

# Jenkins Essentials: Automating Your CI/CD Pipeline

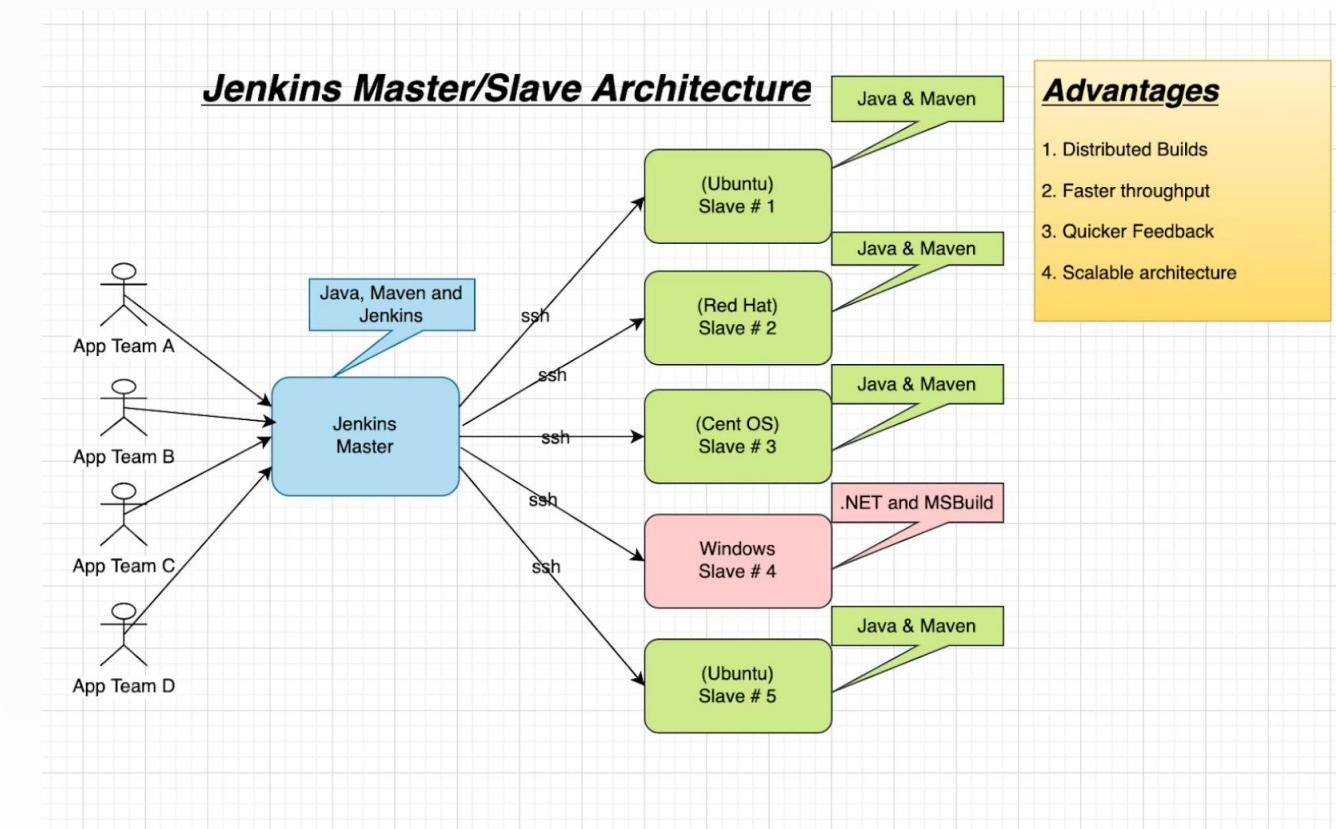
Welcome to this comprehensive guide on using Jenkins to streamline your development workflow. This presentation will cover fundamentals, pipeline types, security best practices, and a practical implementation example to get you started with CI/CD automation.

# Chapter 1: Understanding Jenkins Basics & Architecture

Jenkins has become the industry-standard tool for automation, helping teams move from manual builds to fully automated delivery pipelines. Before diving into advanced topics, let's understand the foundational concepts.

In this chapter, we'll explore:

- What Jenkins is and its core capabilities
- The master-agent architecture model
- How distributed builds work in practice



# What is Jenkins?

## Open-Source Automation

Jenkins is a self-contained, open-source automation server that orchestrates continuous integration and continuous delivery (CI/CD) workflows.

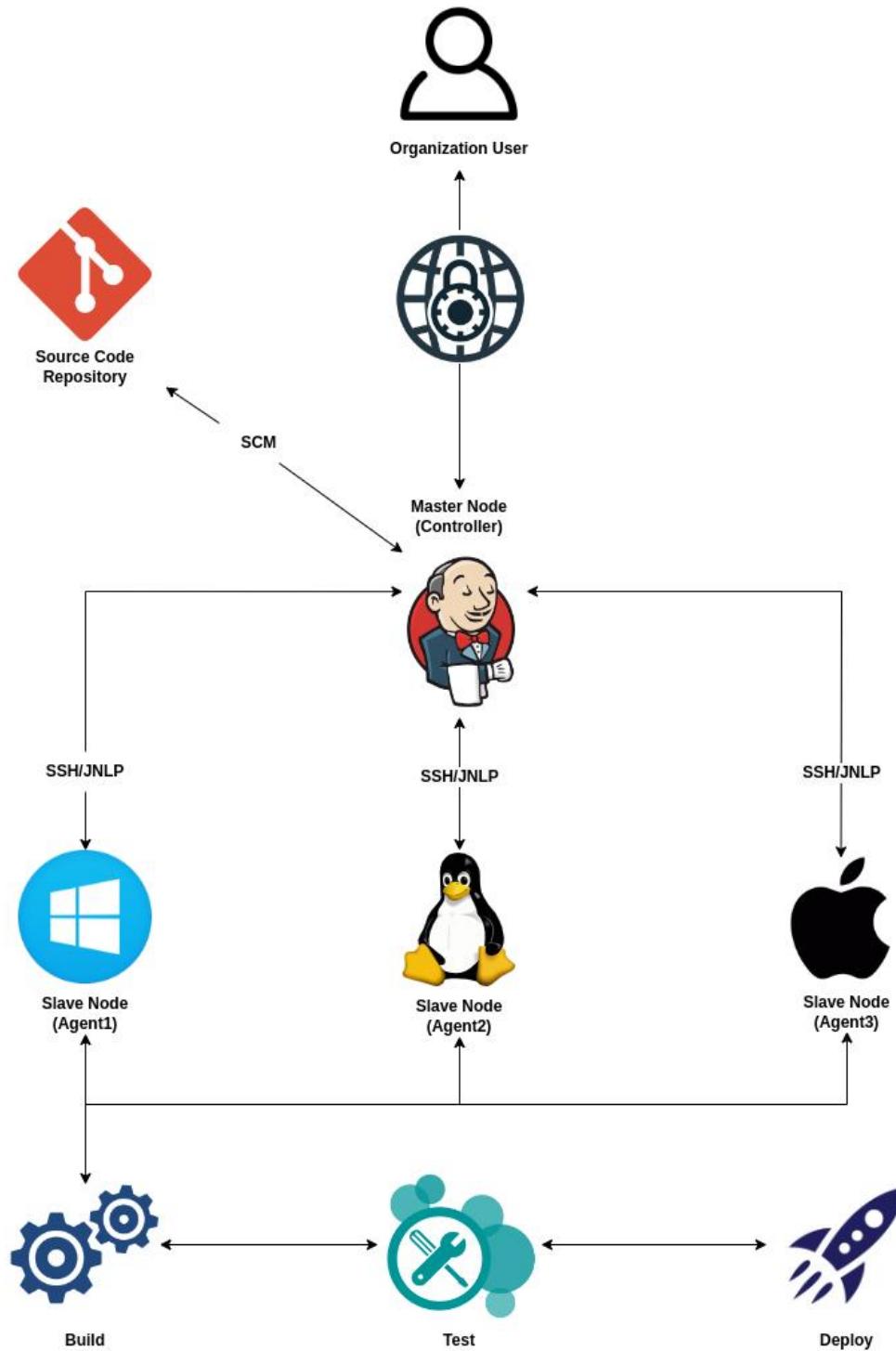
Jenkins' flexibility allows it to integrate with version control systems, build tools, testing frameworks, deployment platforms, and notification services—creating a unified automation pipeline.

## Highly Extensible

With 1,800+ community-contributed plugins, Jenkins adapts to virtually any technology stack or development process.

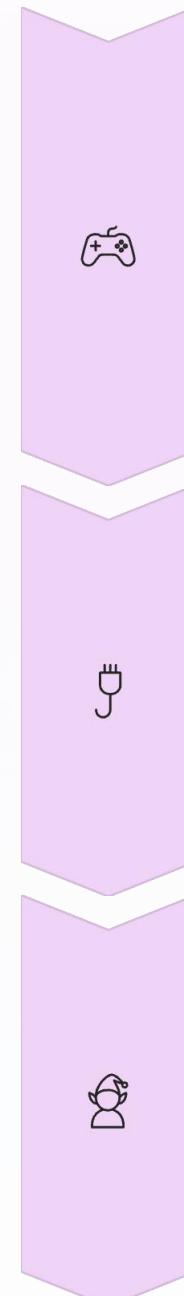
## Java-Based Engine

Built on Java for cross-platform compatibility, Jenkins supports distributed builds across multiple operating systems.



**Basic Jenkins Architecture**

# Jenkins Architecture Overview



## Master Node

Central controller that handles:

- Web UI and configuration
- Build job scheduling
- Agent management
- Result reporting



## Communication

Via:

- JNLP (Java Web Start)
- SSH connections
- Docker API



## Agent Nodes

Worker machines that:

- Execute actual build tasks
- Provide specialized environments
- Scale horizontally as needed

# Chapter 2: Jenkins Pipelines – Declarative vs Scripted

Jenkins pipelines provide a powerful way to define your entire build process as code, making it versioned, testable, and reusable.

Pipeline definitions live in your source code repository as [Jenkinsfile](#), enabling true pipeline-as-code practices.

In this chapter, we'll compare the two syntax models and explore how to structure your pipeline definitions effectively.



**Jenkins Pipeline Scripts: Declarative vs. Scripted Pipelines**

# Pipeline Types Explained

## Declarative Pipeline

A more structured approach with predefined syntax:

Begins with `pipeline` block

Uses predefined sections like `agent`, `stages`, `post`

- Simpler to learn, more readable
- Built-in input validation
- Ideal for standard build workflows

## Scripted Pipeline

Groovy-based scripting approach:

Begins with `node` block

- Full access to Groovy language features
- More flexible for complex logic
- Requires Groovy knowledge
- Better for advanced use cases with custom logic

- ❑ Both pipeline types support the same plugins and can accomplish similar goals. The choice depends on your team's familiarity with Groovy and the complexity of your build logic.

# Pipeline Types Example

## Declarative Pipeline

```
pipeline {  
    agent any  
  
    stages {  
        stage('Build') {  
            steps {  
                echo 'Building...'  
            }  
        }  
        stage('Test') {  
            steps {  
                echo 'Testing...'  
            }  
        }  
    }  
  
    post {  
        success {  
            echo 'Pipeline succeeded!'  
        }  
        failure {  
            echo 'Pipeline failed!'  
        }  
    }  
}
```

## Scripted Pipeline

```
node {  
    try {  
        stage('Build') {  
            echo 'Building...'  
        }  
        stage('Test') {  
            echo 'Testing...'  
        }  
    } catch (err) {  
        echo 'Pipeline failed!'  
        throw err  
    }  
}
```

# Jenkinsfile Structure Essentials

## Pipeline

The top-level container for the entire pipeline definition

## Stages

Logical divisions of the build process (Build, Test, Deploy, etc.)

## Steps

Individual commands that perform the actual work (shell scripts, plugin actions)

Each pipeline also typically includes [environment variables](#), [parameters](#), and [post-actions](#) for notifications or cleanup.

```
pipeline {  
    agent any  
    environment {  
        VERSION = '1.0.0'  
    }  
    stages {  
        stage('Build') {  
            steps {  
                sh 'mvn clean package'  
            }  
        }  
        stage('Test') {  
            steps {  
                sh 'mvn test'  
            }  
        }  
    }  
    post {  
        success {  
            echo 'Build succeeded!'  
        }  
    }  
}
```

# Chapter 3: Managing Secrets with Jenkins Credentials Store

Securing sensitive information is critical in any CI/CD pipeline.

Hardcoding credentials in scripts or configuration files creates significant security risks.

Jenkins provides a centralized credential management system that:

- Safely stores sensitive information
- Provides controlled access to secrets during builds
- Supports audit trails for credential usage

The screenshot shows the Jenkins web interface for managing credentials. At the top, there's a navigation bar with the Jenkins logo, a search bar, and some user icons. Below it, the breadcrumb navigation shows 'Dashboard > Manage Jenkins > Credentials'. The main title is 'Credentials'. A table lists one credential entry:

T	P	Store ↓	Domain	ID	Name
key icon	key icon	System	(global)	monokle-token	Monokle automation token to use remote policies

Below the table, there's a section titled 'Stores scoped to Jenkins' which shows a single store entry:

P	Store ↓	Domains
key icon	System	(global)

At the bottom left, there are icons for 'Icon: S M L'.

# Secure Credential Management



## Credential Types

- Username/password pairs
- SSH private keys
- Secret text (API tokens)
- Secret files (certificates)
- Docker host certificates



## Usage in Pipelines

### withCredentials block

- Environment variables binding
- Credential parameters
- Plugin-specific integrations



## Security Benefits

- Centralized management
- Encrypted storage
- Masked in logs
- Access control integration
- Credential rotation support

```
withCredentials([string(credentialsId: 'my-docker-token', variable: 'DOCKER_TOKEN')]) {  
    sh 'docker login -u myuser -p $DOCKER_TOKEN'  
}
```

# Chapter 4: Example Pipeline with Docker & SonarQube Integration

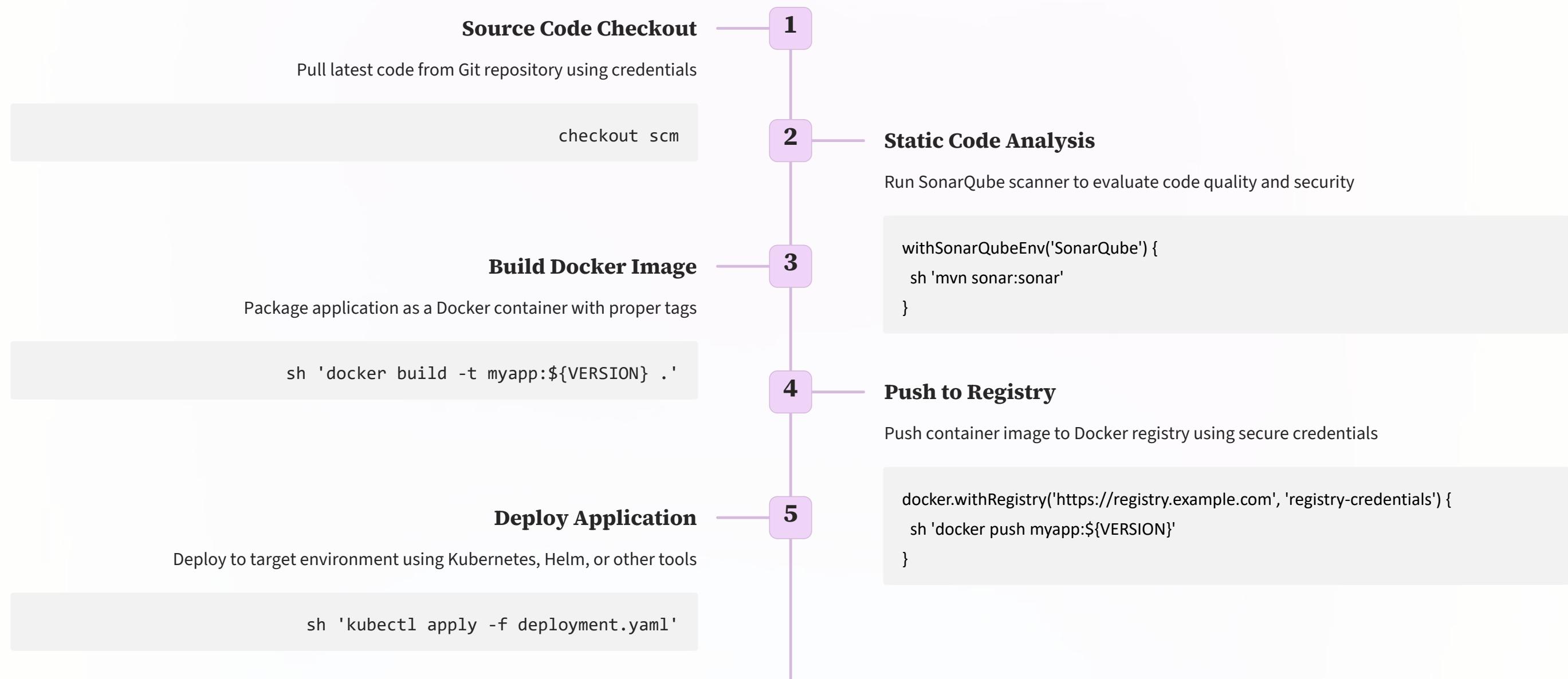
Let's explore a practical CI/CD pipeline that demonstrates how to:

- Pull code from a Git repository
- Perform static code analysis
- Build and package as a Docker image
- Push to a container registry
- Deploy to a target environment

This example showcases many of the concepts we've covered, including credentials management, pipeline structure, and integration with external tools.



# Pipeline Flow Overview



## Sample Declarative Jenkinsfile Snippet

```
pipeline {  
    agent {  
        docker {  
            image 'maven:3.8.4-openjdk-11'  
        }  
    }  
    environment {  
        DOCKER_REGISTRY = 'registry.example.com'  
        IMAGE_NAME = 'myapp'  
        VERSION = "${env.BUILD_NUMBER}"  
    }  
    stages {  
        stage('Checkout') {  
            steps {  
                checkout scm  
            }  
        }  
        stage('Code Analysis') {  
            steps {  
                withSonarQubeEnv('SonarQube') {  
                    sh 'mvn sonar:sonar'  
                }  
                timeout(time: 10, unit: 'MINUTES') {  
                    waitForQualityGate abortPipeline: true  
                }  
            }  
        }  
        stage('Build') {  
            steps {  
                sh 'mvn clean package'  
                sh "docker build -t ${DOCKER_REGISTRY}/${IMAGE_NAME}:${VERSION} ."  
            }  
        }  
        stage('Push') {  
            steps {  
                withCredentials([string(credentialsId: 'docker-registry-token', variable: 'DOCKER_TOKEN')])  
                sh "docker login ${DOCKER_REGISTRY} -u jenkins -p ${DOCKER_TOKEN}"  
                sh "docker push ${DOCKER_REGISTRY}/${IMAGE_NAME}:${VERSION}"  
            }  
        }  
        stage('Deploy') {  
            steps {  
                sh "sed -i 's|IMAGE_TAG|${VERSION}|g' deployment.yaml"  
                withKubeConfig([credentialsId: 'kubeconfig']) {  
                    sh 'kubectl apply -f deployment.yaml'  
                }  
            }  
        }  
        post {  
            success {  
                slackSend channel: '#deployments', color: 'good', message: "Deployment of  
${IMAGE_NAME}:${VERSION} successful!"  
            }  
            failure {  
                slackSend channel: '#deployments', color: 'danger', message: "Deployment of  
${IMAGE_NAME}:${VERSION} failed!"  
            }  
        }  
    }  
}
```