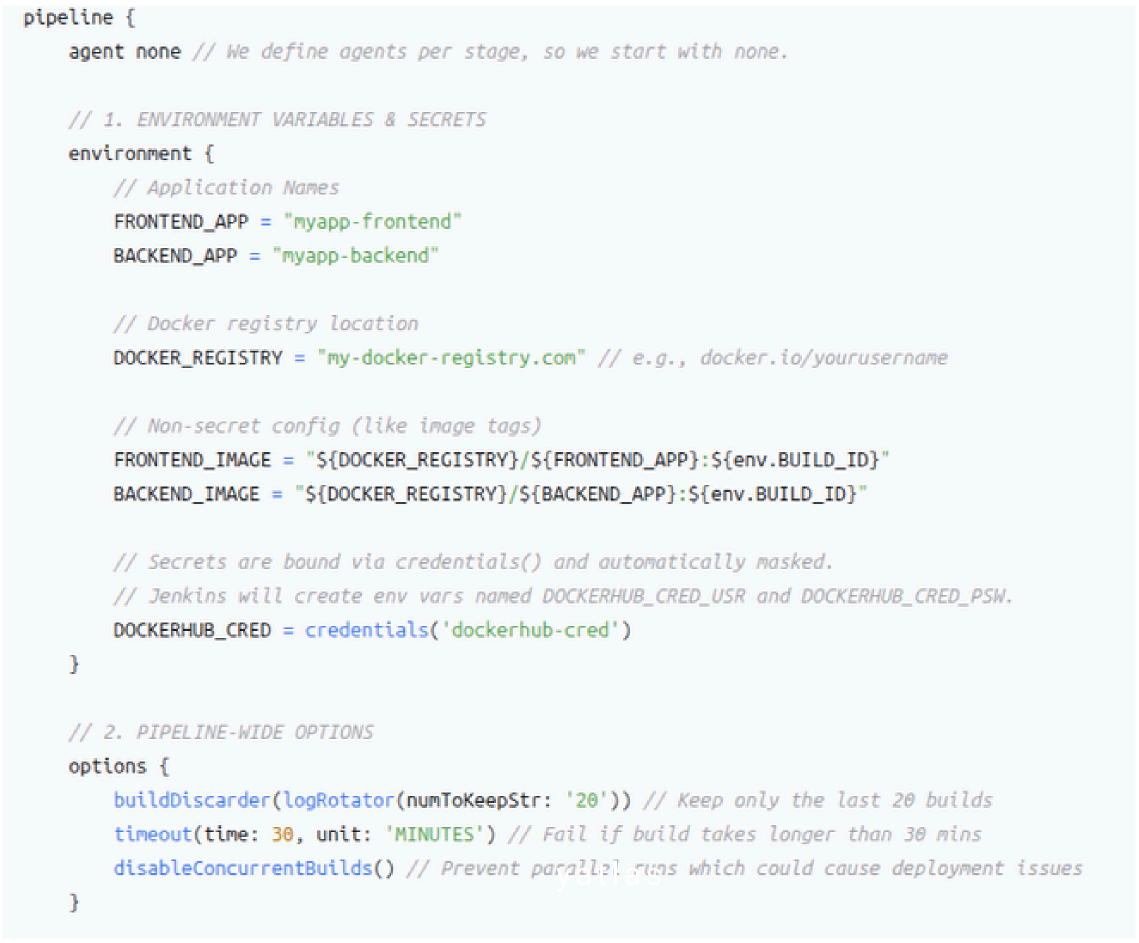
- github-cred: SSH key or user/password for the private repository.
- aws-prod-cred: AWS credentials for deployment.
- dockerhub-cred: Credentials to push images to Docker Hub (or another registry).

3- Tools: They are configured in Manage Jenkins > Global Tool Configuration:

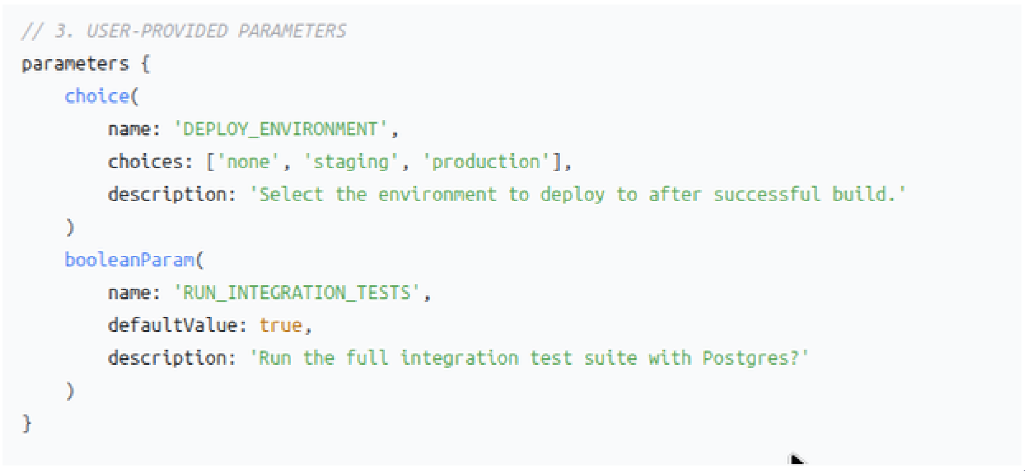
- A nodejs installation named nodejs-18.
- A maven installation named maven-3.8.

The Jenkinsfile

HERE IS THE COMPLETE PIPELINE. READ THE COMMENTS FOR A DETAILED EXPLANATION OF EACH PART.

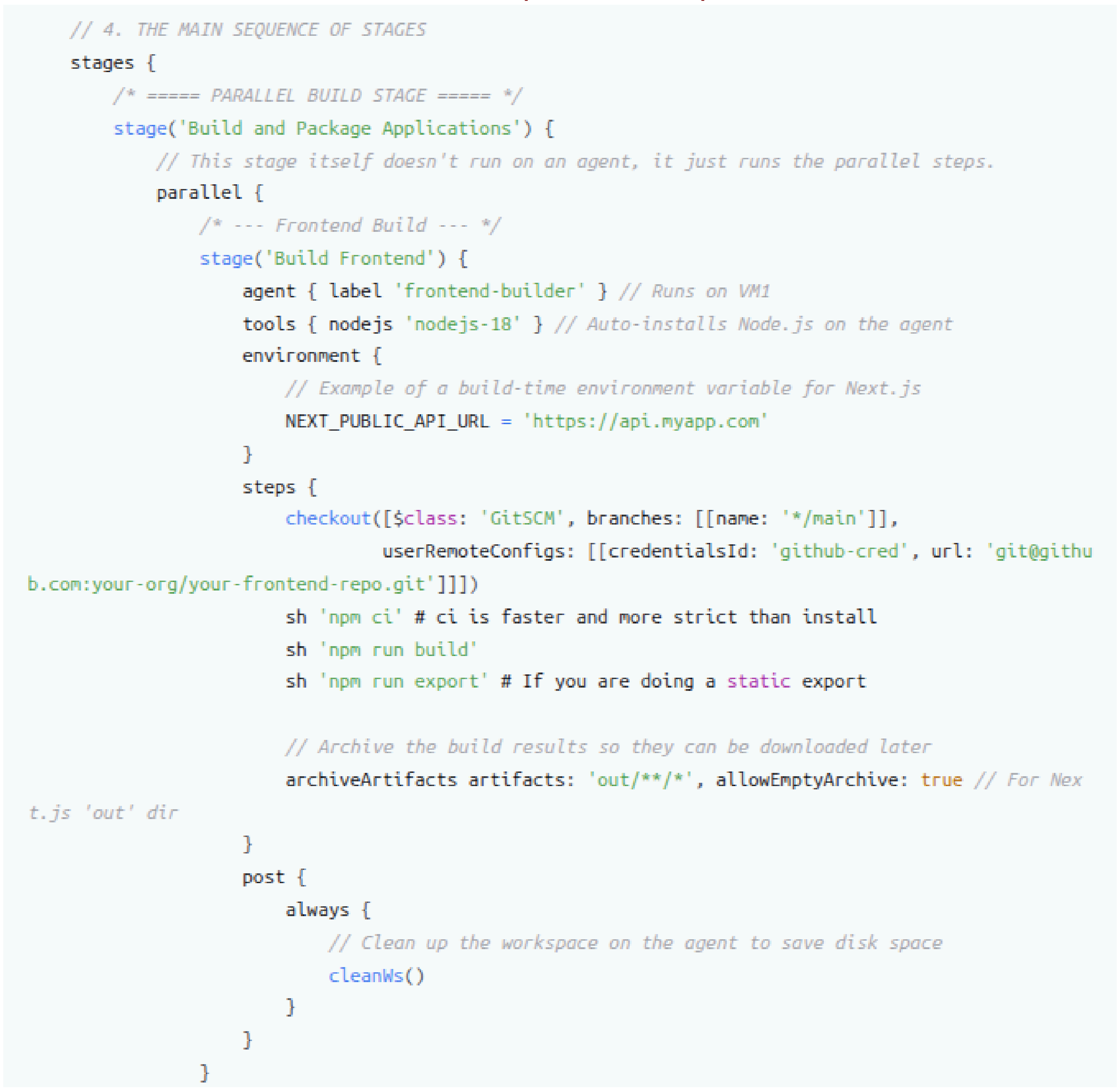


```
pipeline {
    agent none // We define agents per stage, so we start with none.

    // 1. ENVIRONMENT VARIABLES & SECRETS
    environment {
        // Application Names
        FRONTEND_APP = "myapp-frontend"
        BACKEND_APP = "myapp-backend"

        // Docker registry location
        DOCKER_REGISTRY = "my-docker-registry.com" // e.g., docker.io/yourusername

        // Non-secret config (like image tags)
        FRONTEND_IMAGE = "${DOCKER_REGISTRY}/${FRONTEND_APP}:${env.BUILD_ID}"
        BACKEND_IMAGE = "${DOCKER_REGISTRY}/${BACKEND_APP}:${env.BUILD_ID}"

        // Secrets are bound via credentials() and automatically masked.
        // Jenkins will create env vars named DOCKERHUB_CRED_USR and DOCKERHUB_CRED_PSW.
        DOCKERHUB_CRED = credentials('dockerhub-cred')
    }

    // 2. PIPELINE-WIDE OPTIONS
    options {
        buildDiscarder(logRotator(numToKeepStr: '20')) // Keep only the last 20 builds
        timeout(time: 30, unit: 'MINUTES') // Fail if build takes longer than 30 mins
        disableConcurrentBuilds() // Prevent parallel runs which could cause deployment issues
    }
```

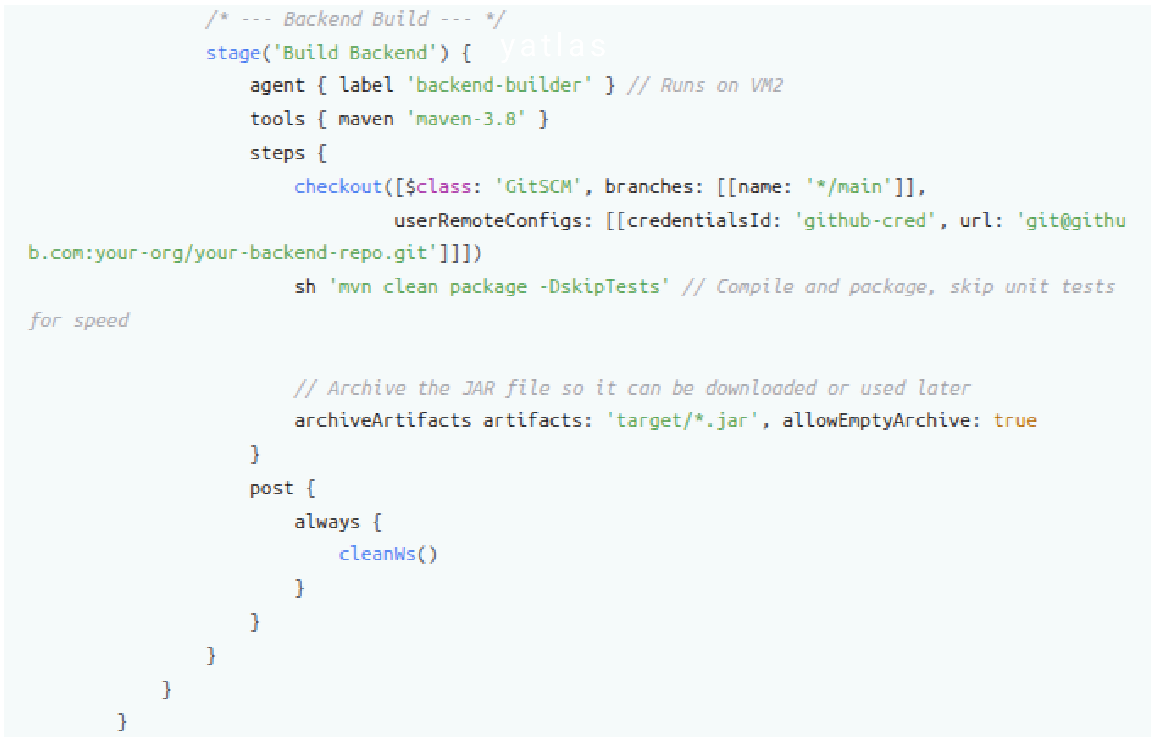


```
    // 3. USER-PROVIDED PARAMETERS
    parameters {
        choice(
            name: 'DEPLOY_ENVIRONMENT',
            choices: ['none', 'staging', 'production'],
            description: 'Select the environment to deploy to after successful build.'
        )
        booleanParam(
            name: 'RUN_INTEGRATION_TESTS',
            defaultValue: true,
            description: 'Run the full integration test suite with Postgres?'
        )
    }
```

frontend and backend parts on parallel excecution mode

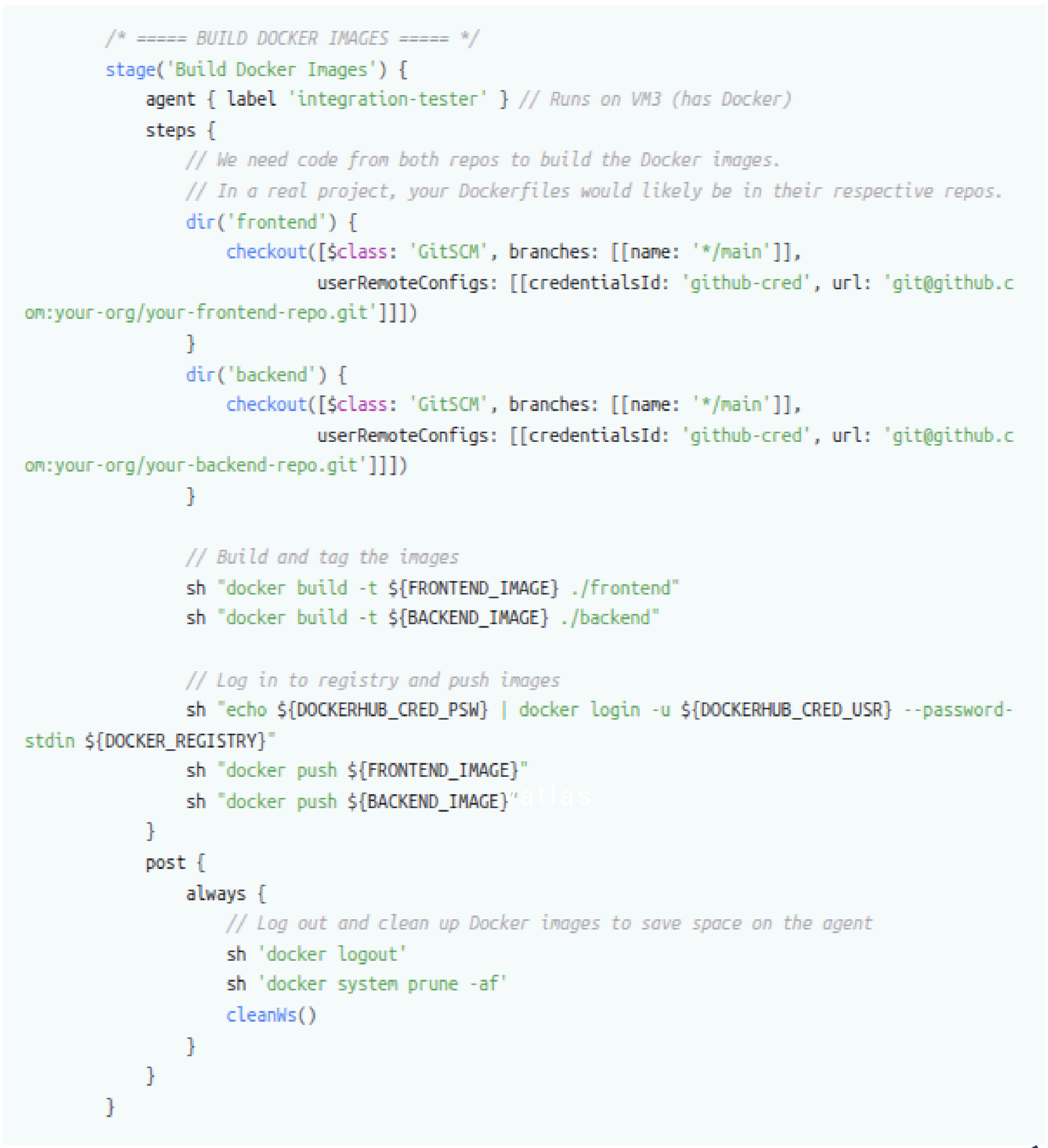


```
    // 4. THE MAIN SEQUENCE OF STAGES
    stages {
        /* ===== PARALLEL BUILD STAGE ===== */
        stage('Build and Package Applications') {
            // This stage itself doesn't run on an agent, it just runs the parallel steps.
            parallel {
                /* --- Frontend Build --- */
                stage('Build Frontend') {
                    agent { label 'frontend-builder' } // Runs on VM1
                    tools { nodejs 'nodejs-18' } // Auto-installs Node.js on the agent
                    environment {
                        // Example of a build-time environment variable for Next.js
                        NEXT_PUBLIC_API_URL = 'https://api.myapp.com'
                    }
                    steps {
                        checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                                  userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@githu
b.com:your-org/your-frontend-repo.git']]])
                        sh 'npm ci' # ci is faster and more strict than install
                        sh 'npm run build'
                        sh 'npm run export' # If you are doing a static export

                        // Archive the build results so they can be downloaded later
                        archiveArtifacts artifacts: 'out/**/*', allowEmptyArchive: true // For Nex
t.js 'out' dir
                    }
                    post {
                        always {
                            // Clean up the workspace on the agent to save disk space
                            cleanWs()
                        }
                    }
                }
```



```
                /* --- Backend Build --- */
                stage('Build Backend') {
                    agent { label 'backend-builder' } // Runs on VM2
                    tools { maven 'maven-3.8' }
                    steps {
                        checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                                  userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@githu
b.com:your-org/your-backend-repo.git']]])
                        sh 'mvn clean package -DskipTests' // Compile and package, skip unit tests
for speed

                        // Archive the JAR file so it can be downloaded or used later
                        archiveArtifacts artifacts: 'target/*.jar', allowEmptyArchive: true
                    }
                    post {
                        always {
                            cleanWs()
                        }
                    }
                }
            }
        }
```

docker build on vm3 on the same level as stage "build and package application"

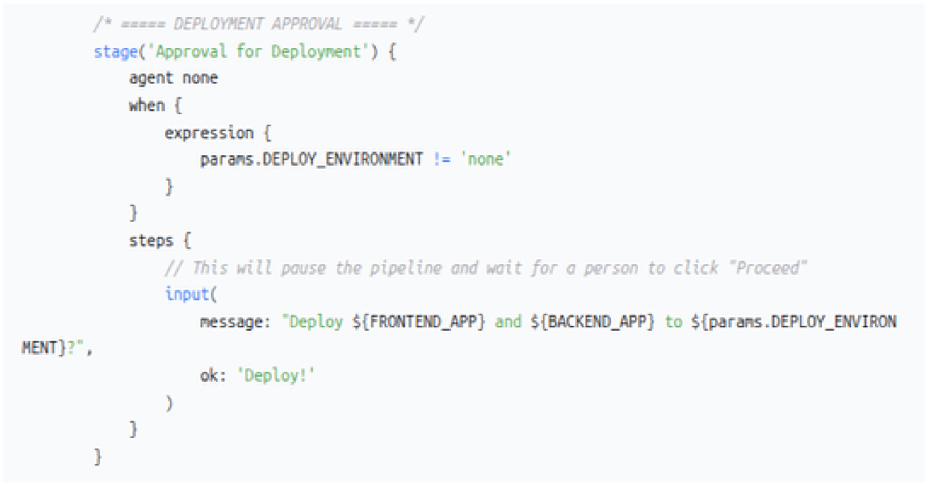


```
        /* ===== BUILD DOCKER IMAGES ===== */
        stage('Build Docker Images') {
            agent { label 'integration-tester' } // Runs on VM3 (has Docker)
            steps {
                // We need code from both repos to build the Docker images.
                // In a real project, your Dockerfiles would likely be in their respective repos.
                dir('frontend') {
                    checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                             userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@github.c
om:your-org/your-frontend-repo.git']]])
                }
                dir('backend') {
                    checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                             userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@github.c
om:your-org/your-backend-repo.git']]])
                }

                // Build and tag the images
                sh "docker build -t ${FRONTEND_IMAGE} ./frontend"
                sh "docker build -t ${BACKEND_IMAGE} ./backend"

                // Log in to registry and push images
                sh "echo ${DOCKERHUB_CRED_PSW} | docker login -u ${DOCKERHUB_CRED_USR} --password-
stdin ${DOCKER_REGISTRY}"
                sh "docker push ${FRONTEND_IMAGE}"
                sh "docker push ${BACKEND_IMAGE}"
            }
            post {
                always {
                    // Log out and clean up Docker images to save space on the agent
                    sh 'docker logout'
                    sh 'docker system prune -af'
                    cleanWs()
                }
            }
        }
```

INTEGRATION TESTS runs on vm3

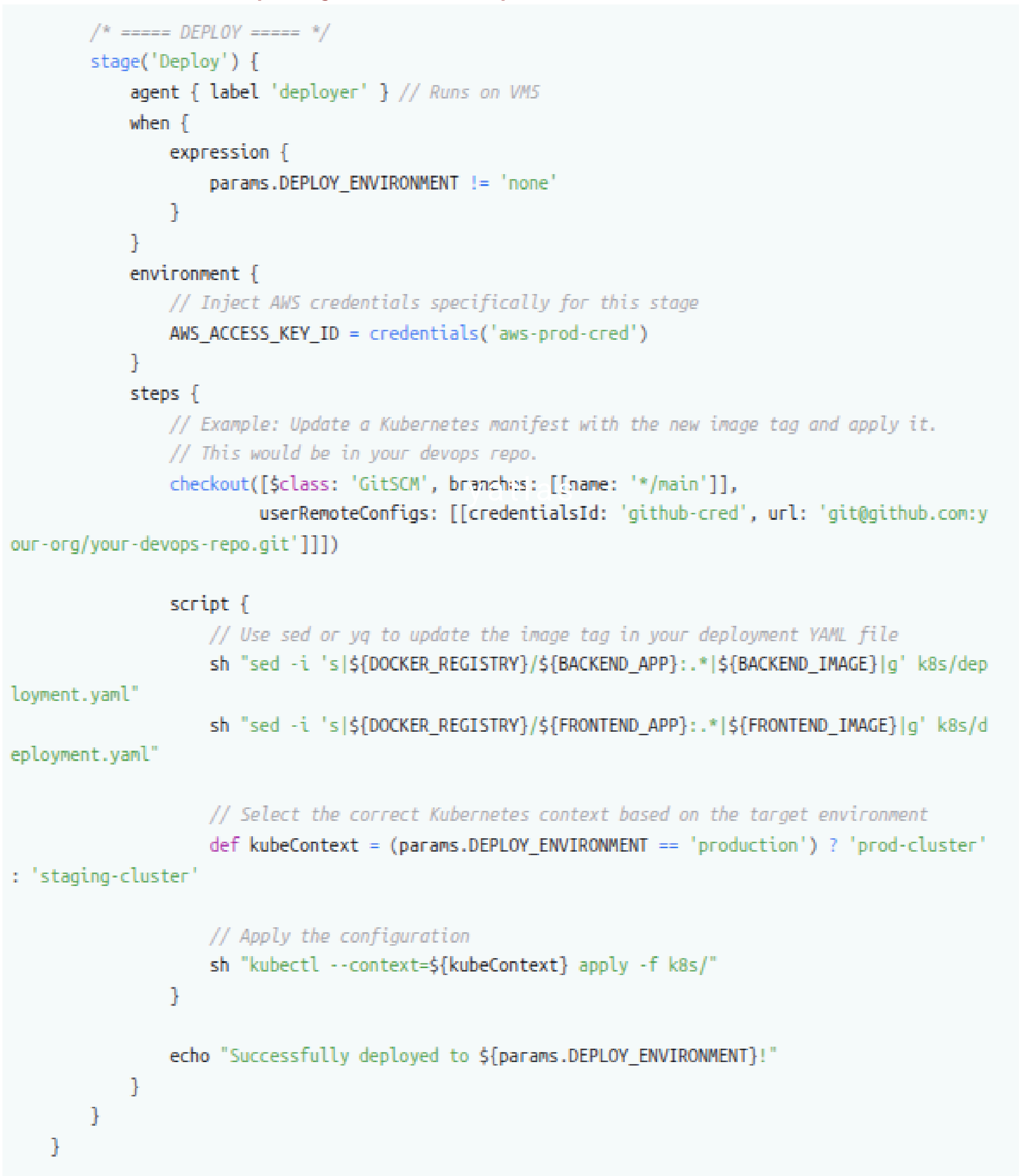
```
        /* ===== INTEGRATION TESTS ===== */
        stage('Integration Tests') {
            agent { label 'integration-tester' } // Runs on VM3
            when {
                expression { params.RUN_INTEGRATION_TESTS == true }
            }
            steps {
                // Checkout a repo that might have docker-compose and test scripts
                checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                         userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@github.com:y
our-org/your-devops-repo.git']]])

                // Use docker-compose to spin up Postgres, the backend, and maybe the frontend for
integration tests.
                sh "docker-compose -f docker-compose.test.yml up --build -d"
                sh 'sleep 30' // Wait for services to be healthy - better to use a healthcheck scr
ipt!

                // Run the actual tests (e.g., a script that calls the backend API)
                sh './run-integration-tests.sh'

                // Bring everything down
                sh "docker-compose -f docker-compose.test.yml down"
            }
            post {
                always {
                    // Ensure cleanup even if tests fail
                    sh "docker-compose -f docker-compose.test.yml down || true"
                    sh 'docker system prune -af'
                    cleanWs()
                }
            }
        }
```

deployement approval waiting for the human to click proceed

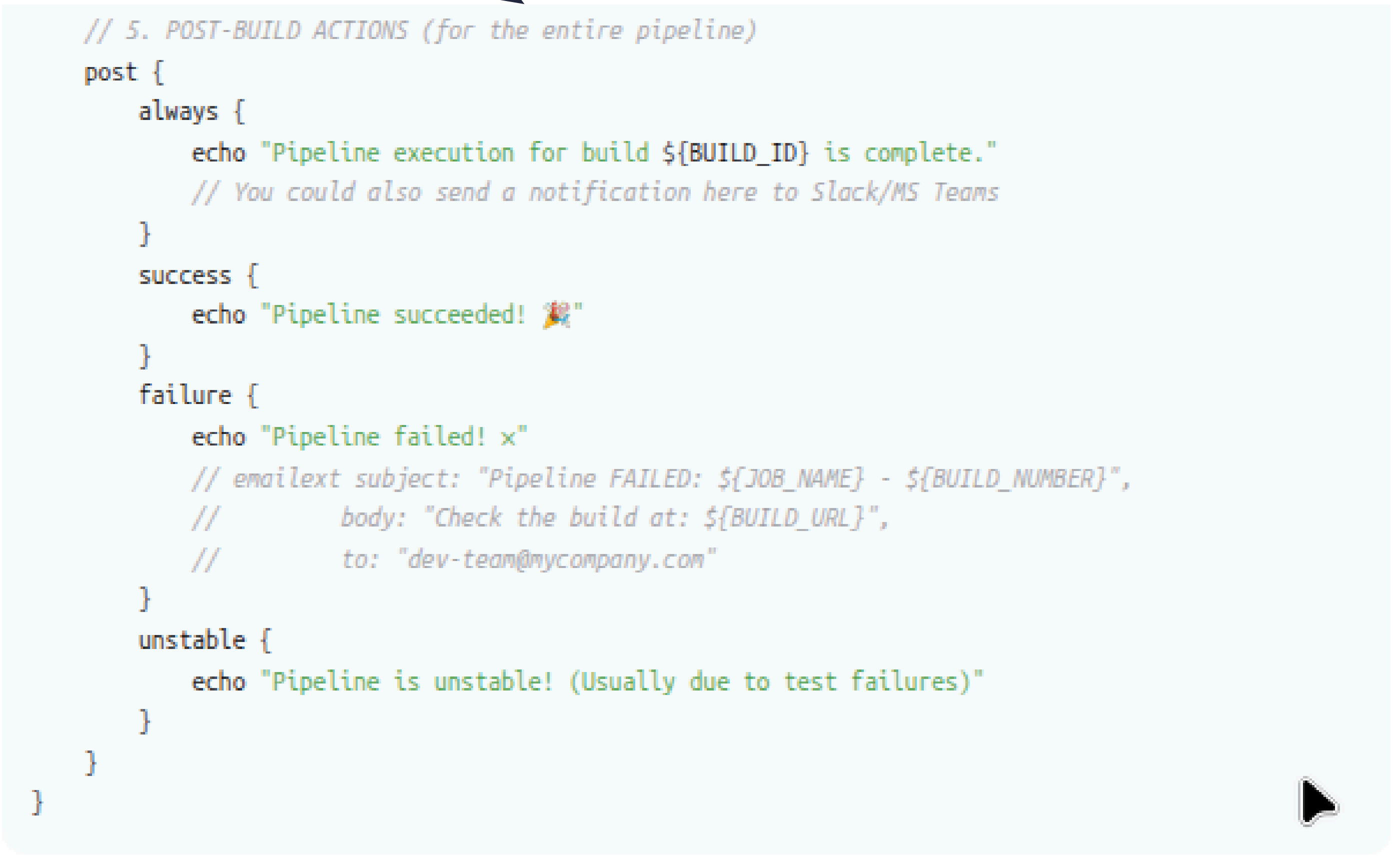


```
        /* ===== DEPLOYMENT APPROVAL ===== */
        stage('Approval for Deployment') {
            agent none
            when {
                expression {
                    params.DEPLOY_ENVIRONMENT != 'none'
                }
            }
            steps {
                // This will pause the pipeline and wait for a person to click "Proceed"
                input(
                    message: "Deploy ${FRONTEND_APP} and ${BACKEND_APP} to ${params.DEPLOY_ENVIRON
MENT}?",
                    ok: 'Deploy!'
                )
            }
        }
```

deployement process on vm5



```
        /* ===== DEPLOY ===== */
        stage('Deploy') {
            agent { label 'deployer' } // Runs on VM5
            when {
                expression {
                    params.DEPLOY_ENVIRONMENT != 'none'
                }
            }
            environment {
                // Inject AWS credentials specifically for this stage
                AWS_ACCESS_KEY_ID = credentials('aws-prod-cred')
            }
            steps {
                // Example: Update a Kubernetes manifest with the new image tag and apply it.
                // This would be in your devops repo.
                checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                          userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@github.com:y
our-org/your-devops-repo.git']]])

                script {
                    // Use sed or yq to update the image tag in your deployment YAML file
                    sh "sed -i 's|${DOCKER_REGISTRY}/${BACKEND_APP}:.*|${BACKEND_IMAGE}|g' k8s/dep
loyment.yaml"
                    sh "sed -i 's|${DOCKER_REGISTRY}/${FRONTEND_APP}:.*|${FRONTEND_IMAGE}|g' k8s/d
eployment.yaml"

                    // Select the correct Kubernetes context based on the target environment
                    def kubeContext = (params.DEPLOY_ENVIRONMENT == 'production') ? 'prod-cluster'
: 'staging-cluster'

                    // Apply the configuration
                    sh "kubectl --context=${kubeContext} apply -f k8s/"
                }

                echo "Successfully deployed to ${params.DEPLOY_ENVIRONMENT}!"
            }
        }
    }
```



```
    // 5. POST-BUILD ACTIONS (for the entire pipeline)
    post {
        always {
            echo "Pipeline execution for build ${BUILD_ID} is complete."
            // You could also send a notification here to Slack/MS Teams
        }
        success {
            echo "Pipeline succeeded! 🎉"
        }
        failure {
            echo "Pipeline failed! x"
            // emailext subject: "Pipeline FAILED: ${JOB_NAME} - ${BUILD_NUMBER}",
            //          body: "Check the build at: ${BUILD_URL}",
            //          to: "dev-team@mycompany.com"
        }
        unstable {
            echo "Pipeline is unstable! (Usually due to test failures)"
        }
    }
}
```

what about Canary Deployment or Regional Rollout strategy?

The goal is to deploy the new version to a small, specific region first, validate it (monitor for errors, performance, etc.), and only then proceed to deploy it to all other regions (global rollout).
This requires precise control and conditionality in your pipeline. Here's how you would modify the Deploy stage to achieve this.

Approach:

We will break the Deploy stage into two new stages:

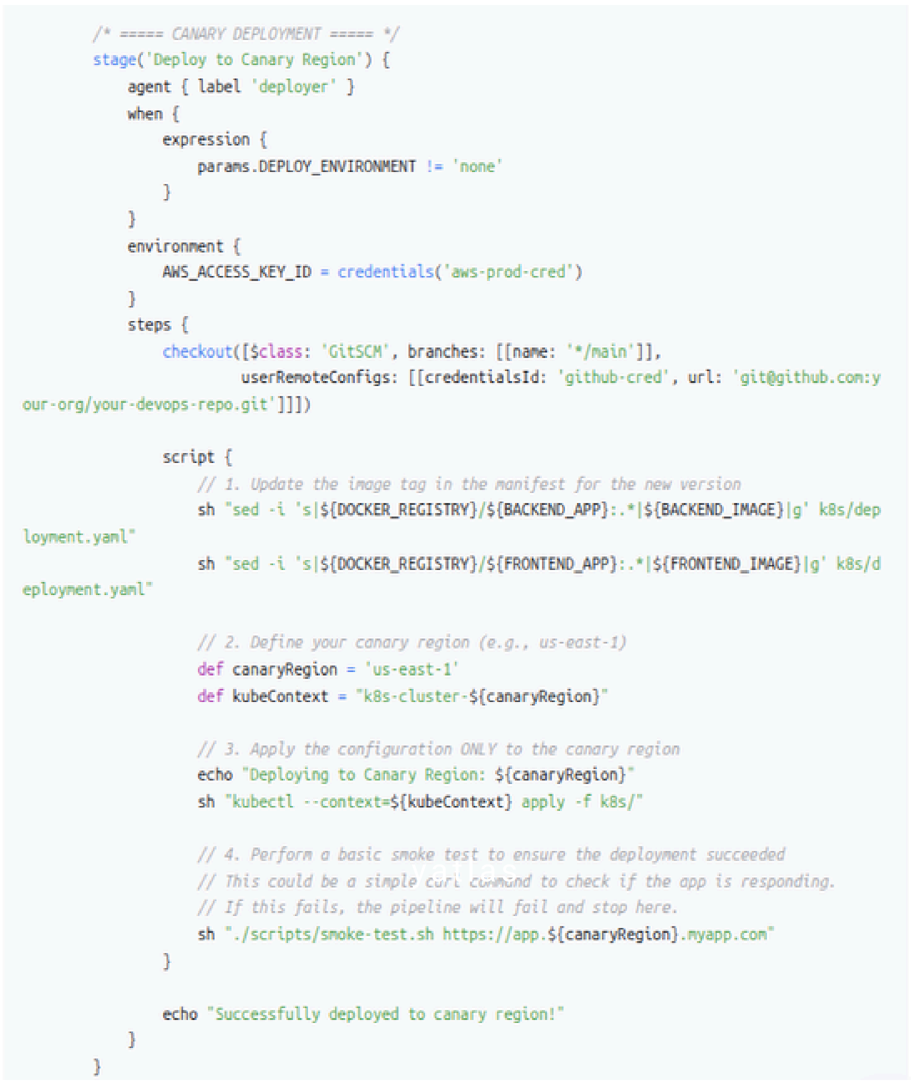
- Deploy to Canary Region
- Deploy to Global (which will only run after the canary is validated).

We'll use the input step again as a manual validation gate after the canary deployment. For full automation, you would integrate with monitoring tools (like Prometheus/Datadog) to automatically validate based on metrics.

Modified Jenkinsfile Snippet (Replacing the Deploy Stage)

This code replaces the single Deploy stage from the previous example.

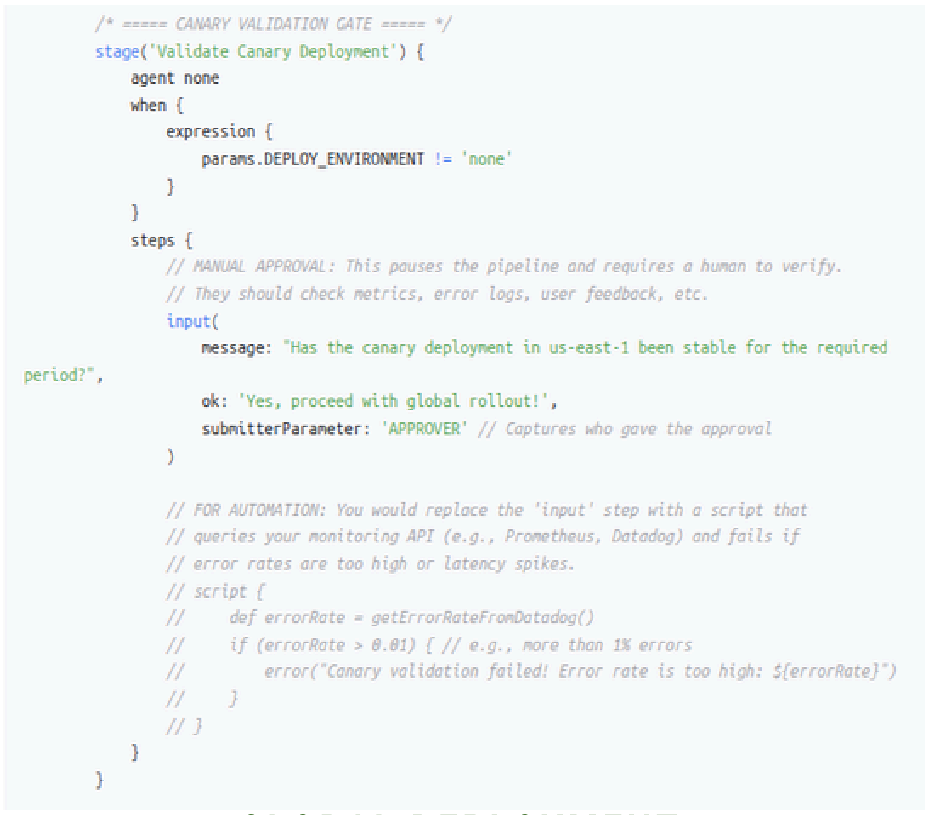
CANARY DEPLOYEMENT PHASE:



```
        /* ===== CANARY DEPLOYMENT ===== */
        stage('Deploy to Canary Region') {
            agent { label 'deployer' }
            when {
                expression {
                    params.DEPLOY_ENVIRONMENT != 'none'
                }
            }
            environment {
                AWS_ACCESS_KEY_ID = credentials('aws-prod-cred')
            }
            steps {
                checkout([$class: 'GitSCM', branches: [[name: '*/main']],
                          userRemoteConfigs: [[credentialsId: 'github-cred', url: 'git@github.com:y
our-org/your-devops-repo.git']]])

                script {
                    // 1. Update the image tag in the manifest for the new version
                    sh "sed -i 's|${DOCKER_REGISTRY}/${BACKEND_APP}:.*|${BACKEND_IMAGE}|g' k8s/dep
loyment.yaml"
                    sh "sed -i 's|${DOCKER_REGISTRY}/${FRONTEND_APP}:.*|${FRONTEND_IMAGE}|g' k8s/d
eployment.yaml"

                    // 2. Define your canary region (e.g., us-east-1)
                    def canaryRegion = 'us-east-1'
                    def kubeContext = "k8s-cluster-${canaryRegion}"

                    // 3. Apply the configuration ONLY to the canary region
                    echo "Deploying to Canary Region: ${canaryRegion}"
                    sh "kubectl --context=${kubeContext} apply -f k8s/"

                    // 4. Perform a basic smoke test to ensure the deployment succeeded
                    // This could be a simple curl command to check if the app is responding.
                    // If this fails, the pipeline will fail and stop here.
                    sh "./scripts/smoke-test.sh https://app.${canaryRegion}.myapp.com"
                }

                echo "Successfully deployed to canary region!"
            }
        }
```

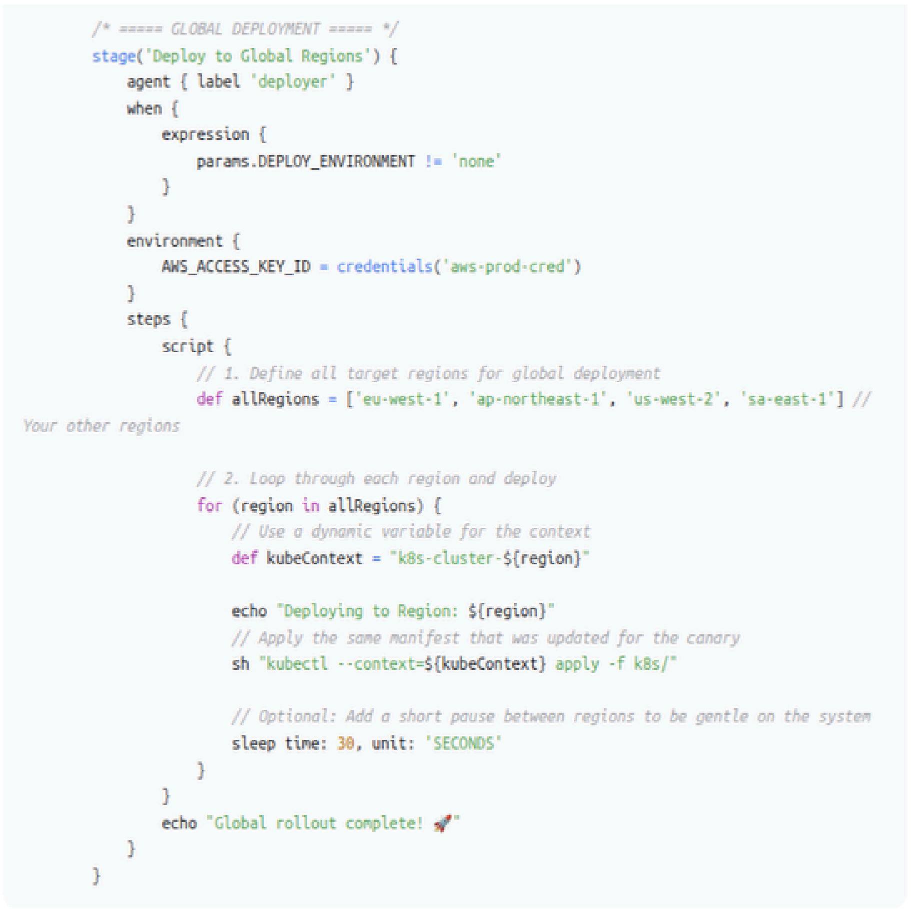
yatlas

CANARY VALIDATION GATE:



```
        /* ===== CANARY VALIDATION GATE ===== */
        stage('Validate Canary Deployment') {
            agent none
            when {
                expression {
                    params.DEPLOY_ENVIRONMENT != 'none'
                }
            }
            steps {
                // MANUAL APPROVAL: This pauses the pipeline and requires a human to verify.
                // They should check metrics, error logs, user feedback, etc.
                input(
                    message: "Has the canary deployment in us-east-1 been stable for the required
period?",
                    ok: 'Yes, proceed with global rollout!',
                    submitterParameter: 'APPROVER' // Captures who gave the approval
                )

                // FOR AUTOMATION: You would replace the 'input' step with a script that
                // queries your monitoring API (e.g., Prometheus, Datadog) and fails if
                // error rates are too high or latency spikes.
                // script {
                //     def errorRate = getErrorRateFromDatadog()
                //     if (errorRate > 0.01) { // e.g., more than 1% errors
                //         error("Canary validation failed! Error rate is too high: ${errorRate}")
                //     }
                // }
            }
        }
```

GLOBAL DEPLOYMENT:



```
        /* ===== GLOBAL DEPLOYMENT ===== */
        stage('Deploy to Global Regions') {
            agent { label 'deployer' }
            when {
                expression {
                    params.DEPLOY_ENVIRONMENT != 'none'
                }
            }
            environment {
                AWS_ACCESS_KEY_ID = credentials('aws-prod-cred')
            }
            steps {
                script {
                    // 1. Define all target regions for global deployment
                    def allRegions = ['eu-west-1', 'ap-northeast-1', 'us-west-2', 'sa-east-1'] //
Your other regions

                    // 2. Loop through each region and deploy
                    for (region in allRegions) {
                        // Use a dynamic variable for the context
                        def kubeContext = "k8s-cluster-${region}"

                        echo "Deploying to Region: ${region}"
                        // Apply the same manifest that was updated for the canary
                        sh "kubectl --context=${kubeContext} apply -f k8s/"

                        // Optional: Add a short pause between regions to be gentle on the system
                        sleep time: 30, unit: 'SECONDS'
                    }
                }
                echo "Global rollout complete! 🚀"
            }
        }
```

YOU DID WELL