ODSOFT: Iteration 1 Critical Analysis

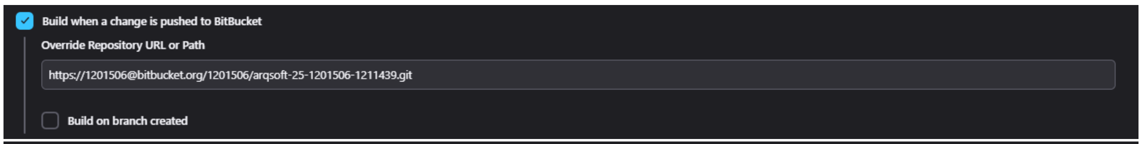
1201506 1211439

This document addresses the optimization strategies implemented within our Jenkins pipeline process, focusing on enhancing efficiency and reducing runtime. We explore various techniques, including the parallel execution of tests, the strategic skipping of unnecessary stages, and the implementation of multi-threading for mutation testing. Additionally, we analyze the impact of these optimizations on overall performance, alongside potential areas for further improvement in future iterations. Through this examination, we aim to establish best practices that will contribute to a more effective and streamlined continuous integration and deployment workflow.

Jenkins Pipeline

Integration with Bitbucket

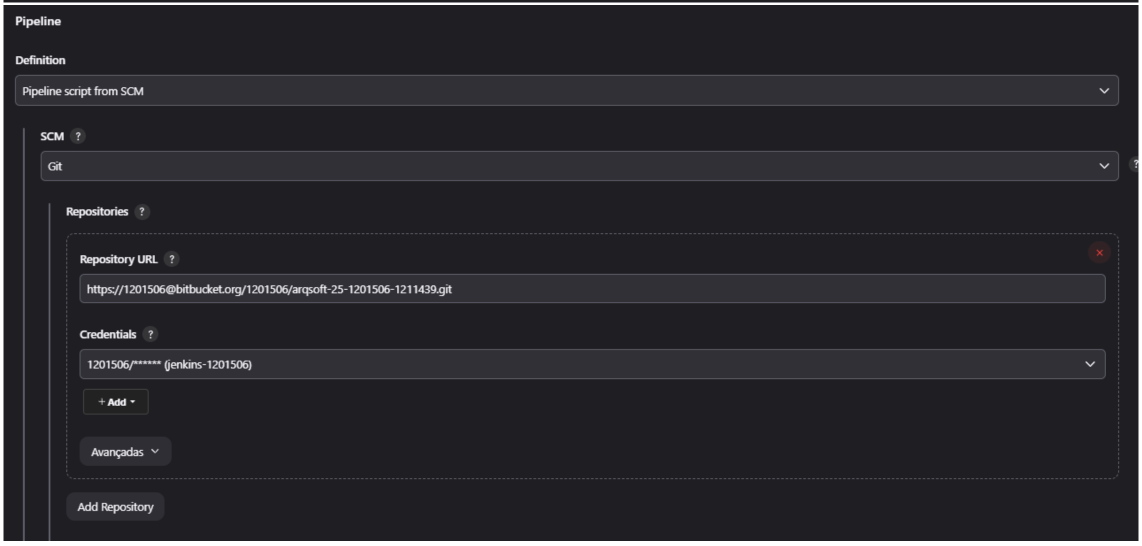
To trigger Jenkins Pipeline builds from Commits, we have set a Webhook on our Bitbucket repository with the following configurations:



✓ Build when a change is pushed to BitBucket

Override Repository URL or Path: https://1201506@bitbucket.org/1201506/arqsoft-25-1201506-1211439.git

☐ Build on branch created



Pipeline

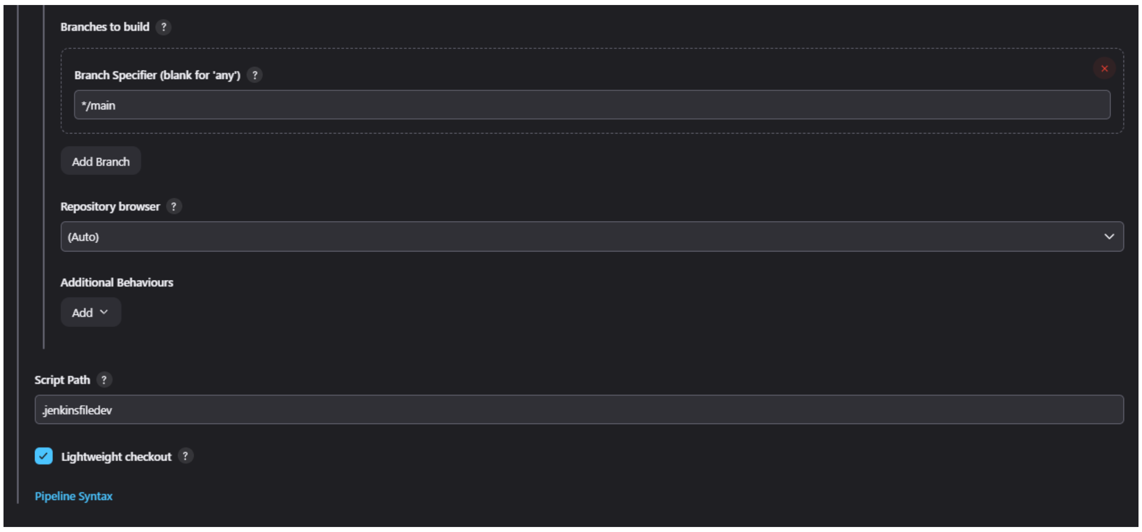
Definition: Pipeline script from SCM

SCM: Git

Repositories

Repository URL: https://1201506@bitbucket.org/1201506/arqsoft-25-1201506-1211439.git

Credentials: 1201506/****** (jenkins-1201506)



Branches to build

Branch Specifier (blank for 'any'): */main

Repository browser: (Auto)

Additional Behaviours

Script Path: jenkinsfiledev

✓ Lightweight checkout

It targets specifically the ***Bitbucket Webhook*** which is critical to trigger the **Bitbucket Plugin** in Jenkins, to **trigger** the pipeline on push events.

Tools and Plugins

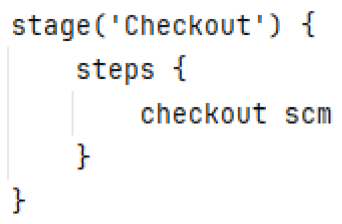
Apart from the Recommended Tools installed in Jenkins by default, we have chosen to utilize the following:

- **Blue Ocean**: Provides an intuitive, user-friendly interface for pipeline visualization and management. *Blue Ocean* is particularly useful for tracking performance improvements and monitoring the overall flow of the pipeline in a graphical format.
- **JaCoCo Plugin**: Offers comprehensive code coverage reports and trends, enabling us to monitor test coverage across different *builds* and ensure our codebase meets quality standards.
- **Maven Integration**: Simplifies the use of Maven across builds by abstracting binary paths, creating a more operating-system-agnostic environment for job execution, and ensuring consistency across different setups.
- **Pipeline: Stage View Plugin**: Enables a straightforward view of pipeline stages directly from the Jenkins dashboard. This plugin provides a simplified, easy-to-navigate view of pipeline stages without needing to switch to the separate Blue Ocean interface.
- **SonarQube Scanner Plugin**: Integrates SonarQube analysis directly into the Jenkins pipeline, enabling static code analysis and generating detailed insights into code quality, potential bugs, and vulnerabilities.

1. Source Code Management (SCM)

Bitbucket is currently the Source Code Management System of choice.

Bitbucket serves as our primary Source Code Management (SCM) system. In our Jenkins pipeline, the source code is retrieved through the following stage:



```
stage('Checkout') {
    steps {
        checkout scm
    }
}
```

This approach leverages Jenkins' built-in capability to automatically detect the SCM configuration, the specified repository, and the target branch defined in the pipeline's settings. By using '***checkout scm'***, we avoid the redundancy of explicitly the repository. This method also enhances flexibility, allowing for seamless scalability and plugin extensibility if further SCM-related functionality is required in the future.

2. **Build**

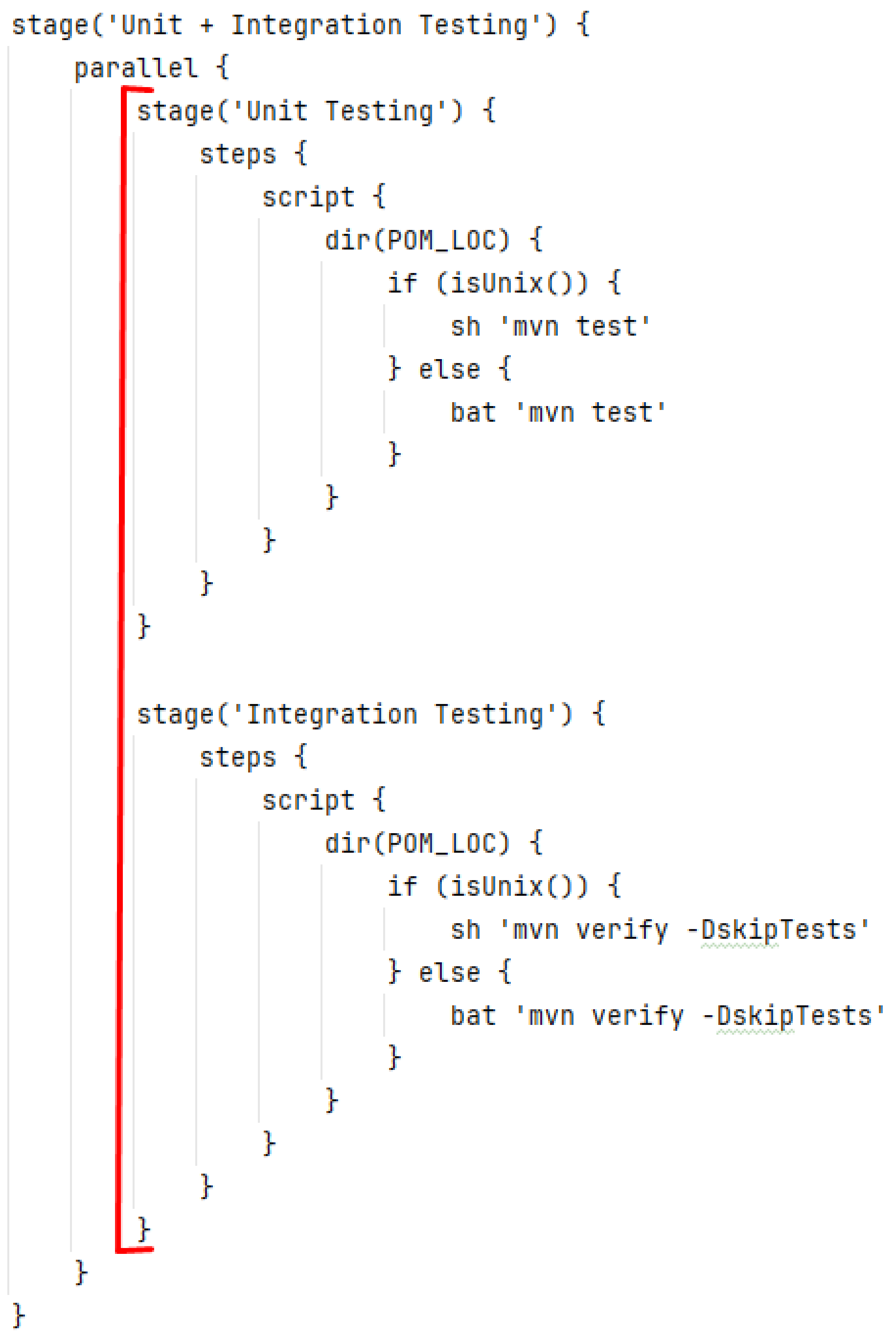
This **Build** stage in the Jenkins pipeline clean and compiles the source code and packages the application into a deployable format (e.g., JAR or WAR, JAR in our case) using Maven, while skipping the execution of tests to speed up the build process.

'-DskipTests' This option skips test execution, which saves time during the build phase by focusing solely on creating the package without validating it through tests, what we are going to do next.

```
stage('Build') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'mvn clean package -DskipTests'
                } else {
                    bat 'mvn clean package -DskipTests'
                }
            }
        }
    }
}
```

3. Unit + Mutation Testing

This ‘**Unit + Integration Testing**’ stage in the Jenkins pipeline performs two types of testing—unit tests and integration tests—concurrently to improve efficiency. By running these tests in parallel, it reduces the overall testing time and ensures both the individual components (through unit tests) and the application’s integration points (through integration tests) are validated.



```
stage('Unit + Integration Testing') {
    parallel {
        stage('Unit Testing') {
            steps {
                script {
                    dir(POM_LOC) {
                        if (isUnix()) {
                            sh 'mvn test'
                        } else {
                            bat 'mvn test'
                        }
                    }
                }
            }
        }

        stage('Integration Testing') {
            steps {
                script {
                    dir(POM_LOC) {
                        if (isUnix()) {
                            sh 'mvn verify -DskipTests'
                        } else {
                            bat 'mvn verify -DskipTests'
                        }
                    }
                }
            }
        }
    }
}
```

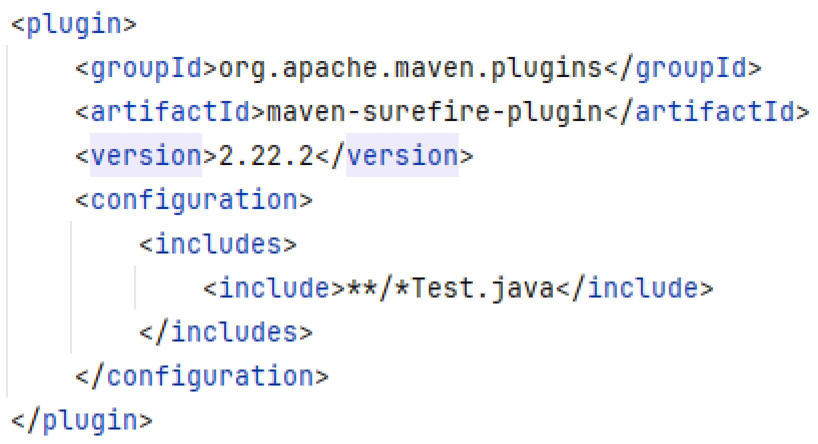
Parallel Execution:

- ***parallel* { ... }** runs the Unit Testing and Integration Testing stages simultaneously, which reduces the total time required for testing.

Unit Testing Stage:

Command: ***mvn test***

This command runs unit tests, which focus on testing individual components or methods in isolation. Unit tests are executed by leveraging Maven's test lifecycle phase, specifically configured through the ***maven-surefire-plugin***. This setup restricts execution to classes that follow the convention of ending in ***Test.java**. By standardizing this naming convention within the plugin configuration, we ensure that only unit test classes are selected, optimizing test scope and reducing unintended executions of other test types. This approach not only streamlines the testing process but also maintains consistency and control over which tests are included in the pipeline.

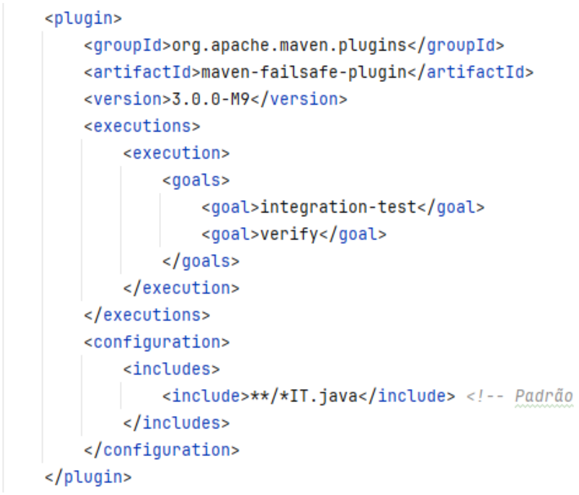


```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <configuration>
        <includes>
            <include>**/*Test.java</include>
        </includes>
    </configuration>
</plugin>
```

Integration Testing Stage:

Command: ***mvn verify -DskipTests***

Here, ***mvn* verify** executes all integration tests, validating the interaction between different parts of the system. The ***-DskipTests*** flag here refers to skipping unit tests, which ensures only integration tests run in this stage. For integration tests ***maven-failsafe-plugin*** was used to manage and run integration tests within the Maven build lifecycle. The plugin is set up to include and execute only tests that follow the naming pattern ****IT.java***, which designates them as Integration Tests.

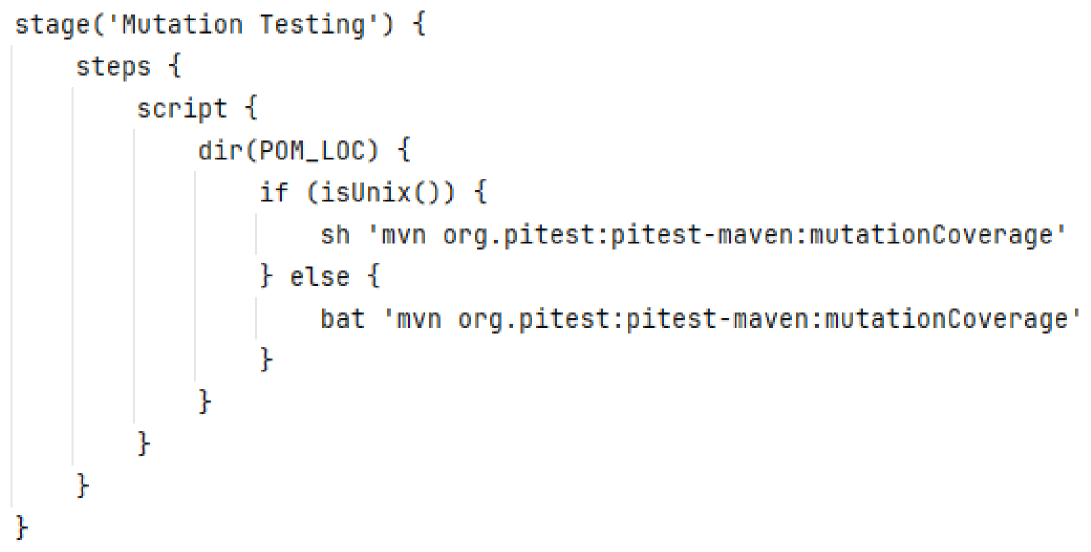


```xml
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>3.0.0-M9</version>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <includes>
            <include>**/*IT.java</include> <!-- Padrão
        </includes>
    </configuration>
</plugin>
```

Only files matching the pattern *IT.java will be included, ensuring that only integration tests are executed by the maven-failsafe-plugin. This separation of unit and integration tests helps keep the test suite organized and manageable.

4. Mutation Testing

The '***mvn org.pitest:pitest-maven:mutationCoverage'*** command in the pipeline is responsible for executing mutation tests using the PIT (PITest) framework. Mutation testing evaluates the quality of your unit tests by introducing small code changes (mutations) and checking if the existing tests can detect these changes. This ensures your tests are comprehensive and robust.



```
stage('Mutation Testing') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'mvn org.pitest:pitest-maven:mutationCoverage'
                } else {
                    bat 'mvn org.pitest:pitest-maven:mutationCoverage'
                }
            }
        }
    }
}
```

The PIT Maven Plugin is configured in pom.xml as follows:

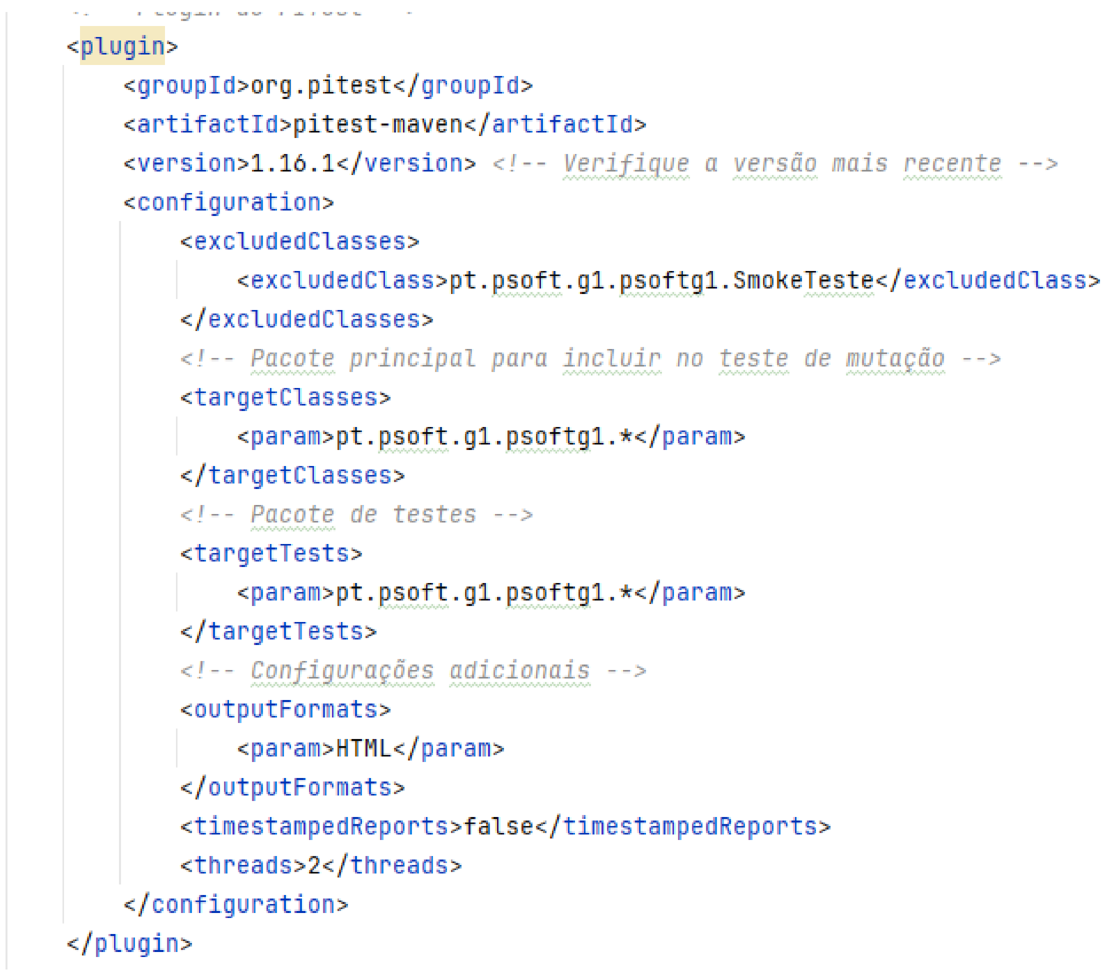
Scope: Only the target classes within pt.psoft.g1.psoftg1.* are considered for mutation testing, ensuring focused analysis.

Exclusions: Specific classes like SmokeTeste are excluded, typically to bypass utility or setup classes that aren't suitable for mutation.

Output: Reports are generated in HTML format, making it easier to visualize mutation coverage.

Concurrency: The test execution is optimized by running with two threads, striking a balance between performance and resource usage. We set the thread count to two based on the resources of the given virtual machine. With only one CPU available, using a maximum of two threads ensures efficient resource management without overloading the system.

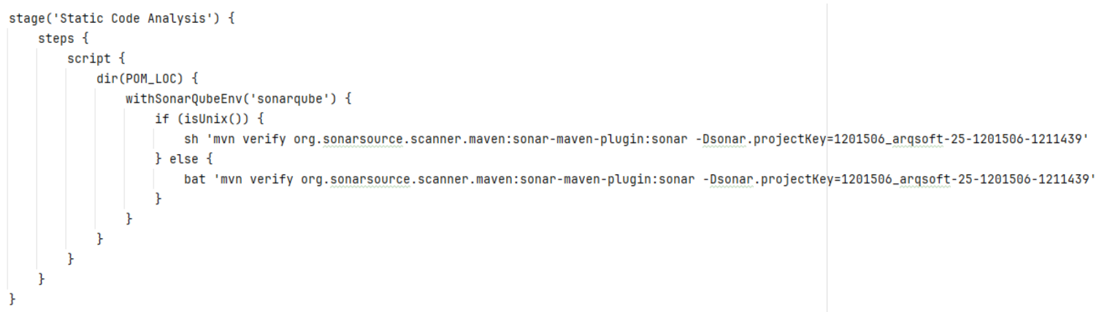
Only files and methods included in the specified packages are mutated, and the reports generated provide insight into which tests are effective in catching defects and which areas may require more thorough testing.



```xml
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.16.1</version> <!-- Verifique a versão mais recente -->
    <configuration>
        <excludedClasses>
            <excludedClass>pt.psoft.g1.psoftg1.SmokeTeste</excludedClass>
        </excludedClasses>
        <!-- Pacote principal para incluir no teste de mutação -->
        <targetClasses>
            <param>pt.psoft.g1.psoftg1.*</param>
        </targetClasses>
        <!-- Pacote de testes -->
        <targetTests>
            <param>pt.psoft.g1.psoftg1.*</param>
        </targetTests>
        <!-- Configurações adicionais -->
        <outputFormats>
            <param>HTML</param>
        </outputFormats>
        <timestampedReports>false</timestampedReports>
        <threads>2</threads>
    </configuration>
</plugin>
```

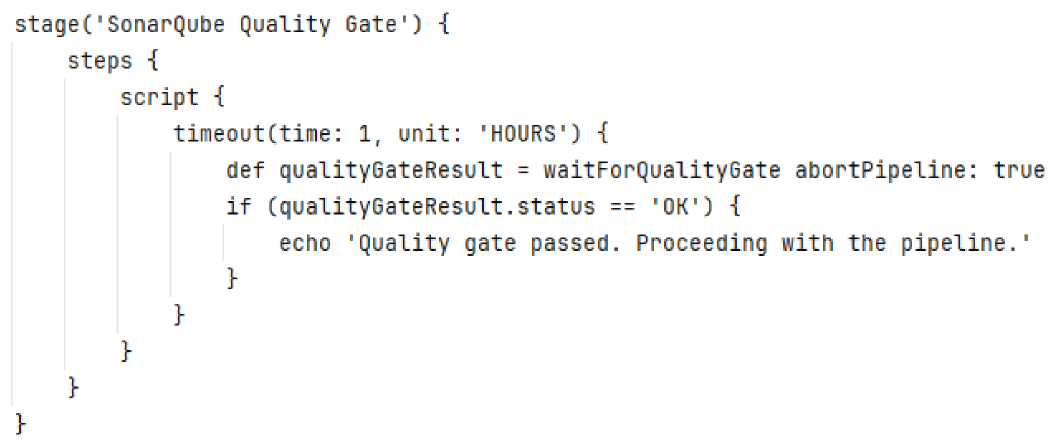
5. SonarQube Static Code Analysis

In this section, SonarQube is used to perform static code analysis and ensure code quality standards.Static Code Analysis:



```groovy
stage('Static Code Analysis') {
    steps {
        script {
            dir(POM_LOC) {
                withSonarQubeEnv('sonarqube') {
                    if (isUnix()) {
                        sh 'mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=1201506_arqsoft-25-1201506-1211439'
                    } else {
                        bat 'mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=1201506_arqsoft-25-1201506-1211439'
                    }
                }
            }
        }
    }
}
```

The Static Code Analysis stage leverages SonarQube's Maven plugin to scan the codebase for potential bugs, vulnerabilities, and code smells. The ***withSonarQubeEnv*** block configures the environment for SonarQube, using the project key ***1201506_arqsoft-25-1201506-1211439*** to uniquely identify the project in SonarQube Cloud.



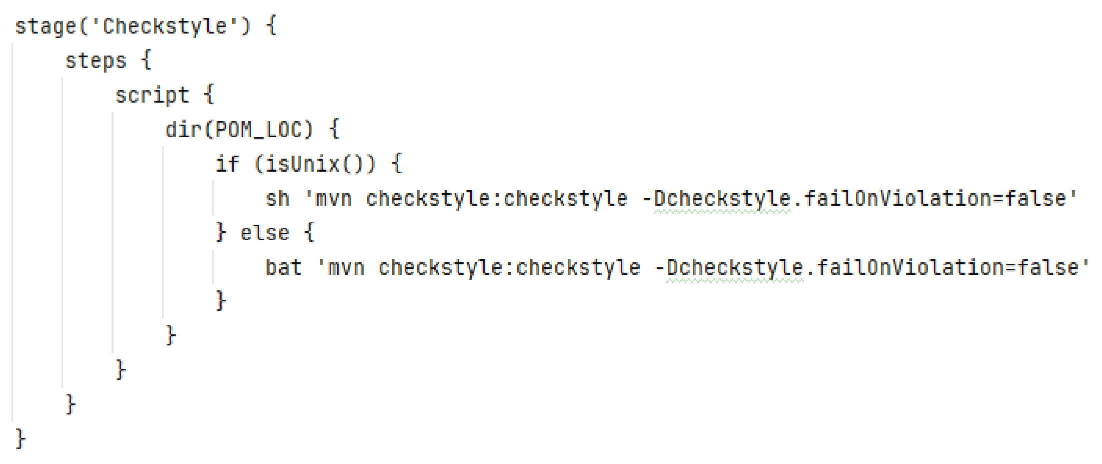
```
stage('SonarQube Quality Gate') {
    steps {
        script {
            timeout(time: 1, unit: 'HOURS') {
                def qualityGateResult = waitForQualityGate abortPipeline: true
                if (qualityGateResult.status == 'OK') {
                    echo 'Quality gate passed. Proceeding with the pipeline.'
                }
            }
        }
    }
}
```

SonarQube Quality Gate: After analysis, the Quality Gate stage checks if the code meets the predefined quality standards (such as thresholds for code coverage or code smells). The pipeline waits up to one hour for the quality gate result, and if the quality gate passes ***(status 'OK')***, the pipeline continues; otherwise, it will abort. This enforces code quality early in the CI/CD process.

Our performance tests aimed at comparing the efficiency of SonarQube with the combination of *PMD*, *Checkstyle*, and *SpotBugs* yielded inconclusive results regarding the selection of these tools. While SonarQube offers a unified platform for comprehensive code analysis, it may require a longer execution time. However, its effectiveness in identifying and addressing issues can ultimately lead to significant time savings during the resolution process. This advantage underscores the importance of considering both execution time and issue resolution efficiency when evaluating static code analysis tools. Since the team was in the first iteration where we took a brownfield project, we knew that issues that we were not aware of arose. We pick SonarQube for an initial phase to help us and streamline the issue more effectively, but the other tools will be considered in further iterations.

6. Checkstyle Analysis

The Checkstyle stage in the Jenkins pipeline uses the Maven Checkstyle plugin to ensure adherence to coding standards. The command '***mvn checkstyle:checkstyle***' runs a static code analysis, with the flag ***'-Dcheckstyle.failOnViolation=false'*** configured to prevent build failure even if violations are found. This setup allows for the reporting of code style issues without interrupting the build process.



```
stage('Checkstyle') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'mvn checkstyle:checkstyle -Dcheckstyle.failOnViolation=false'
                } else {
                    bat 'mvn checkstyle:checkstyle -Dcheckstyle.failOnViolation=false'
                }
            }
        }
    }
}
```

7. **Code Coverage Analysis**

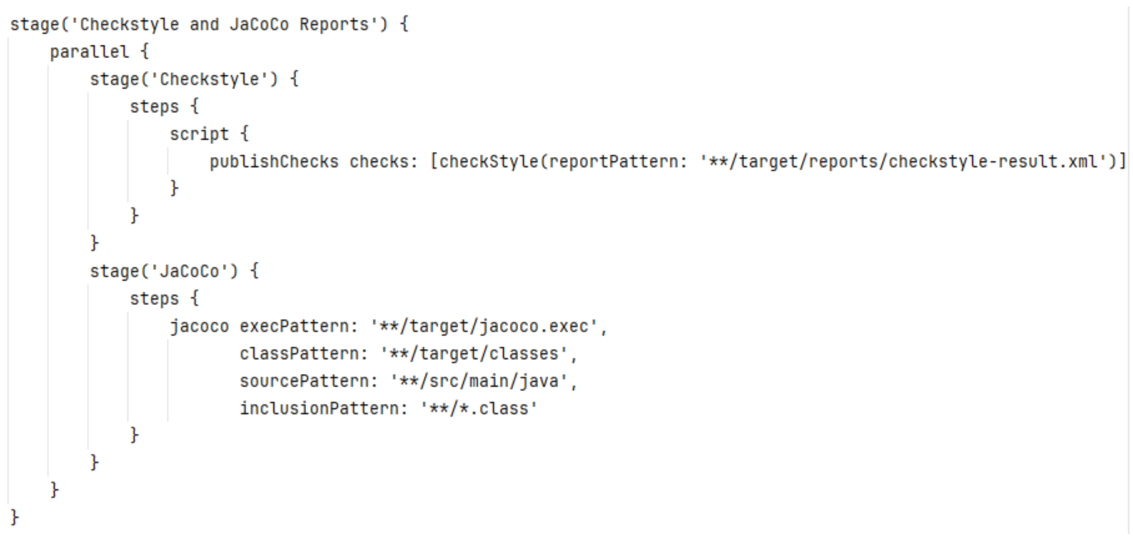
The Checkstyle and JaCoCo Reports stage in our Jenkins pipeline is designed to ensure both code quality and test coverage metrics are properly captured. It runs in parallel to improve pipeline performance by executing Checkstyle and JaCoCo reports simultaneously.

Checkstyle Report

The results are collected in the checkstyle-result.xml file, which is then published using the publishChecks step, providing visibility into the code style compliance.

JaCoCo Report

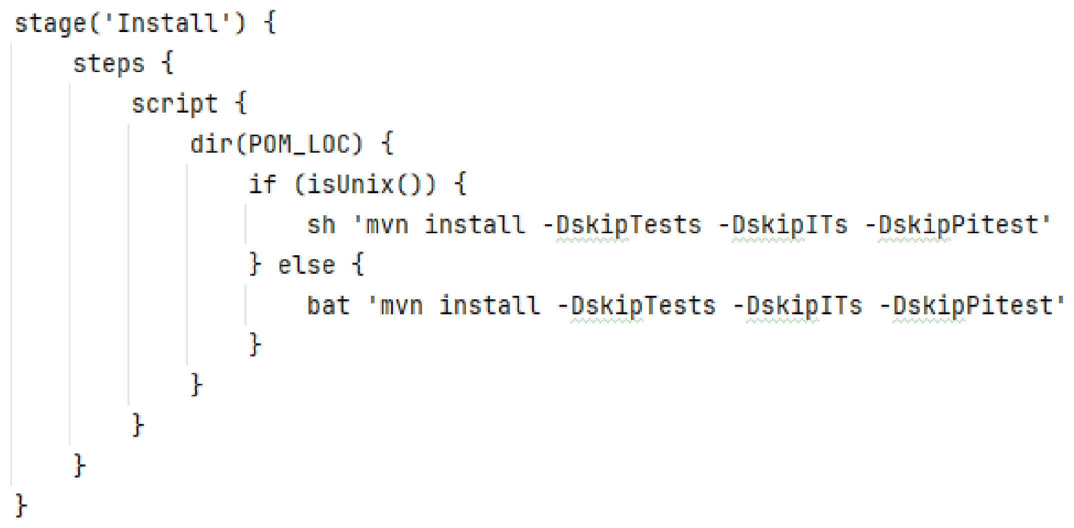
The JaCoCo stage generates a code coverage report based on test execution. It processes the jacoco.exec file created during the testing phase, detailing the percentage of code covered by tests. The configuration includes patterns for **execPattern**, **classPattern**, and **sourcePattern** to correctly locate the generated coverage data, compiled classes, and source files. This report helps the team identify areas of code that are untested and improve overall test coverage.



```
stage('Checkstyle and JaCoCo Reports') {
    parallel {
        stage('Checkstyle') {
            steps {
                script {
                    publishChecks checks: [checkStyle(reportPattern: '**/target/reports/checkstyle-result.xml')]
                }
            }
        }
        stage('JaCoCo') {
            steps {
                jacoco execPattern: '**/target/jacoco.exec',
                       classPattern: '**/target/classes',
                       sourcePattern: '**/src/main/java',
                       inclusionPattern: '**/*.class'
            }
        }
    }
}
```

8. **Installation stage**

In the stage('Install') of the Jenkins pipeline, the team executes the Maven installation process for the Java project. This stage is dedicated to building and installing the project's artifacts. The pipeline first changes the working directory to the location specified by the POM_LOC variable, which typically contains the pom.xml file needed for Maven commands.

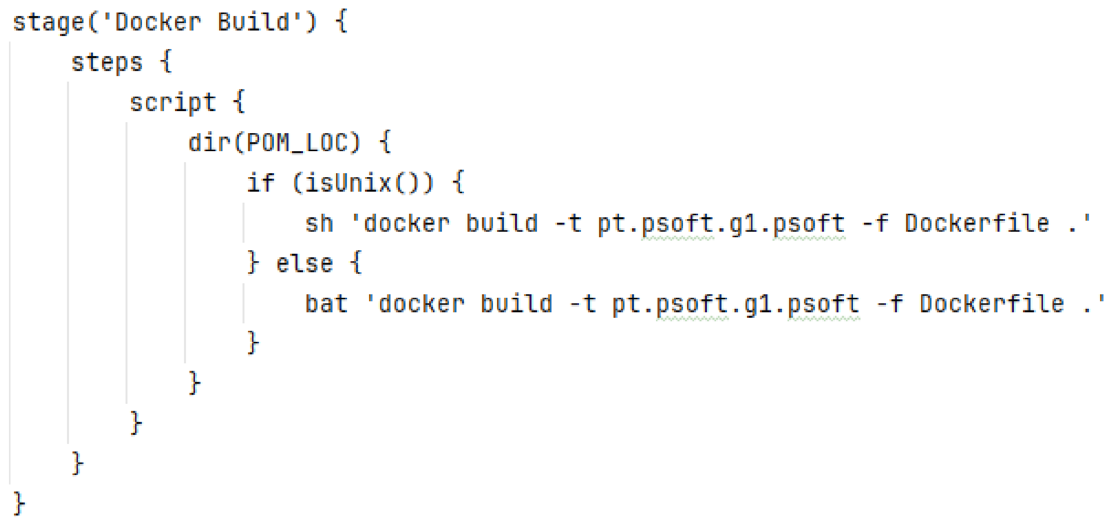


```
stage('Install') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'mvn install -DskipTests -DskipITs -DskipPitest'
                } else {
                    bat 'mvn install -DskipTests -DskipITs -DskipPitest'
                }
            }
        }
    }
}
```

The Maven command performs the installation phase, compiling the code and packaging the application while skipping unit tests, integration tests, and mutation tests. This approach accelerates the build process, which is particularly beneficial in continuous integration environments where quick feedback is essential.

9. Docker Build

In the ***stage('Docker Build')*** of the Jenkins pipeline, the team initiates the process of building a Docker image for the application. The stage begins by changing the working directory to the location specified by the POM_LOC variable, which typically contains the pom.xml file relevant to the Maven project.

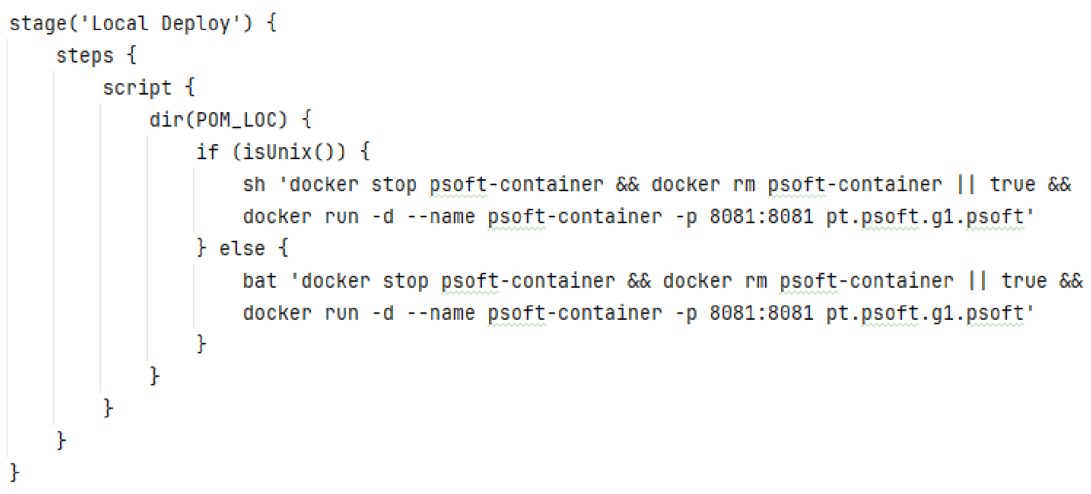


```
stage('Docker Build') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'docker build -t pt.psoft.g1.psoft -f Dockerfile .'
                } else {
                    bat 'docker build -t pt.psoft.g1.psoft -f Dockerfile .'
                }
            }
        }
    }
}
```

The '***docker build'*** command is responsible for creating a Docker image from the specified Dockerfile. The '***-t flag'*** tags the image with the name '***pt.psoft.g1.psoft'***, making it easier to reference in subsequent stages or deployments.

10. Local Deploy

In the '***stage('Local Deploy')'***, the team implements the steps necessary to deploy the Docker container for the application locally, prepared by the previous step. T



```
stage('Local Deploy') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'docker stop psoft-container && docker rm psoft-container || true &&
                    docker run -d --name psoft-container -p 8081:8081 pt.psoft.g1.psoft'
                } else {
                    bat 'docker stop psoft-container && docker rm psoft-container || true &&
                    docker run -d --name psoft-container -p 8081:8081 pt.psoft.g1.psoft'
                }
            }
        }
    }
}
```

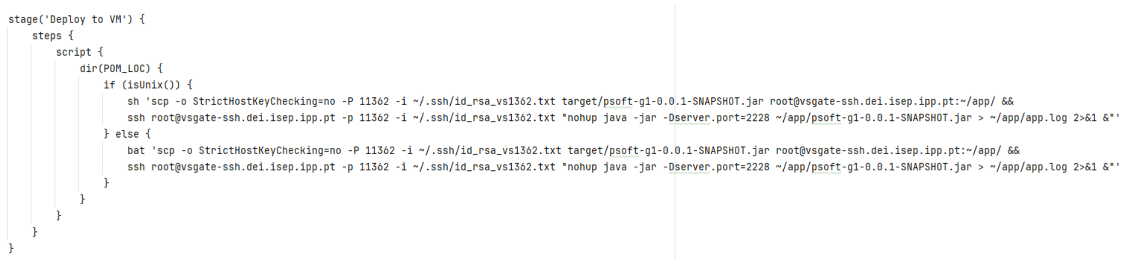
This stage is crucial for testing the deployed application in a local environment after it has been built. It ensures that the latest changes are reflected and that the team can verify the application's functionality before pushing it to production or further environments. The use of Docker for deployment also aligns with modern practices of containerization, which promotes consistency and scalability across development and production environments.

11. Deployment to the Virtual Machine

The '***stage('Deploy to VM'')*** in the Jenkins pipeline is responsible for deploying the built application to a virtual machine (VM).

First, it navigates to the specified directory containing the Maven project. Then, it uses the '**scp'** command to securely copy the built **JAR** file (psoft-g1-0.0.1-SNAPSHOT.jar) from the local target directory to the remote VM at vsgate-ssh.dei.isep.ipp.pt. The deployment uses the '***SSH key located at ~/.ssh/id_rsa_vs1362.txt'*** for authentication and connects through port 11362.

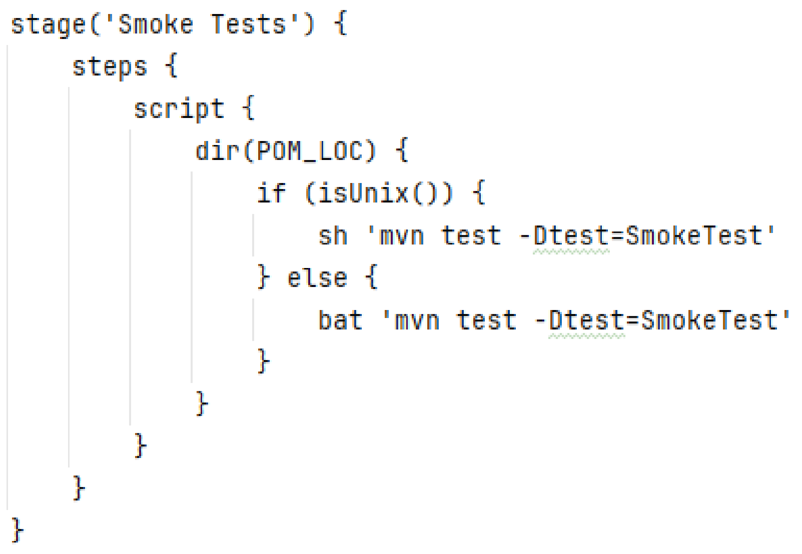
After copying the JAR file, it establishes an SSH connection to the VM and runs the application using Java command with 'nohup', allowing it to run in the background on port 2228. The output and any errors are logged to ***'~/app/app.log'***. This stage ensures the latest application version is deployed and running on the VM.



```
stage('Deploy to VM') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'scp -o StrictHostKeyChecking=no -P 11362 -i ~/.ssh/id_rsa_vs1362.txt target/psoft-g1-0.0.1-SNAPSHOT.jar root@vsgate-ssh.dei.isep.ipp.pt:~/app/ &&
                    ssh root@vsgate-ssh.dei.isep.ipp.pt -p 11362 -i ~/.ssh/id_rsa_vs1362.txt "nohup java -jar -Dserver.port=2228 ~/app/psoft-g1-0.0.1-SNAPSHOT.jar > ~/app/app.log 2>&1 &"'
                } else {
                    bat 'scp -o StrictHostKeyChecking=no -P 11362 -i ~/.ssh/id_rsa_vs1362.txt target/psoft-g1-0.0.1-SNAPSHOT.jar root@vsgate-ssh.dei.isep.ipp.pt:~/app/ &&
                    ssh root@vsgate-ssh.dei.isep.ipp.pt -p 11362 -i ~/.ssh/id_rsa_vs1362.txt "nohup java -jar -Dserver.port=2228 ~/app/psoft-g1-0.0.1-SNAPSHOT.jar > ~/app/app.log 2>&1 &"'
                }
            }
        }
    }
}
```

12. Smoke Tests

The ***stage('Smoke Tests')*** is designed to execute smoke tests on the application after it has been deployed to the virtual machine. This stage is particularly important as it should listen on the specified URL (https://vs-gate.dei.isep.ipp.pt:31362).

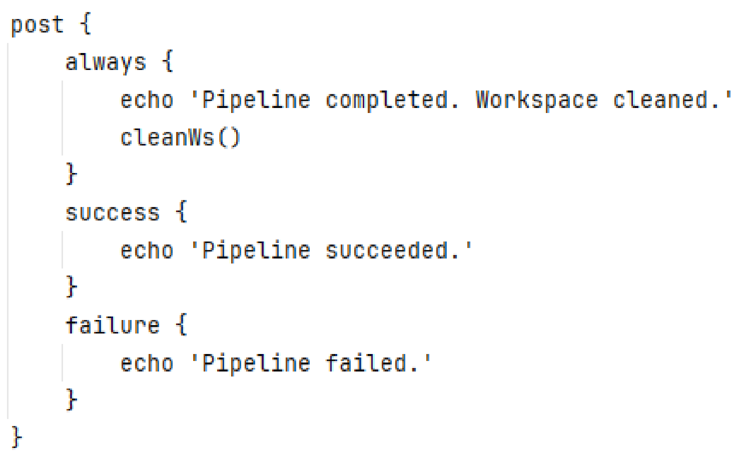


```
stage('Smoke Tests') {
    steps {
        script {
            dir(POM_LOC) {
                if (isUnix()) {
                    sh 'mvn test -Dtest=SmokeTest'
                } else {
                    bat 'mvn test -Dtest=SmokeTest'
                }
            }
        }
    }
}
```

It runs the Maven command to execute the smoke tests, specifically targeting a test class named ‘***SmokeTest’***. The command ‘***mvn test -Dtest=SmokeTest’*** instructs Maven to execute only the tests defined in the SmokeTest class, which typically contains basic tests to verify that the application starts up correctly and essential functionalities work as expected, saving also time on the ‘***mvn test***’ command execution.

This stage is crucial for quickly validating that the deployment was successful and that the application is functioning at a basic level, ensuring that critical paths are operational before further usage.

13. Post



```
post {
    always {
        echo 'Pipeline completed. Workspace cleaned.'
        cleanWs()
    }
    success {
        echo 'Pipeline succeeded.'
    }
    failure {
        echo 'Pipeline failed.'
    }
}
```

The ‘***post’*** section of the Jenkins pipeline defines actions to be taken after the pipeline execution is completed, regardless of the outcome. The always block executes cleanup operations and logs a message indicating that the pipeline has completed and the workspace is being cleaned. The success block logs a message if the pipeline succeeds, while the failure block logs a message if the pipeline fails. This structure ensures that appropriate messages are displayed and resources are managed effectively after the pipeline runs.

Optimizations

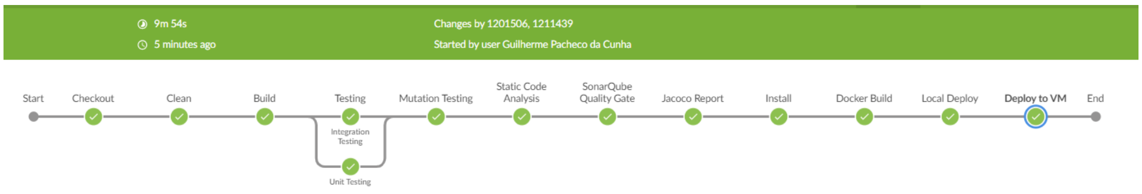
The pipeline optimization was achieved through three main approaches: skipping unnecessary tests in certain stages, parallelizing unit and integration tests, and utilizing four threads for mutation testing. Please note that the results we obtained in local stages since the DEI servers’ performance can fluctuate according to the affluency and availability. Let’s elaborate on each of these strategies:

1. Skipping Unnecessary Tests

The skipping strategy involves disabling tests that are not relevant for a specific stage of the pipeline. This is particularly useful in scenarios where running all tests may be unnecessary or redundant. For instance, when building a new feature, one might choose not to execute integration tests that depend on components that have not been changed. This approach reduces the overall execution time of the pipeline, allowing developers to receive quicker feedback on the code status. Additionally, it avoids running tests that have already been validated in previous runs, saving resources and time.

2. Parallelization of Unit and Integration Tests

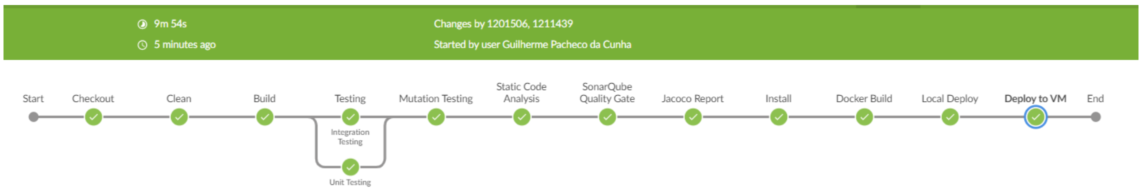
Parallelizing tests is an effective technique for increasing pipeline efficiency. By allowing unit and integration tests to be executed simultaneously rather than sequentially, the total execution time of the pipeline is significantly reduced. This is especially beneficial in large projects, where the number of tests can be substantial. Parallelization also makes better use of available resources, such as multi-core CPUs, maximizing hardware utilization. This results in faster feedback regarding code quality, enabling the team to identify and address issues more swiftly.


9m 54s
5 minutes ago
Changes by 1201506, 1211439
Started by user Guilherme Pacheco da Cunha

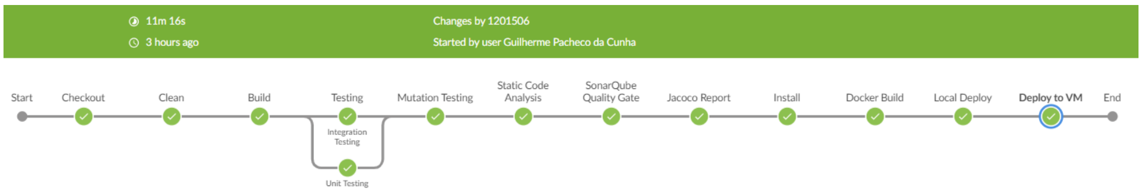
3. Utilizing Four Threads for Mutation Testing

Configuring the pipeline to utilize four threads during mutation testing is a strategy that complements parallelization. By allowing multiple mutation tests to be executed in parallel across different threads, the team can further reduce the execution time. This approach is particularly beneficial in projects with many mutation tests, as it maximizes execution efficiency and fully leverages available hardware resources. By distributing the workload across multiple threads, the team can obtain results more rapidly, allowing for a quicker feedback cycle.

With multiple threads (4):


9m 54s
5 minutes ago
Changes by 1201506, 1211439
Started by user Guilherme Pacheco da Cunha

With a single thread:


11m 16s
3 hours ago
Changes by 1201506
Started by user Guilherme Pacheco da Cunha

The use of a single thread resulted in a pipeline duration of 11 minutes and 16 seconds, while employing four threads reduced the time to only 9 minutes and 54 seconds. Although this improvement may seem minor, it translates to a time reduction of approximately

12.13%. In a production environment, even slight optimizations can yield significant cost savings, potentially amounting to millions. Therefore, it is essential to keep pipeline runtimes as low as possible, making such enhancements a best practice for efficient development and deployment processes.

Conclusion

Implementing these optimizations results in a more agile and responsive pipeline that not only reduces execution time but also improves the effectiveness of problem detection and resolution. The use of skips for unnecessary tests, the parallelization of tests, and the employment of multiple threads for mutation testing contribute to a more efficient development process, promoting a continuous development cycle and higher quality in delivered code. These improvements are crucial in an agile development environment, where speed and quality are essential for project success.

Possible improvements on next **iteration:**

For the next iteration, we should explore further enhancements to parallelism by identifying additional stages that can be parallelized to optimize runtime. Additionally, researching alternative code analysis tools could provide insights into more efficient solutions. Finally, a thorough review of all pipeline steps will help determine if any are redundant or unnecessary, streamlining the overall process.