Jenkins Pipeline Documentation

Pipeline Overview

This Jenkins pipeline automates the process of building, testing, and deploying the Docker image for the Ambiview backend Ruby on Rails application. It performs the following steps:

1 . Environment Variables:

- These variables define the Docker image details, Git repository, and workspace directory on the server, which allows easy configuration and reusability across stages.

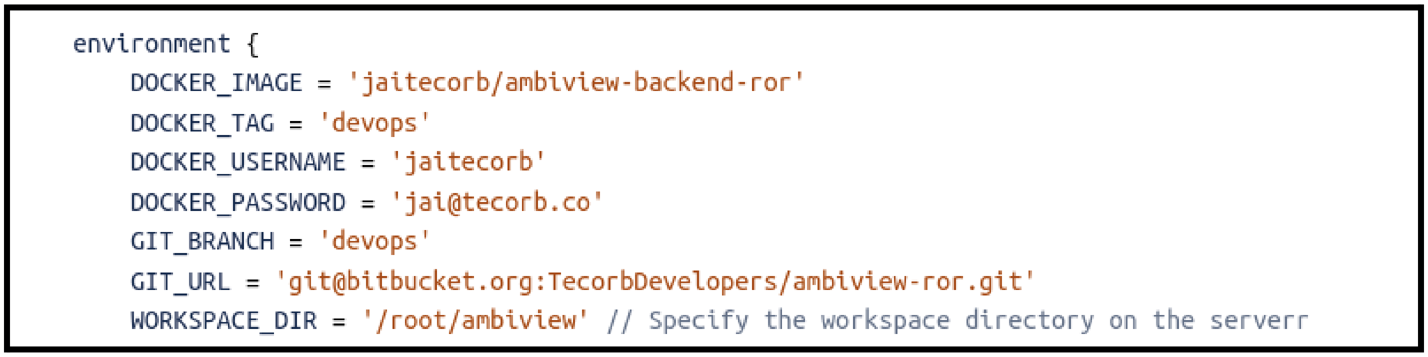
2 . Stages:

- **Create Workspace Directory**: Checks if the workspace directory exists on the remote server. If not, it creates the directory.
- **Checkout**: Clones or pulls the latest code from the specified Git repository branch. If the repo directory already exists, it performs a `git pull` to update it; otherwise, it clones the repo.
- **Build Docker Image**: Build a Docker image from the Dockerfile located in the `.dockerdev` directory in the repo.
- **Test Docker Image**: Runs tests on the Docker image. If no tests are available, it outputs a message indicating this.
- **Push Docker Image**: Logs into Docker Hub using the provided credentials, then pushes the built image to the Docker Hub repository.
- **Cleanup Old Docker Image**: Removes the old Docker image from the remote server.
- **Pull Latest Docker Image**: Logs into Docker Hub and pulls the latest version of the Docker image from Docker Hub to the server.
- **Run Docker Compose**: Executes the `docker-compose up` command to start the application in detached mode (`-d`). If any containers from a previous deployment are running, they are stopped and removed before starting the new ones.

Environment Variables

These are set at the beginning of the pipeline for reusability and easier configuration:

- **DOCKER_IMAGE**: Specifies the name of the Docker image to be used.
- **DOCKER_TAG**: Tag for the Docker image, identifying the version or environment (in this case, **devops**).
- **DOCKER_USERNAME** **&** **DOCKER_PASSWORD**: Docker Hub credentials for pushing and pulling the Docker image.
- **GIT_BRANCH** **&** **GIT_URL:** Specifies the branch and repository URL to fetch the application code.
- **WORKSPACE_DIR:** Defines the directory path on the remote server where the application will be stored and deployed.



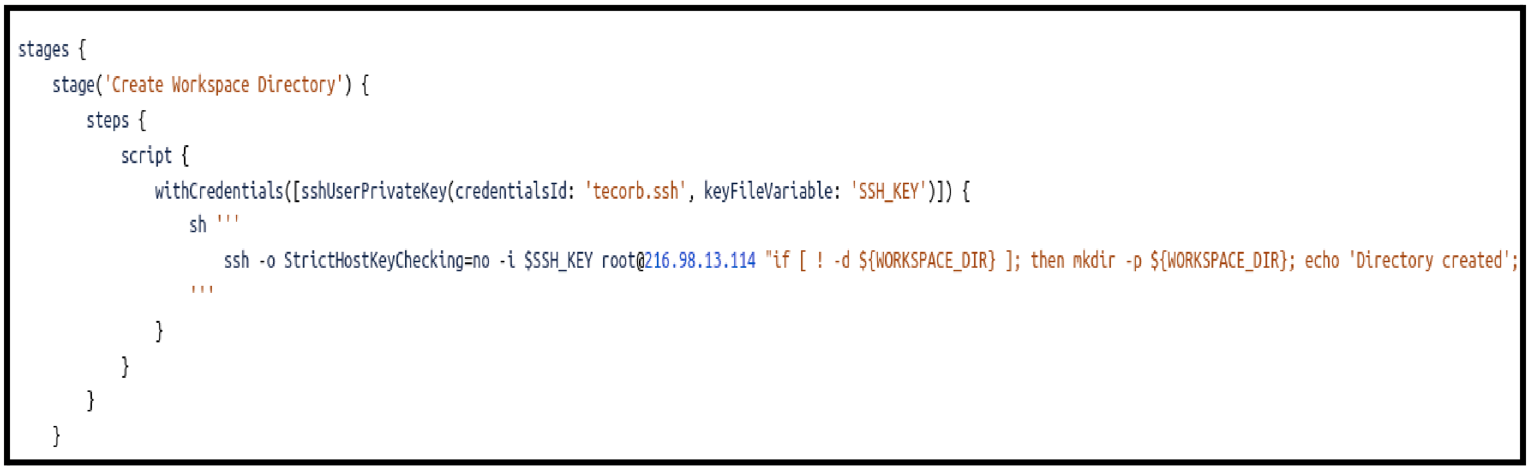
```
environment {
    DOCKER_IMAGE = 'jaitecorb/ambiview-backend-ror'
    DOCKER_TAG = 'devops'
    DOCKER_USERNAME = 'jaitecorb'
    DOCKER_PASSWORD = 'jai@tecorb.co'
    GIT_BRANCH = 'devops'
    GIT_URL = 'git@bitbucket.org:TecorbDevelopers/ambiview-ror.git'
    WORKSPACE_DIR = '/root/ambiview' // Specify the workspace directory on the serverr
```

Stages

1. Create Workspace Directory

This stage is called **'Create Workspace Directory'** and its purpose is to ensure that a specific directory exists on a remote server where files for the project will be stored.

- **Goal**: Ensure the specified directory exists on the remote server for the application's files.
- **Process**:
  - Connects to the server using SSH.
  - Check if the directory (**/root/ambiview**) exists.
  - If it doesn't exist, the directory is created, and a confirmation message is displayed.
  - If it exists, it logs that the directory already exists.



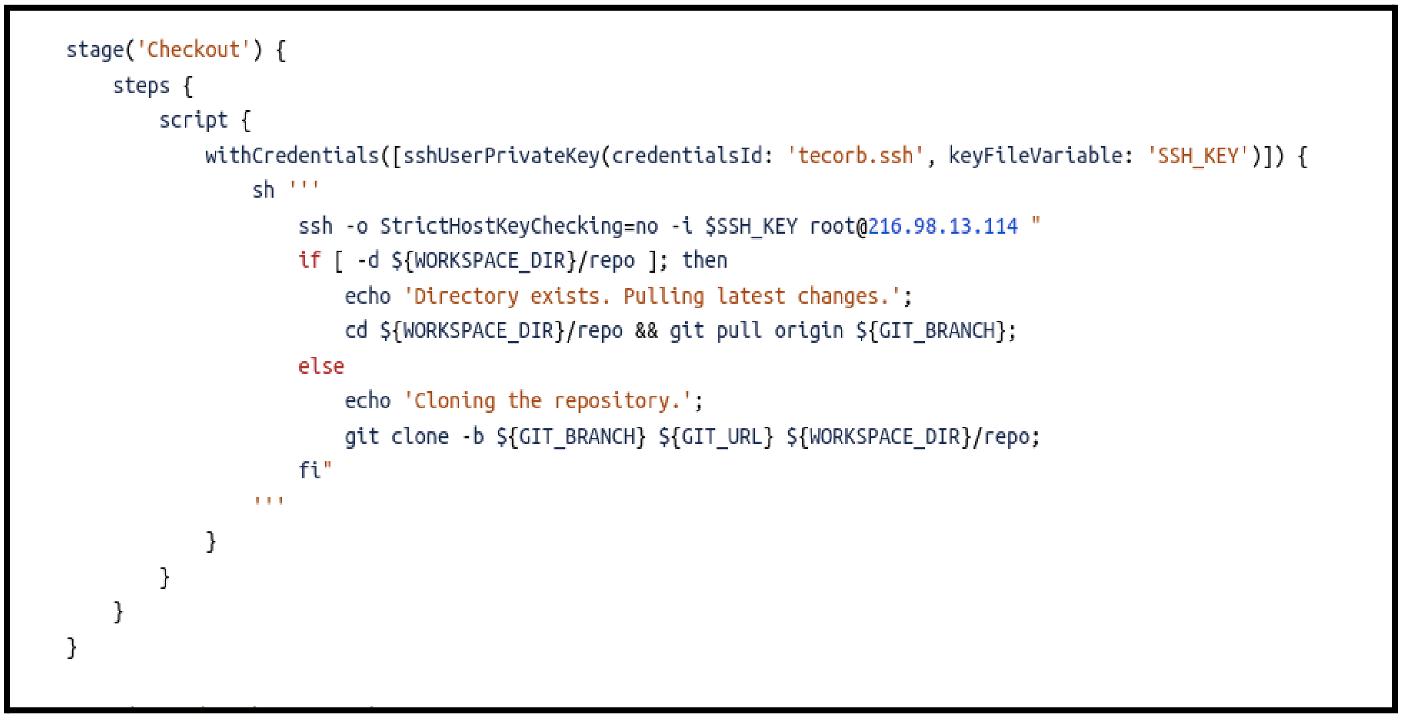
```
stages {
    stage('Create Workspace Directory') {
        steps {
            script {
                withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                    sh '''
                        ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "if [ ! -d ${WORKSPACE_DIR} ]; then mkdir -p ${WORKSPACE_DIR}; echo 'Directory created';
                    '''
                }
            }
        }
    }
```

Steps Breakdown:

1. **SSH Connection Setup:**
    - The **withCredentials** block is used to securely provide the SSH credentials (**tecorb.ssh)**, which are used to connect to the remote server.
    - The SSH private key (**SSH_KEY**) is used for authentication when connecting to the server at **216.98.13.114.**
2. **SSH Command Execution:**
    - The **sh** block is used to execute a shell command on the remote server.
    - The **ssh** command connects to the remote server, where:
        - The **-o StrictHostKeyChecking=no** option ensures that the SSH connection doesn't prompt to verify the host's authenticity. This is useful when automating the process.
        - The **-i** $**SSH_KEY** option provides the private key to authenticate the connection.
        - The remote command checks if the directory **${WORKSPACE_DIR}** exists on the server.
3. **Directory Check and Creation:**
    - The command **if [! -d ${WORKSPACE_DIR}]; then mkdir -p ${WORKSPACE_DIR}; echo 'Directory created'; else echo 'Directory already exists'; fi** does the following:
        - It checks whether the directory **${WORKSPACE_DIR}** exists.
        - If the directory does **not exist**, it creates the directory and prints "Directory created".
        - If the directory already exists, it simply prints "Directory already exists".

2. Checkout

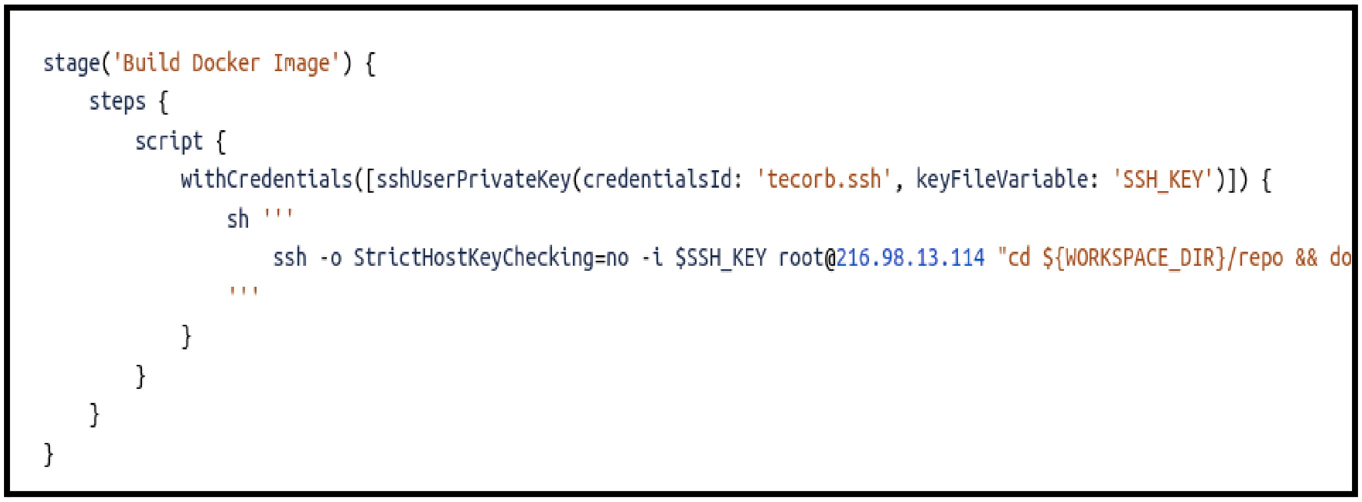
- **Goal**: Clone the repository or pull the latest changes if it's already cloned.
- **Process**:
  - Connects to the server and checks if the repo directory exists within the workspace.
  - If the directory exists, it navigates to the directory and runs **git pull** to fetch the latest changes from the **devops** branch.
  - If the directory does not exist, it clones the repository into the workspace.



```
stage('Checkout') {
    steps {
        script {
            withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                sh '''
                    ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "
                    if [ -d ${WORKSPACE_DIR}/repo ]; then
                        echo 'Directory exists. Pulling latest changes.';
                        cd ${WORKSPACE_DIR}/repo && git pull origin ${GIT_BRANCH};
                    else
                        echo 'Cloning the repository.';
                        git clone -b ${GIT_BRANCH} ${GIT_URL} ${WORKSPACE_DIR}/repo;
                    fi"
                '''
            }
        }
    }
}
```

3. Build Docker Image

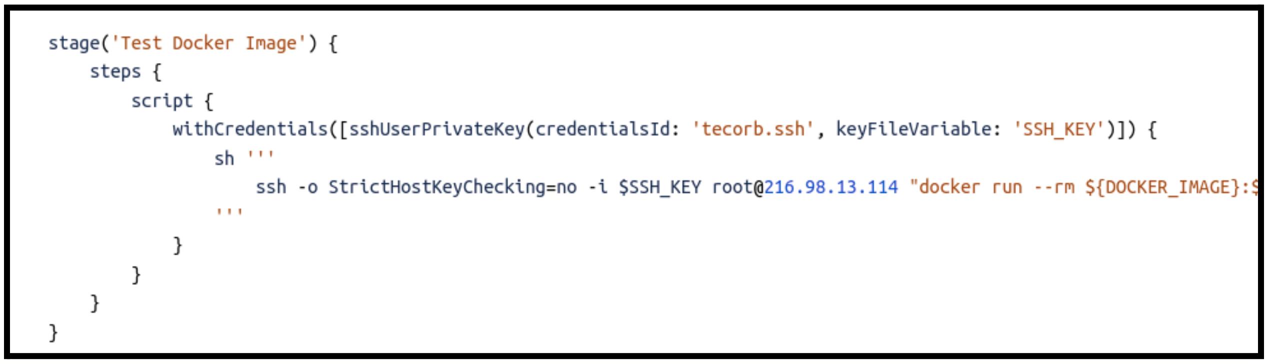
- **Goal**: Build a Docker image from the Dockerfile located in the repo.
- **Process**:
  - Connects to the server and navigates to the repo directory.
  - Builds a Docker image using the Dockerfile in **.dockerdev**, tagging the image as **jaitecorb/ambiview-backend-ror:devops.**



```
stage('Build Docker Image') {
    steps {
        script {
            withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                sh '''
                    ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "cd ${WORKSPACE_DIR}/repo && do
                '''
            }
        }
    }
}
```

4. **Test Docker Image**

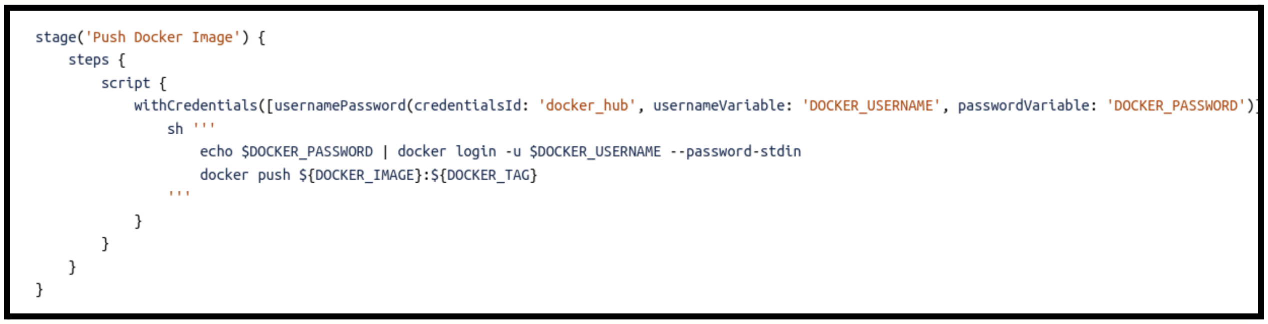
- **Goal**: Run tests to ensure the Docker image is functional.
- **Process**:
  - Connects to the server and runs a test command within a temporary Docker container.
  - If a test command is provided and it passes, it proceeds; if there are no tests available, a message is displayed, and it moves to the next stage.



```
stage('Test Docker Image') {
    steps {
        script {
            withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                sh '''
                    ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "docker run --rm ${DOCKER_IMAGE}:$
                '''
            }
        }
    }
}
```

5. **Push Docker Image**

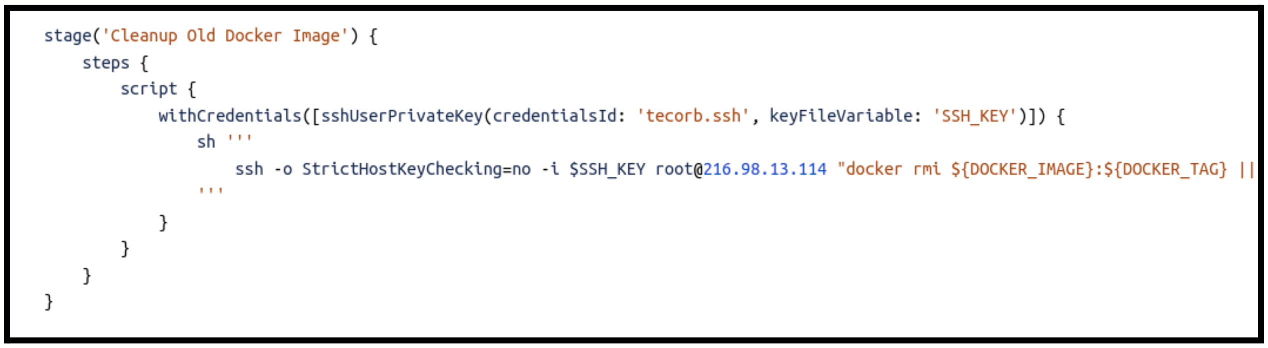
- **Goal**: Push the Docker image to Docker Hub for distribution.
- **Process**:
  - Log into Docker Hub using the credentials.
  - Pushes the Docker image with the tag devops to the Docker repository.



```
stage('Push Docker Image') {
    steps {
        script {
            withCredentials([usernamePassword(credentialsId: 'docker_hub', usernameVariable: 'DOCKER_USERNAME', passwordVariable: 'DOCKER_PASSWORD')]
                sh '''
                    echo $DOCKER_PASSWORD | docker login -u $DOCKER_USERNAME --password-stdin
                    docker push ${DOCKER_IMAGE}:${DOCKER_TAG}
                '''
            }
        }
    }
}
```

6. **Cleanup Old Docker Image**

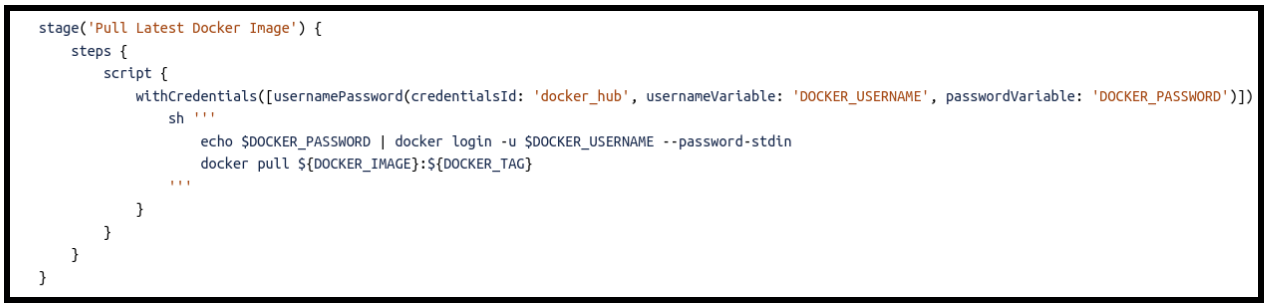
- **Goal**: Remove any old versions of the Docker image from the server.
- **Process**:
  - Connects to the server.
  - Attempts to remove the Docker image tagged **devops** from the server.
  - If the image doesn't exist, a message indicating "Image not found" is displayed.



```
stage('Cleanup Old Docker Image') {
    steps {
        script {
            withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                sh '''
                    ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "docker rmi ${DOCKER_IMAGE}:${DOCKER_TAG} ||
                '''
            }
        }
    }
}
```

7. **Pull Latest Docker Image**

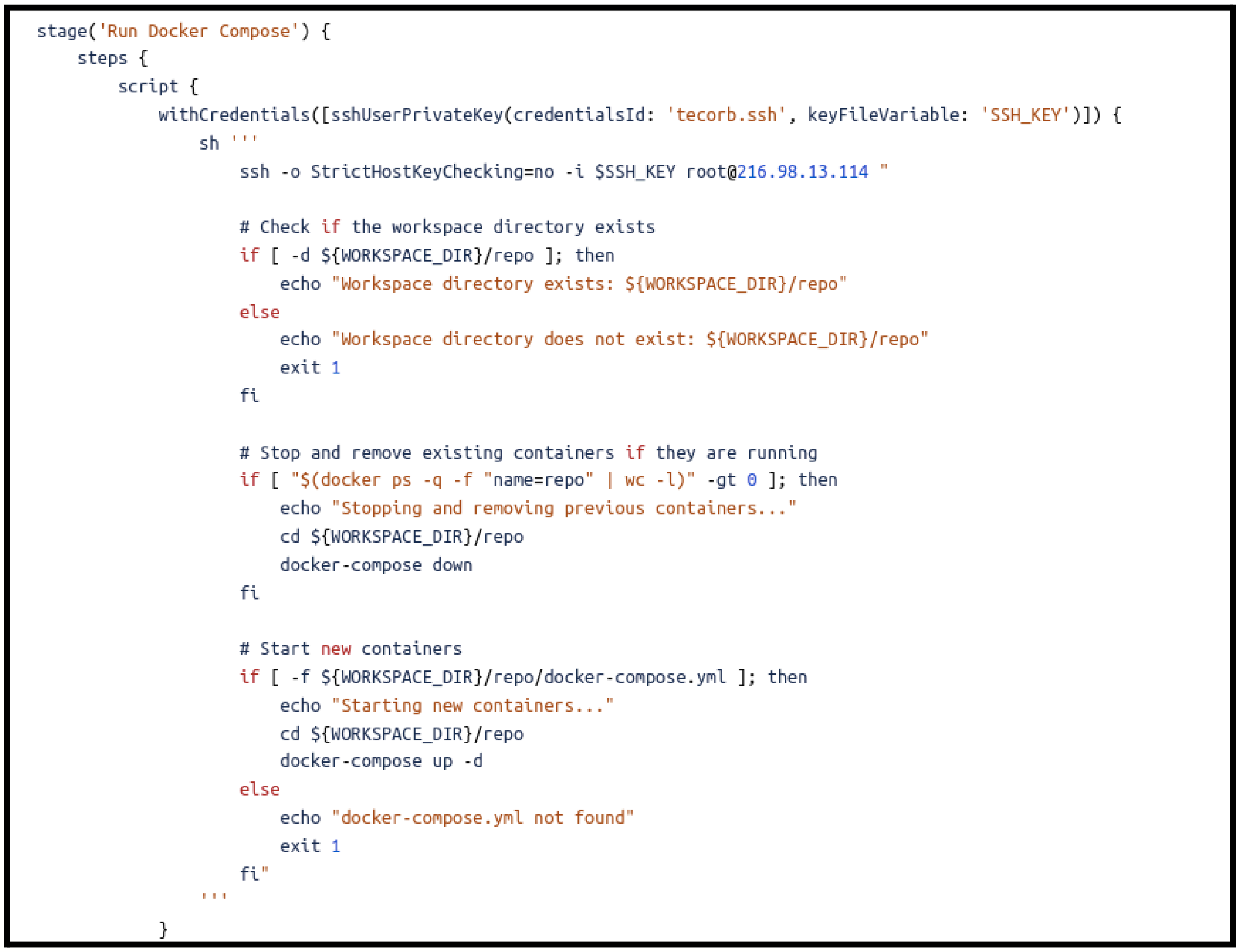
- **Goal**: Retrieve the latest Docker image from Docker Hub for deployment.
- **Process**:
  - Log into Docker Hub using the credentials.
  - Pulls the Docker image with the tag **devops** from Docker Hub.



```
stage('Pull Latest Docker Image') {
    steps {
        script {
            withCredentials([usernamePassword(credentialsId: 'docker_hub', usernameVariable: 'DOCKER_USERNAME', passwordVariable: 'DOCKER_PASSWORD')])
                sh '''
                    echo $DOCKER_PASSWORD | docker login -u $DOCKER_USERNAME --password-stdin
                    docker pull ${DOCKER_IMAGE}:${DOCKER_TAG}
                '''
            }
        }
    }
}
```

8. Run Docker Compose

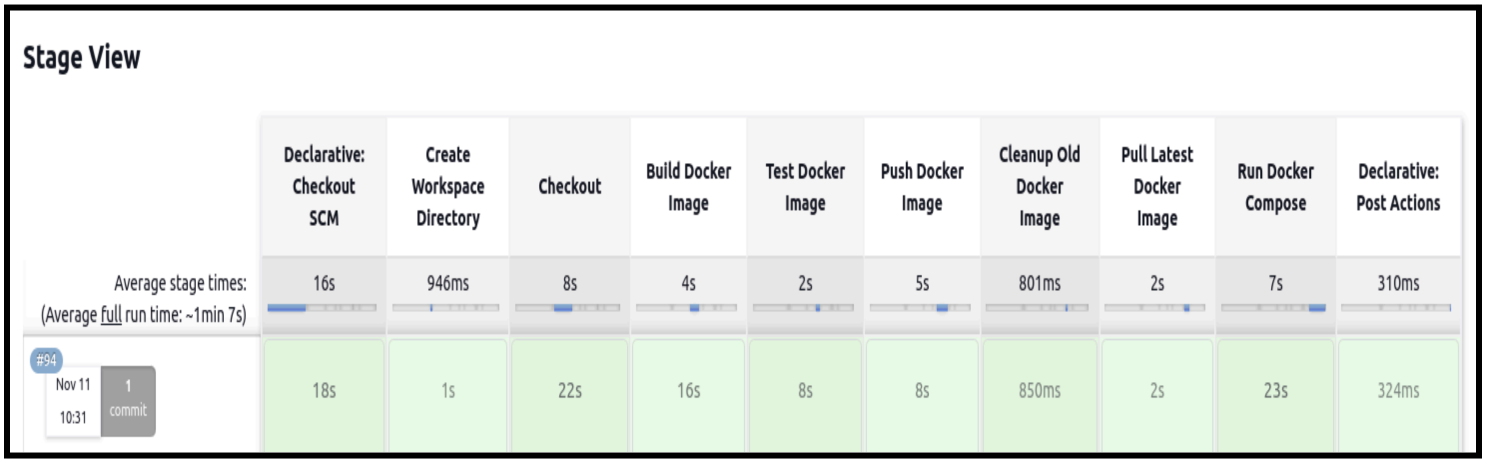
- **Goal**: Deploy the application using Docker Compose.
- **Process**:
  - Connects to the server and verifies if the repo directory and **docker-compose.yml** file exist.
  - If existing containers are running with the same name, it stops and removes them.
  - Starts the application using **docker-compose up -d,** launching the services defined in **docker-compose.yml** in detached mode.



```
stage('Run Docker Compose') {
    steps {
        script {
            withCredentials([sshUserPrivateKey(credentialsId: 'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                sh '''
                    ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@216.98.13.114 "

                    # Check if the workspace directory exists
                    if [ -d ${WORKSPACE_DIR}/repo ]; then
                        echo "Workspace directory exists: ${WORKSPACE_DIR}/repo"
                    else
                        echo "Workspace directory does not exist: ${WORKSPACE_DIR}/repo"
                        exit 1
                    fi

                    # Stop and remove existing containers if they are running
                    if [ "$(docker ps -q -f "name=repo" | wc -l)" -gt 0 ]; then
                        echo "Stopping and removing previous containers..."
                        cd ${WORKSPACE_DIR}/repo
                        docker-compose down
                    fi

                    # Start new containers
                    if [ -f ${WORKSPACE_DIR}/repo/docker-compose.yml ]; then
                        echo "Starting new containers..."
                        cd ${WORKSPACE_DIR}/repo
                        docker-compose up -d
                    else
                        echo "docker-compose.yml not found"
                        exit 1
                    fi"
                '''
            }
```

How the Connection Works

Credentials: The SSH connection uses an SSH private key (**tecorb.ssh**), which is stored in Jenkins Credentials and specified in the **withCredentials** block.



Stage View

	Declarative: Checkout SCM	Create Workspace Directory	Checkout	Build Docker Image	Test Docker Image	Push Docker Image	Cleanup Old Docker Image	Pull Latest Docker Image	Run Docker Compose	Declarative: Post Actions
Average stage times: (Average full run time: ~1min 7s)	16s	946ms	8s	4s	2s	5s	801ms	2s	7s	310ms
#94 Nov 11 10:31 1 commit	18s	1s	22s	16s	8s	8s	850ms	2s	23s	324ms

....

```
pipeline {
   agent any

   environment {
       DOCKER_IMAGE = 'jaitecorb/ambiview-backend-ror'
       DOCKER_TAG = 'devops'
       DOCKER_USERNAME = 'jaitecorb'
       DOCKER_PASSWORD = 'jai@tecorb.co'
       GIT_BRANCH = 'devops'
       GIT_URL = 'git@bitbucket.org:TecorbDevelopers/ambiview-ror.git'
       WORKSPACE_DIR = '/root/ambiview' // Specify the workspace directory on
the serverr
   }
```

```
    stages {
        stage('Create Workspace Directory') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "if [ ! -d ${WORKSPACE_DIR} ]; then mkdir -p
${WORKSPACE_DIR}; echo 'Directory created'; else echo 'Directory already
exists'; fi"
                        '''
                    }
                }
            }
        }

        stage('Checkout') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "
                            if [ -d ${WORKSPACE_DIR}/repo ]; then
                                echo 'Directory exists. Pulling latest
changes.';
                                cd ${WORKSPACE_DIR}/repo && git pull origin
${GIT_BRANCH};
                            else
                                echo 'Cloning the repository.';
                                git clone -b ${GIT_BRANCH} ${GIT_URL}
${WORKSPACE_DIR}/repo;
                            fi"
                        '''
                    }
                }
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "cd ${WORKSPACE_DIR}/repo && docker build -t
${DOCKER_IMAGE}:${DOCKER_TAG} -f ./.dockerdev/Dockerfile ."
                        '''
                    }
                }
            }
        }

        stage('Test Docker Image') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "docker run --rm ${DOCKER_IMAGE}:${DOCKER_TAG}
your-test-command || echo 'No tests available'"
                        '''
                    }
                }
            }
        }
        stage('Push Docker Image') {
            steps {
                script {
                    withCredentials([usernamePassword(credentialsId:
'docker_hub', usernameVariable: 'DOCKER_USERNAME', passwordVariable:
'DOCKER_PASSWORD')]) {
                        sh '''
                            echo $DOCKER_PASSWORD | docker login -u
$DOCKER_USERNAME --password-stdin
                            docker push ${DOCKER_IMAGE}:${DOCKER_TAG}
                        '''
                    }
                }
            }
        }

        stage('Cleanup Old Docker Image') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "docker rmi ${DOCKER_IMAGE}:${DOCKER_TAG} || echo 'Image
not found'"
                        '''
                    }
                }
            }
        }

        stage('Pull Latest Docker Image') {
            steps {
                script {
                    withCredentials([usernamePassword(credentialsId:
'docker_hub', usernameVariable: 'DOCKER_USERNAME', passwordVariable:
'DOCKER_PASSWORD')]) {
                        sh '''
                            echo $DOCKER_PASSWORD | docker login -u
$DOCKER_USERNAME --password-stdin
                            docker pull ${DOCKER_IMAGE}:${DOCKER_TAG}
                        '''
                    }
                }
            }
        }

        stage('Run Docker Compose') {
            steps {
                script {
                    withCredentials([sshUserPrivateKey(credentialsId:
'tecorb.ssh', keyFileVariable: 'SSH_KEY')]) {
                        sh '''
                            ssh -o StrictHostKeyChecking=no -i $SSH_KEY
root@216.98.13.114 "

                            # Check if the workspace directory exists
                            if [ -d ${WORKSPACE_DIR}/repo ]; then
                                echo "Workspace directory exists:
${WORKSPACE_DIR}/repo"
                            else
                                echo "Workspace directory does not exist:
${WORKSPACE_DIR}/repo"
                                exit 1
                            fi

                            # Stop and remove existing containers if they are
running
                            if [ "$(docker ps -q -f "name=repo" | wc -l)" -gt 0
]; then
```

```
                                echo "Stopping and removing previous
containers..."
                                cd ${WORKSPACE_DIR}/repo
                                docker-compose down
                            fi

                            # Start new containers
                            if [ -f ${WORKSPACE_DIR}/repo/docker-compose.yml ];
then
                                echo "Starting new containers..."
                                cd ${WORKSPACE_DIR}/repo
                                docker-compose up -d
                            else
                                echo "docker-compose.yml not found"
                                exit 1
                            fi"
                        '''
                    }
                }
            }
        }
    }

    post {
        always {
            cleanWs()
        }
        success {
            echo 'Pipeline completed successfully!'
        }
        failure {
            echo 'Pipeline failed! Check logs for details.'
        }
    }
}
```