DEVOPS

CI/CD

INTRODUCTION TO CI/CD

- Continuous Integration (CI) and Continuous Deployment/Delivery (CD) are practices used in modern software development to improve code quality, reduce integration issues, and accelerate the delivery of software.

BENEFITS OF CI/CD

Faster Delivery of Features:

- Automating the process of integrating and deploying code allows for faster and more frequent releases.

Improved Code Quality:

- Automated testing and integration help catch bugs and issues early, leading to higher-quality code.

Reduced Integration Issues:

- Continuous integration ensures that code changes are frequently merged and tested, reducing the likelihood of integration problems.

Enhanced Collaboration:

- CI/CD practices promote collaboration among team members by providing a unified and automated workflow.

Early Detection of Bugs:

- Automated tests run as part of the CI/CD pipeline catch bugs early in the development cycle, making them easier and cheaper to fix.

Consistent Deployment Process:

- Automated deployments ensure that the deployment process is consistent and repeatable, reducing human error.

KEY CONCEPTS

Key Concepts

Build

Test

Deploy

BUILD

- **Definition**:
  - The build process involves compiling the source code, resolving dependencies, and packaging the software into a deployable format (e.g., executable, container).
- **Automation**:
  - Tools like Maven, Gradle, or npm are often used to automate the build process.
- **Example**:
  - In a Java project, the build process might involve compiling .java files into .class files and then packaging them into a .jar or .war file.

TEST

- Automated tests are run to validate the functionality, performance, and security of the code.
- **Types of Tests**:
  - **Unit Tests**: Test individual components or functions.
  - **Integration Tests**: Test the interaction between different components.
  - **End-to-End Tests**: Test the application from the user's perspective.
- **Tools**:
  - Common testing frameworks include JUnit, TestNG, Selenium, and Jest.
- **Example**:
  - Running JUnit tests as part of the build process to ensure that all methods in a Java class work as expected.

DEPLOY

- **Definition**:
  - Deployment is the process of releasing the built and tested code to a production or staging environment.
- **Automation**:
  - Deployment pipelines automate the steps required to deploy the application, including environment setup, database migrations, and service restarts.
- **Tools**:
  - Popular deployment tools and platforms include Jenkins, GitLab CI, CircleCI, and AWS CodePipeline.
- **Example**:
  - Using Jenkins to automate the deployment of a web application to a production server after passing all tests.

CI/CD PIPELINE STEPS

- **Code Commit**: Developers commit code changes to a version control system (e.g., Git).
- **Build**: The CI server automatically triggers a build process.
- **Test**: Automated tests are run on the built code.
- **Deploy**: If tests pass, the code is automatically deployed to a staging or production environment.
- **Feedback**: Results of the build, test, and deploy processes are reported back to the development team for further action.

TOOLS FOR CI/CD

Jenkins:

- An open-source automation server used for building, testing, and deploying code.

GitLab CI:

- A built-in CI/CD tool in GitLab that allows for seamless integration with GitLab repositories.

CircleCI:

- A cloud-based CI/CD tool that automates the build, test, and deploy processes.

Travis CI:

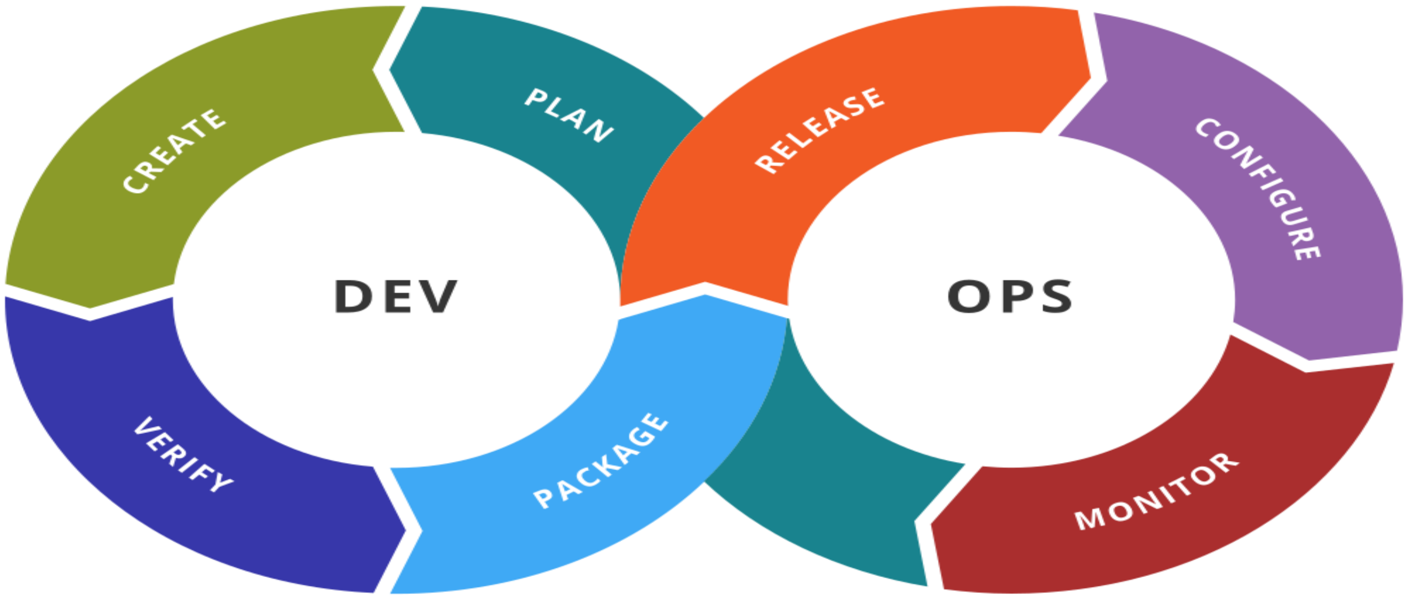
- A CI service used to build and test projects hosted on GitHub.

Azure DevOps:

- A suite of development tools provided by Microsoft, which includes CI/CD capabilities.

DEVOPS

- DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) with the aim of shortening the software development lifecycle and providing continuous delivery with high software quality.



KEY CONCEPTS AND BENEFITS:

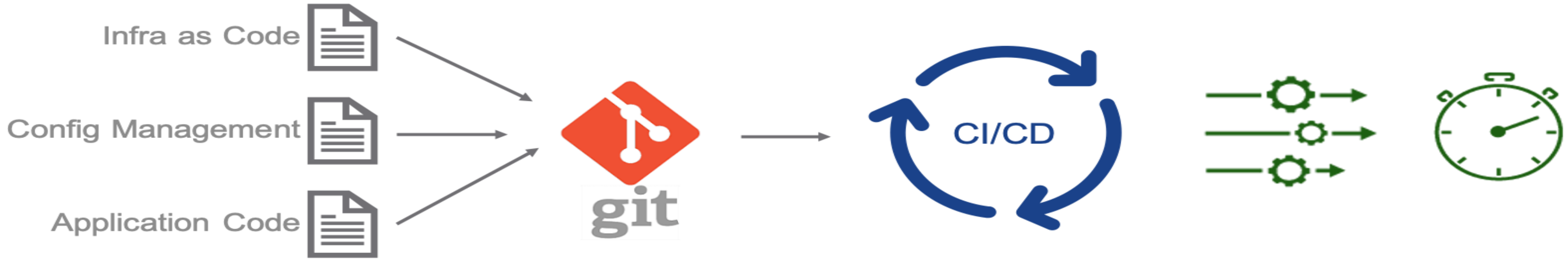
- **Collaboration and Communication**: Encourages collaboration between development and operations teams to improve productivity and reduce silos.
- **Automation**: Emphasizes the automation of repetitive tasks, such as testing, integration, and deployment, to increase efficiency and reduce errors.
- **CI/CD**: Ensures code changes are automatically built, tested, and deployed to production, allowing for faster and more reliable releases.
- **Monitoring and Logging**: Continuously monitors the performance and logs of applications to identify and resolve issues promptly.
- **Infrastructure as Code (IaC)**: Manages and provisions computing infrastructure through machine-readable scripts, enhancing consistency and scalability.

TOOLS

- **Jenkins**, **Travis CI**, **CircleCI**: CI/CD tools for automating the build and deployment processes.
- **Docker**, **Kubernetes**: Containerization and orchestration tools for managing application deployments.
- **Ansible**, **Terraform**: IaC tools for provisioning and managing infrastructure.
- **Prometheus**, **Grafana**: Monitoring and alerting tools.

GITOPS

GitOps-in-a-nutshell



Infra as Code

Config Management

Application Code

git

CI/CD

“Source of Truth” for declarative code

Update to code source triggers a pipeline

Pipeline runs a series of tasks, resulting in the update of the runtime environment to match the source

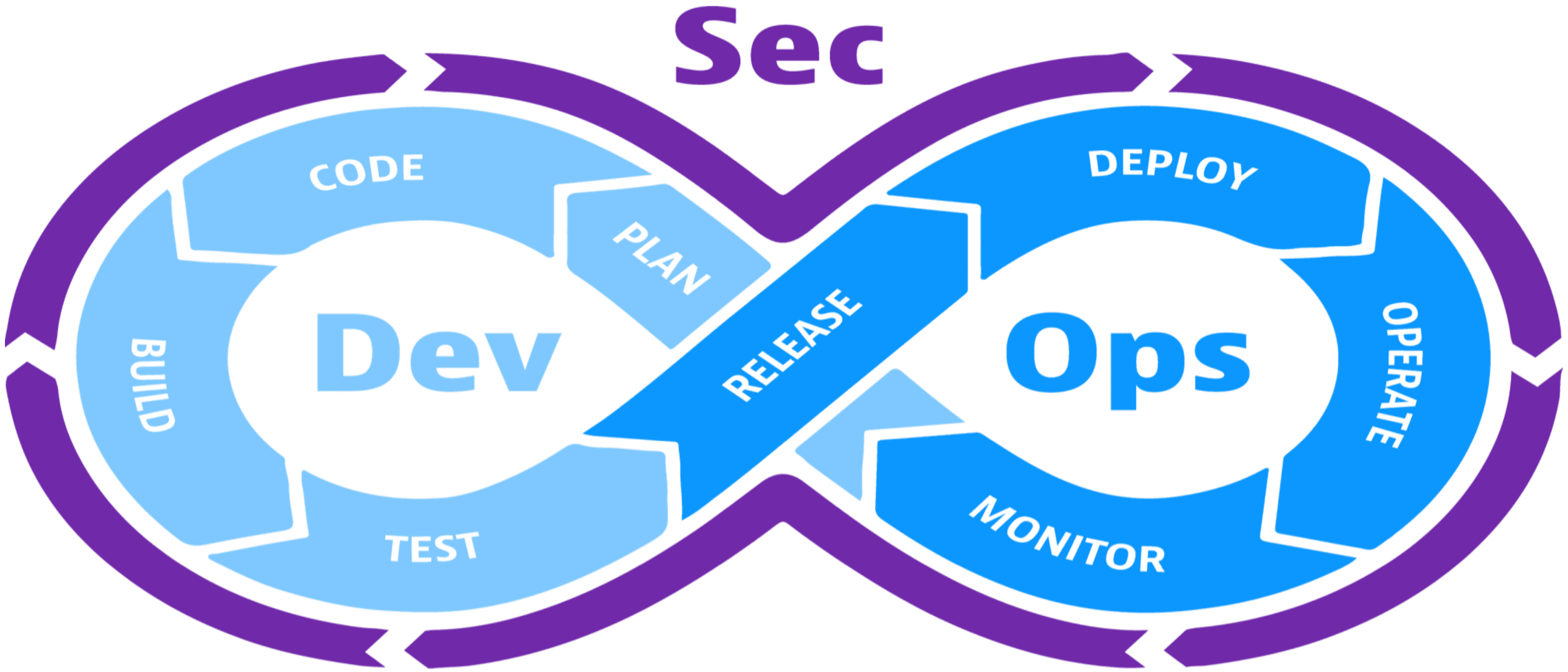
KEY CONCEPTS AND BENEFITS

- **Declarative Configuration**: Defines the desired state of the infrastructure and applications in Git, making it easier to manage and track changes.
- **Version Control**: Uses Git to version control configurations and code, providing a clear audit trail of changes.
- **Automated Deployments**: Deploys changes automatically when updates are pushed to the Git repository, ensuring consistency between the Git state and the deployed state.
- **Rollback and Recovery**: Facilitates easy rollback to previous states in case of issues, improving reliability and stability.
- **Enhanced Security**: Ensures that all changes go through the Git repository, making unauthorized changes less likely.

TOOLS

- **Flux**, **Argo CD**:
  - GitOps tools for automating the deployment of applications using Git as the source of truth.
- **Kustomize**, **Helm**:
  - Tools for managing Kubernetes configurations declaratively.
- **GitHub**, **GitLab**:
  - Version control systems for managing code and configuration.

DEVSECOPS


Sec
CODE
PLAN
BUILD
Dev
TEST
RELEASE
DEPLOY
Ops
OPERATE
MONITOR

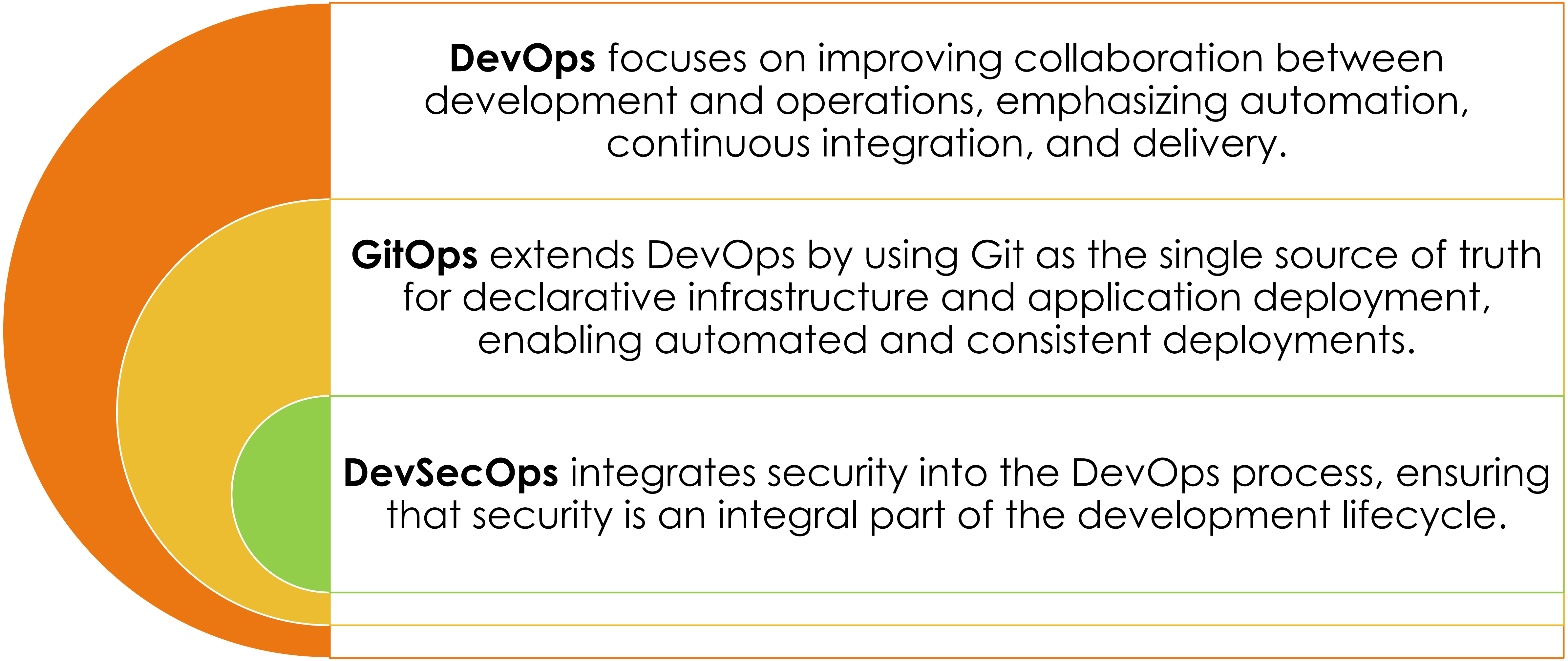
KEY CONCEPTS AND BENEFITS

- **Shift Left Security**: Incorporates security practices early in the development process to identify and fix vulnerabilities sooner.
- **Automation**: Automates security testing and compliance checks as part of the CI/CD pipeline to reduce manual efforts and ensure consistent security standards.
- **Collaboration**: Promotes collaboration between development, security, and operations teams to address security concerns proactively.
- **Continuous Monitoring**: Continuously monitors applications and infrastructure for security threats and vulnerabilities.
- **Compliance and Governance**: Ensures that applications comply with regulatory requirements and organizational policies.

TOOLS

- **SonarQube**, **Checkmarx**: Static code analysis tools for identifying security vulnerabilities in the code.
- **Aqua Security**, **Twistlock**: Container security tools for scanning and protecting containerized applications.
- **OWASP ZAP**, **Burp Suite**: Tools for performing dynamic application security testing (DAST).
- **Vault**, **CyberArk**: Secrets management tools for securely storing and managing credentials and sensitive information.

COMPARISON



DevOps focuses on improving collaboration between development and operations, emphasizing automation, continuous integration, and delivery.

GitOps extends DevOps by using Git as the single source of truth for declarative infrastructure and application deployment, enabling automated and consistent deployments.

DevSecOps integrates security into the DevOps process, ensuring that security is an integral part of the development lifecycle.

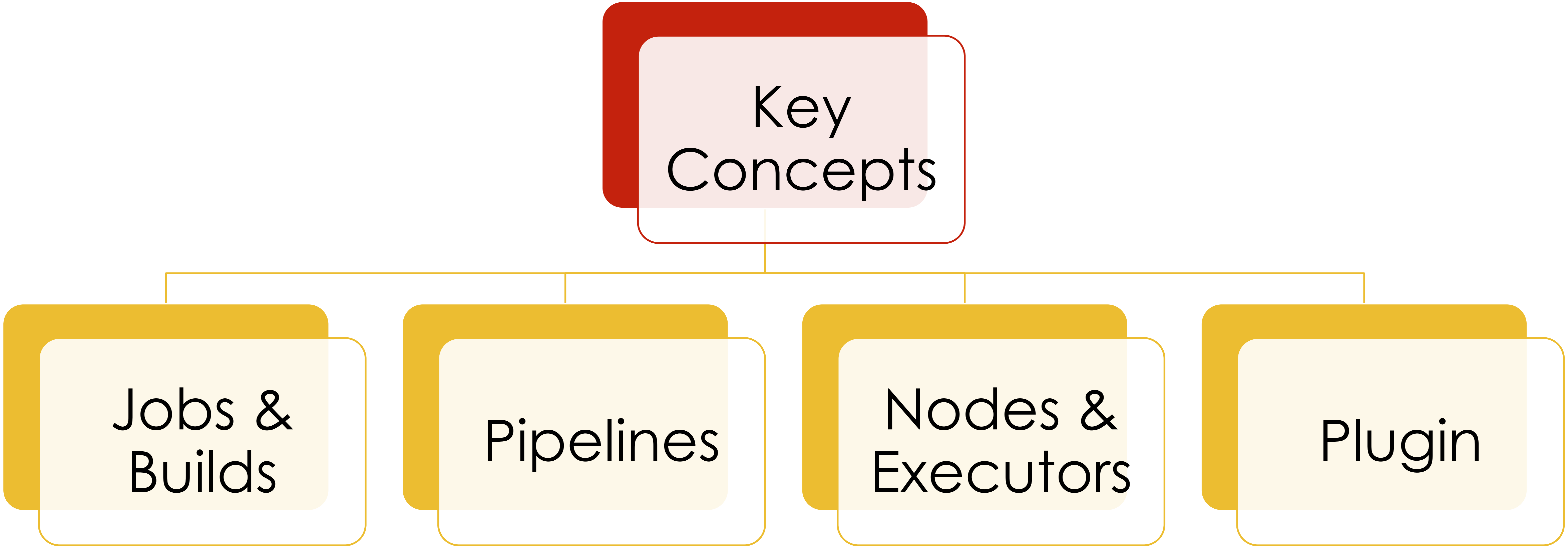
INTRODUCTION TO JENKINS

- **Jenkins** is an open-source automation server used to automate various aspects of the software development process, including building, testing, and deploying applications.
- It is one of the most popular tools in the DevOps toolkit and is widely used for continuous integration and continuous delivery (CI/CD).

KEY FEATURES

- **Extensible**: Jenkins has a rich ecosystem of plugins that allow it to integrate with many other tools and platforms, making it highly customizable.
- **Distributed Builds**: Jenkins can distribute builds and tests across multiple machines to improve performance and reliability.
- **Easy Configuration**: Configuration can be done through a web-based interface, making it accessible to users without a deep understanding of the command line.
- **Pipeline as Code**: Jenkins supports writing build pipelines as code using the Groovy-based Domain Specific Language (DSL), allowing version control and more complex workflows.
- **Community Support**: As an open-source tool, Jenkins has a large community that contributes plugins, documentation, and support.

KEY CONCEPTS



Key Concepts

Jobs & Builds

Pipelines

Nodes & Executors

Plugin

- **Jobs**:
  - In Jenkins, a job (or project) is a task or set of tasks that Jenkins executes.
  - Jobs can be configured to perform various tasks such as building code, running tests, or deploying applications.
- **Builds**:
  - Each execution of a job is called a build.
  - Builds can be triggered manually, scheduled, or triggered automatically by events such as code commits.
- **Pipeline**:
  - A pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines. It defines the entire build lifecycle of an application.
  - **Declarative Pipeline**: A simpler and more structured way to define pipelines using a specific syntax.
  - **Scripted Pipeline**: A more flexible way to define pipelines using a general-purpose Groovy script.

- **Nodes**:
  - Machines where Jenkins runs jobs. The main Jenkins server is called the "master," and additional machines are called "agents" or "slaves."
- **Executors**:
  - The number of concurrent jobs a node can run.
- **Plugins**:
  - Jenkins' functionality can be extended through plugins, which integrate with other tools and add new features.
  - Examples include Git, Docker, Maven, and Slack integrations.

INSTALL JENKINS

- **Download Jenkins**: Jenkins can be downloaded from official website
- **Install Jenkins**: Follow the installation instructions for your operating system. Jenkins can be installed as a standalone application or deployed in a servlet container such as Apache Tomcat.
- **Start Jenkins**: Once installed, start Jenkins and access it through a web browser at http://localhost:8080 by default.
- **Initial Setup**: Complete the initial setup by installing recommended plugins and creating an admin user.

ACTIVITY 1

- Let’s Create 1 AWS Instance
- Install Jenkins
- Install JDK
- Start Jenkins Service
- Check status

- Download Jenkins.war file
- Execute below command
- Java –jar jenkins.war
- It will start jenkins service on 8080 port
- Localhost:8080
- Install suggested plugins
- Start working with jenkins

ACTIVITY 2

- Create Free Style Project
- In configuration:
  - Discard old build: select
- If git needed you can select otherwise keep none
- Select any **Build Triggers** and **Build Environment** as per your task requirements
- Now, configure your build by clicking on **Add build step** and selecting **Execute Shell**
- Add the list of shell commands that you want the **Build** to execute automatically
  - Java -version
- Add additional **Build** steps by clicking **Add build step > Execute Shell**. These build steps can be different, as they are separate jobs that Jenkins executes sequentially.
  - Echo "thank you for using Jenkins"
- Save
- Click on Build now
- Check output in console

ACTIVITY: BUILD MAVEN APP IN JENKINS

- Create free style project
- In git add maven project repo link
- Go to the Build tab and select Invoke top-level Maven targets
    - Add clean
- One more build with target: package
- Build now check console

MAVEN PLUGIN INTEGRATION

- Configuring plugin by clicking on manage plugins
- Click on the AVAILABLE button to access the available plugins
- search for Maven Integration u
- After the plugins are installed successfully, select Restart Jenkins
- Now you can see maven project creation option on Jenkins Dashboard

CONFIGURE MAVEN VERSION

- Manage jenkins
- Global tools configuration
- Maven → Add maven
- Give the name → install automatically
- Click save

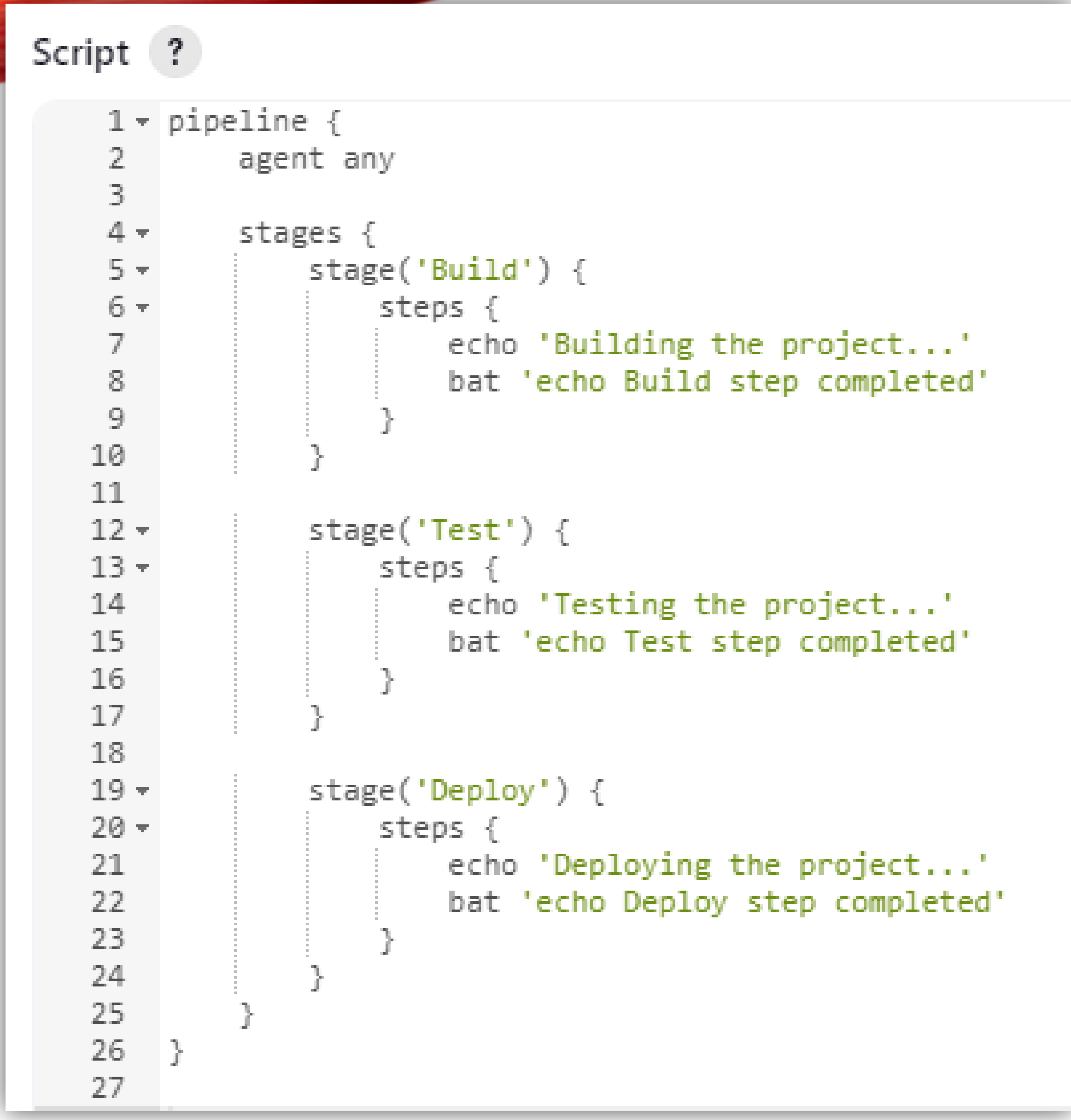
CREATE MAVEN PROJECT

- Select Maven project in JOB
- Add description
- Add git repo link with branch name
- In configure add goals and options: clean package
- Scroll and save
- Build now
- Check console

WORKING WITH JENKINS PIPELINE

- Click on newItem
- Select pipeline
- In pipeline select hello pipeline script
- Save
- Build now
- Check console

PIPELINE WITH STAGES

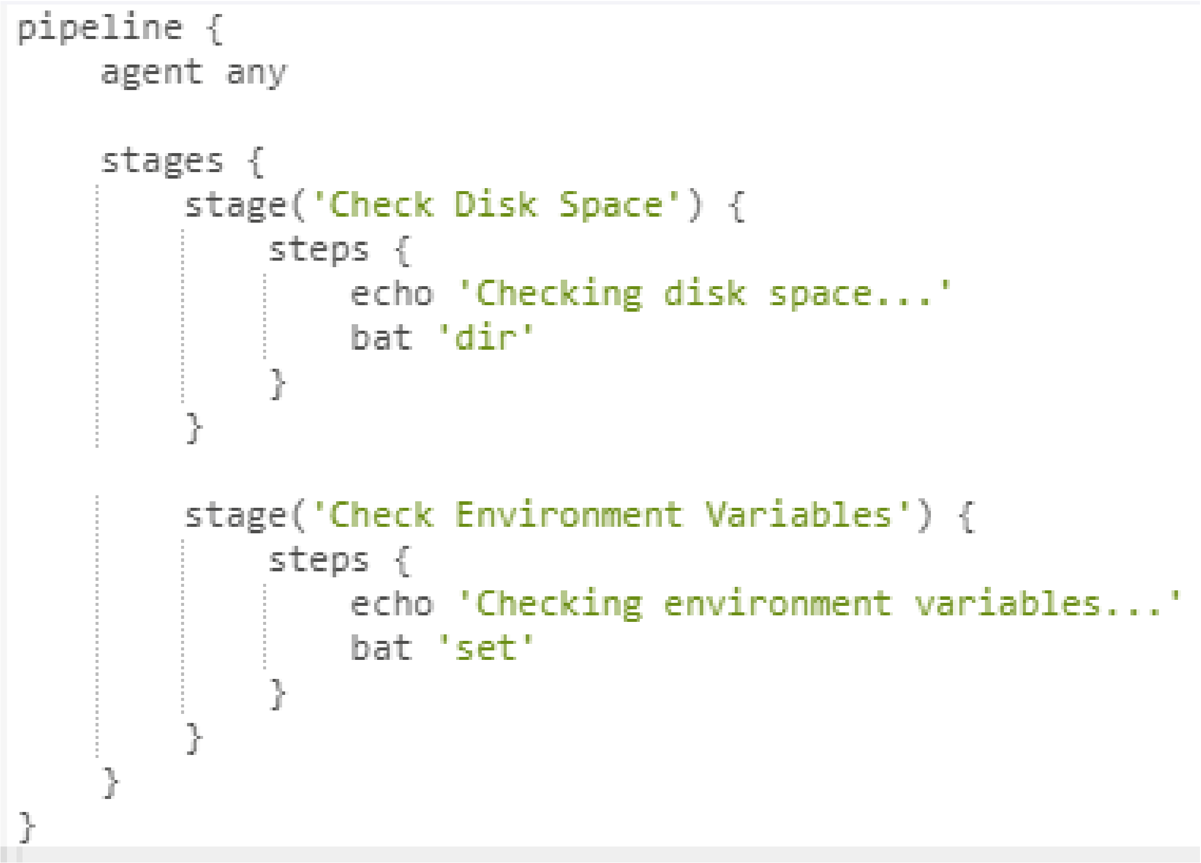


```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
                bat 'echo Build step completed'
            }
        }

        stage('Test') {
            steps {
                echo 'Testing the project...'
                bat 'echo Test step completed'
            }
        }

        stage('Deploy') {
            steps {
                echo 'Deploying the project...'
                bat 'echo Deploy step completed'
            }
        }
    }
}

```

- newItem → pipeline
- Select hello script
- Edit with other stages like build, test, deploy
- Save
- Build now
- Check console

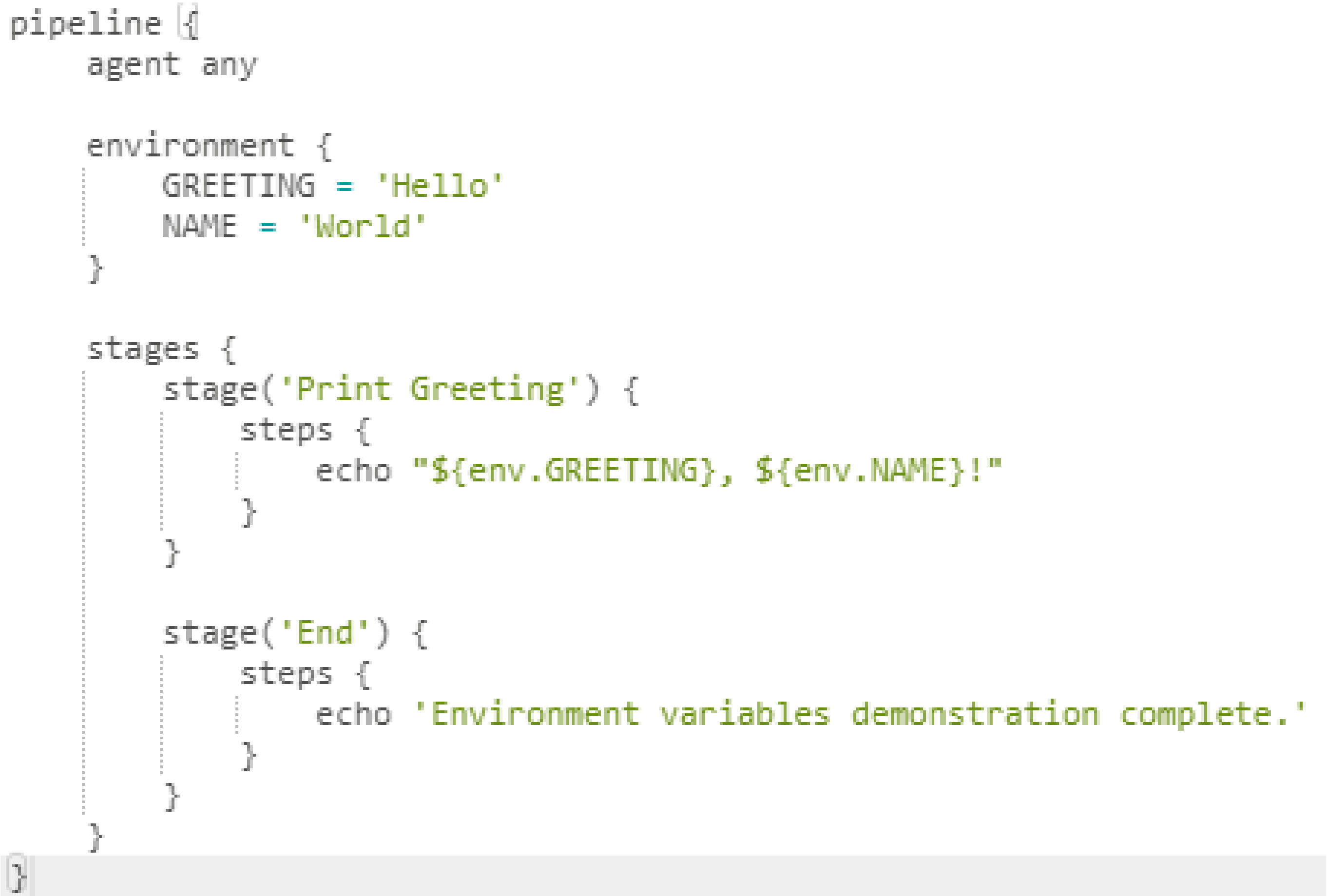
PIPELINE ACTIVITY 1

- Simple Shell Commands Pipeline



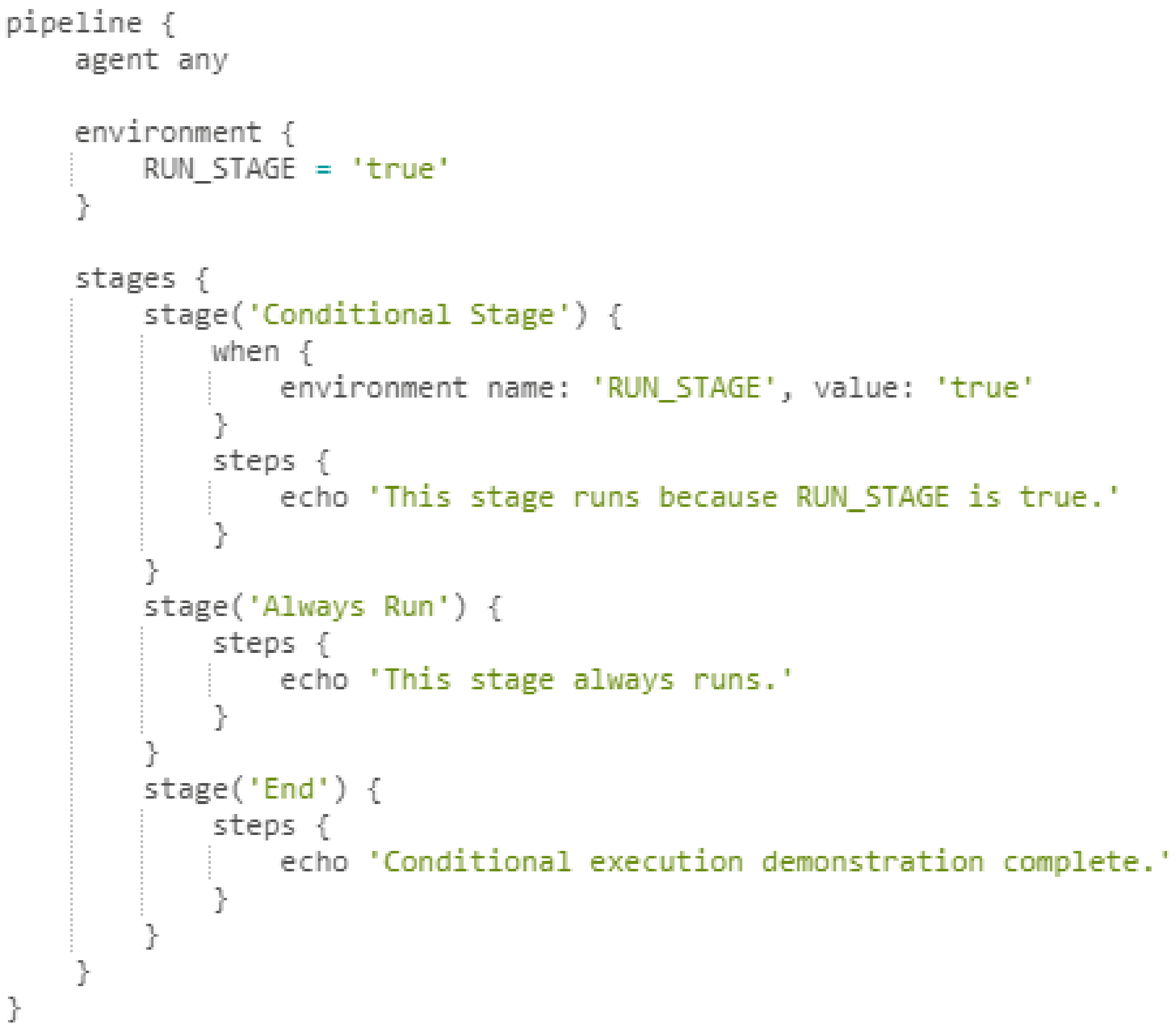
```
pipeline {
    agent any

    stages {
        stage('Check Disk Space') {
            steps {
                echo 'Checking disk space...'
                bat 'dir'
            }
        }

        stage('Check Environment Variables') {
            steps {
                echo 'Checking environment variables...'
                bat 'set'
            }
        }
    }
}
```

PIPELINE ACTIVITY 2
USING ENVIRONMENT VARIABLES



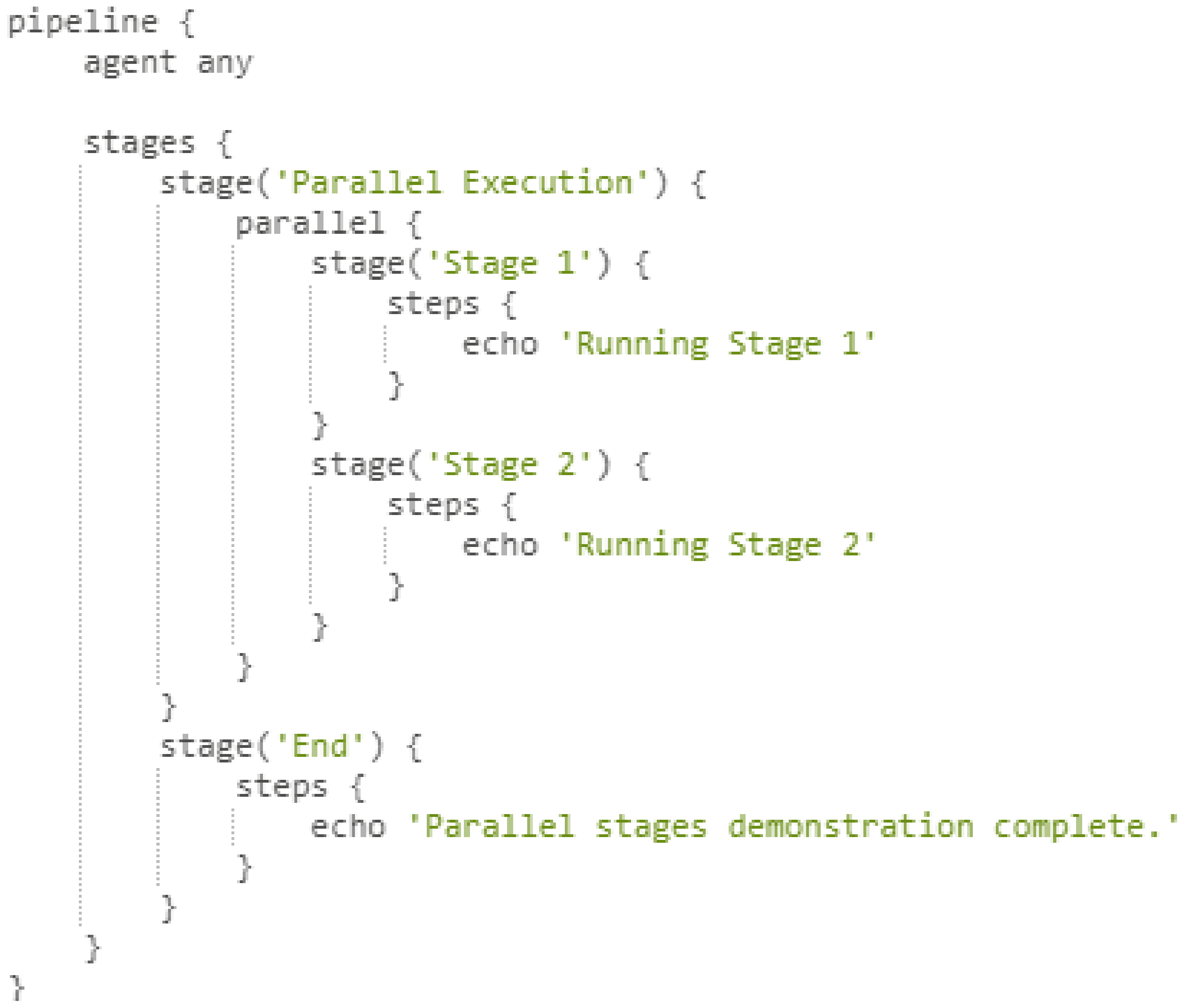
```
pipeline {
    agent any

    environment {
        GREETING = 'Hello'
        NAME = 'World'
    }

    stages {
        stage('Print Greeting') {
            steps {
                echo "${env.GREETING}, ${env.NAME}!"
            }
        }

        stage('End') {
            steps {
                echo 'Environment variables demonstration complete.'
            }
        }
    }
}
```

PIPELINE ACTIVITY 3: CONDITIONS



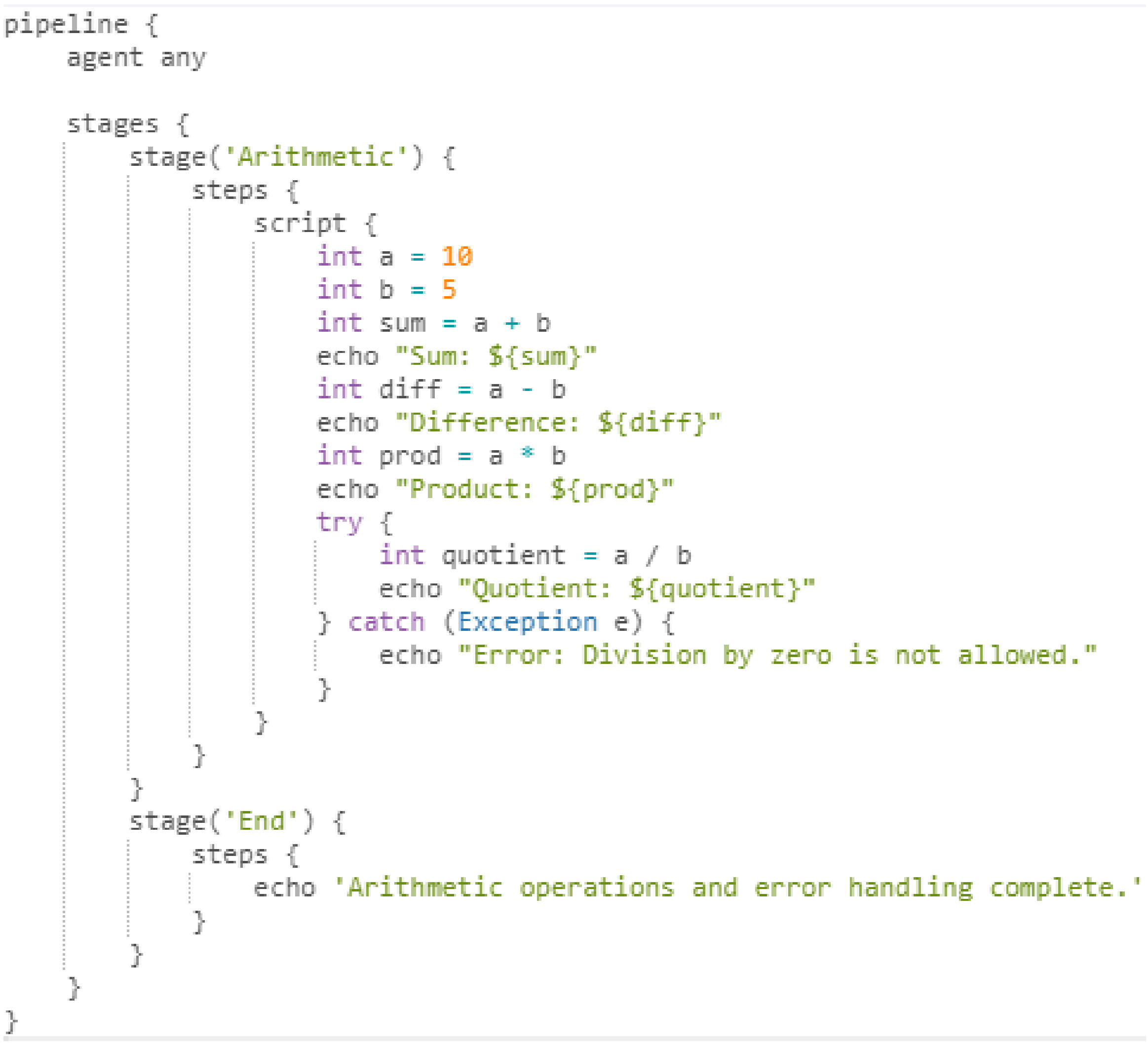
```
pipeline {
    agent any

    environment {
        RUN_STAGE = 'true'
    }

    stages {
        stage('Conditional Stage') {
            when {
                environment name: 'RUN_STAGE', value: 'true'
            }
            steps {
                echo 'This stage runs because RUN_STAGE is true.'
            }
        }
        stage('Always Run') {
            steps {
                echo 'This stage always runs.'
            }
        }
        stage('End') {
            steps {
                echo 'Conditional execution demonstration complete.'
            }
        }
    }
}
```

ACTIVITY 4:PARALLEL EXECUTION



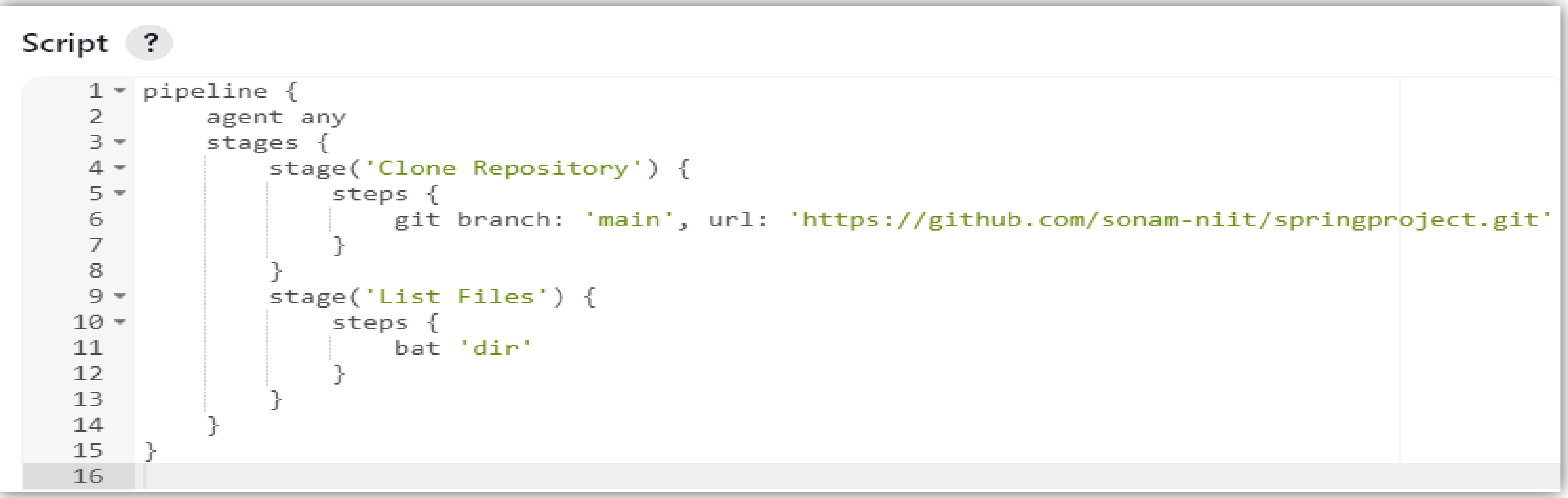
```
pipeline {
    agent any

    stages {
        stage('Parallel Execution') {
            parallel {
                stage('Stage 1') {
                    steps {
                        echo 'Running Stage 1'
                    }
                }
                stage('Stage 2') {
                    steps {
                        echo 'Running Stage 2'
                    }
                }
            }
        }
        stage('End') {
            steps {
                echo 'Parallel stages demonstration complete.'
            }
        }
    }
}
```

ACTIVITY 5:ARITHMETIC & ERROR HANDLING



```
pipeline {
    agent any

    stages {
        stage('Arithmetic') {
            steps {
                script {
                    int a = 10
                    int b = 5
                    int sum = a + b
                    echo "Sum: ${sum}"
                    int diff = a - b
                    echo "Difference: ${diff}"
                    int prod = a * b
                    echo "Product: ${prod}"
                    try {
                        int quotient = a / b
                        echo "Quotient: ${quotient}"
                    } catch (Exception e) {
                        echo "Error: Division by zero is not allowed."
                    }
                }
            }
        }
        stage('End') {
            steps {
                echo 'Arithmetic operations and error handling complete.'
            }
        }
    }
}
```

PIPELINE FOR GIT CLONE REPO

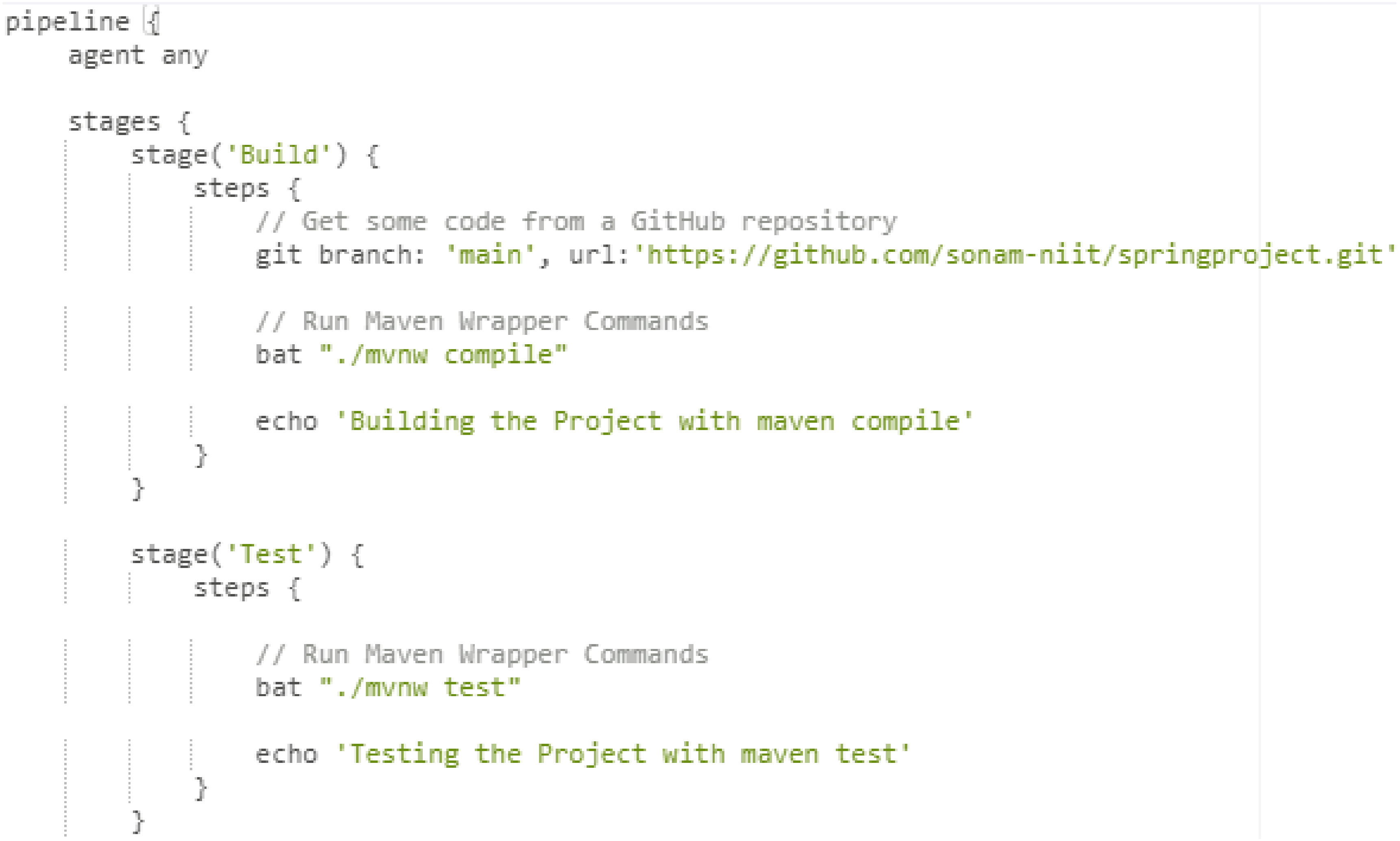


```
pipeline {
    agent any
    stages {
        stage('Clone Repository') {
            steps {
                git branch: 'main', url: 'https://github.com/sonam-niit/springproject.git'
            }
        }
        stage('List Files') {
            steps {
                bat 'dir'
            }
        }
    }
}
```

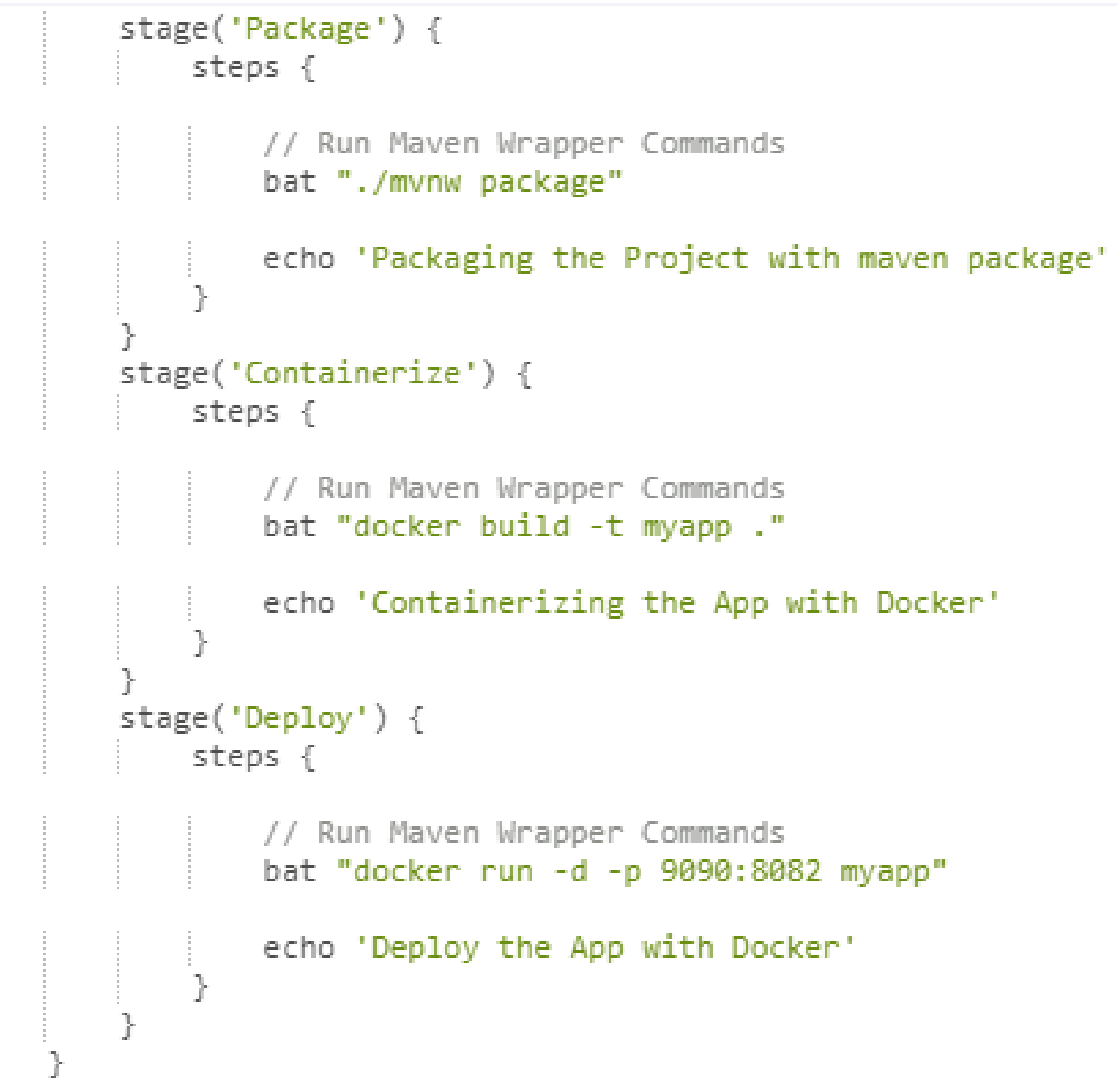
POLL SCM

- To set up Jenkins to automatically trigger a build whenever there are changes in your source code repository.
- Polling Schedule:
- In the **Schedule** field, you need to enter a cron-like schedule to tell Jenkins how often to check for changes.
  - H/5 * * * * - Poll every 5 minutes.
  - H 9 * * 1-5 - Poll at 9 AM every weekday.
  - H 2,14 * * * - Poll at 2 AM and 2 PM every
  - **H**: A Jenkins-specific feature that spreads load evenly by choosing a random value within the specified range.
  - ***/5***: Every 5 minutes.
  - *** * * * *** : Specifies the time format: minute hour day month dayOfWeek.

PROJECT PIPELINE

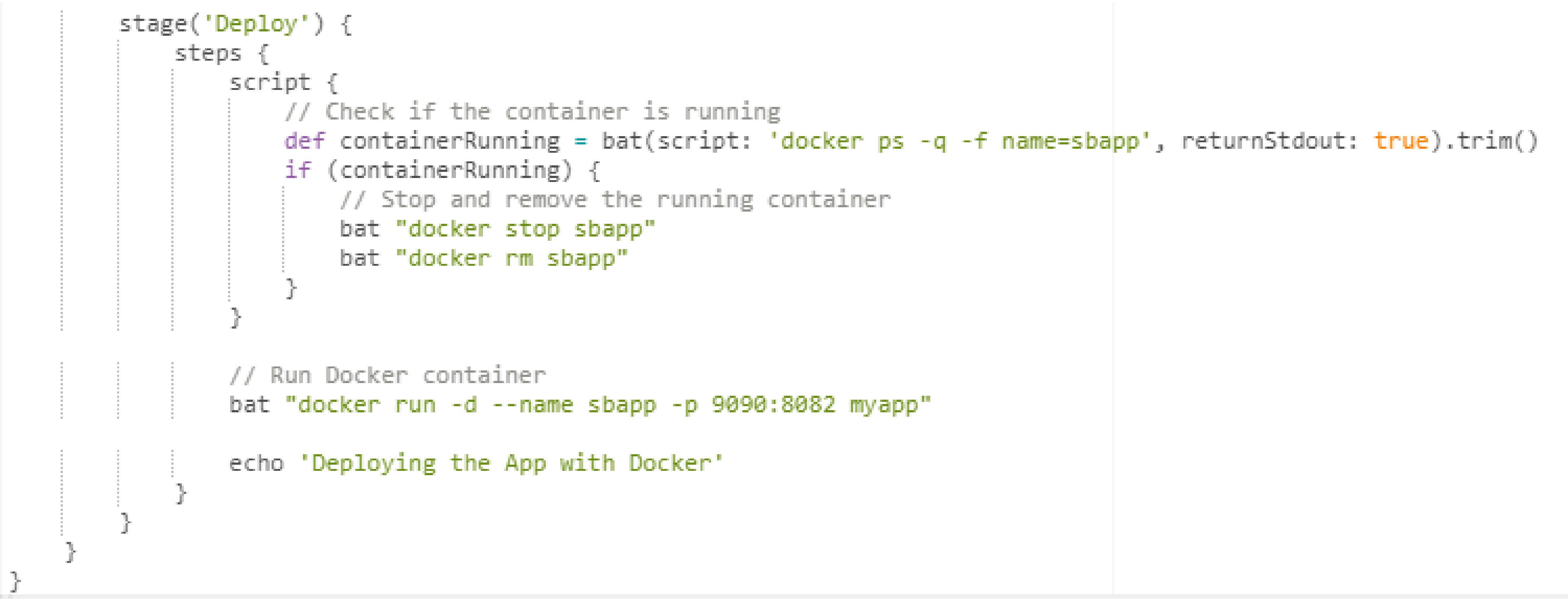


```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                // Get some code from a GitHub repository
                git branch: 'main', url:'https://github.com/sonam-niit/springproject.git'

                // Run Maven Wrapper Commands
                bat "./mvnw compile"

                echo 'Building the Project with maven compile'
            }
        }

        stage('Test') {
            steps {

                // Run Maven Wrapper Commands
                bat "./mvnw test"

                echo 'Testing the Project with maven test'
            }
        }
```



```
    stage('Package') {
        steps {

            // Run Maven Wrapper Commands
            bat "./mvnw package"

            echo 'Packaging the Project with maven package'
        }
    }
    stage('Containerize') {
        steps {

            // Run Maven Wrapper Commands
            bat "docker build -t myapp ."

            echo 'Containerizing the App with Docker'
        }
    }
    stage('Deploy') {
        steps {

            // Run Maven Wrapper Commands
            bat "docker run -d -p 9090:8082 myapp"

            echo 'Deploy the App with Docker'
        }
    }
}
```

IMPROVING DEPLOY STAGE



```
        stage('Deploy') {
            steps {
                script {
                    // Check if the container is running
                    def containerRunning = bat(script: 'docker ps -q -f name=sbapp', returnStdout: true).trim()
                    if (containerRunning) {
                        // Stop and remove the running container
                        bat "docker stop sbapp"
                        bat "docker rm sbapp"
                    }
                }

                // Run Docker container
                bat "docker run -d --name sbapp -p 9090:8082 myapp"

                echo 'Deploying the App with Docker'
            }
        }
    }
}
```