



TUS

**Technological University of the Shannon:
Midlands Midwest**

Ollscoil Teicneolaíochta na Sionainne:
Lár Tíre Iarthar Láir

www.tus.ie

A Comprehensive Project Report on "Implementation of a CI/CD Pipeline for the Student Management API Project"

Presented by: Atharva Dinesh Pailwan

Course Name: MSc. in Software Design with Cloud Native Computing

Subject: Continuous Build and Delivery

Professor: Dr. Mary Giblin, Dr. Amit Hirway

Student ID: A00325723

Email: a00325723@student.tus.ie

TABLE OF CONTENTS

1. Introduction.....	3
1.1 Student Management API.....	3
1.2 Assignment Objectives.....	3
2. Project Architecture and Design.....	4
2.1 Key Components.....	4
2.2 CI/CD Pipeline Overview.....	4
2.2.1 Code Checkout.....	4
2.2.2 Build Execution.....	5
2.2.3 Testing.....	5
2.2.4 Static Code Analysis.....	6
2.2.5 Containerization.....	6
2.2.6 Deployment via Ansible.....	7
3. Testing and Coverage.....	8
3.1 Code Coverage Results.....	8
4. Static Code Analysis using SonarQube.....	10
5. Challenges and Improvements.....	12
6. Screenshots.....	13
7. Conclusion and Evaluation.....	22

1. Introduction

1.1 Student Management API

The Student Management API is a Java-based Spring Boot microservice designed to manage student records via a RESTful interface. This project was developed to demonstrate a robust application of Continuous Integration and Continuous Delivery (CI/CD) methodologies, focusing on automation in build, testing, and deployment within a microservice architecture..

1.2 Assignment Objectives:

This project aims to deliver a fully functional API with an accompanying automated CI/CD pipeline, fulfilling the following key requirements:

- **Automated Builds and Version Control:** Implementation of automated builds using Jenkins, with code housed in a Git repository.
- **Static Code Analysis:** Integration of SonarQube to ensure code quality across builds.
- **Test Automation and Coverage:** Comprehensive testing using JUnit and Mockito, with coverage analysis via JaCoCo.
- **Dockerization and Automated Deployment:** Utilization of Docker for containerization and Ansible for automated deployment.
- **Evaluation of Pipeline Efficiency:** Analysis and discussion of the pipeline's performance, including challenges faced and potential improvements.

This report details the successful implementation of the Student Management API, outlining the architectural choices, technology stack, and key components of the CI/CD pipeline. It concludes with a reflection on the pipeline's effectiveness and lessons learned throughout the project lifecycle.

2. Project Architecture and Design

The Student Management API was architecturally designed as a monolithic Spring Boot application, adhering to a clean, layered architecture consisting of controllers, services, repositories, and domain models. The project leveraged a range of technologies including Java 17, Spring Boot 3, Maven, JUnit, Mockito, Docker, and Ansible, among others.

2.1 Key Components:

- **Controllers:** Managed HTTP request routing and response handling.
- **Services:** Encapsulated the business logic of the application.
- **Repositories:** Provided data access mechanisms, interfacing with the H2 database.
- **Domain Models:** Represented the data structure within the application.

The entire application was packaged using Maven, facilitating dependency management and build automation.

2.2 CI/CD Pipeline Overview

The CI/CD pipeline for the Student Management API, orchestrated through Jenkins, was meticulously designed to automate each step from code integration to deployment seamlessly. This comprehensive automation ensured consistent and efficient development practices, minimizing human error and maximizing reproducibility. Below are the expanded details of each key stage in the pipeline:



2.2.1 Code Checkout

The first stage in the pipeline involved the automated checkout of the latest code from the Git repository. This step was crucial as it ensured that the pipeline always used the most current version of the source code for every build. Jenkins was configured to watch for any commits to the repository; upon detecting a change, it automatically fetched the updated code. This setup supported a trigger-based mechanism where new commits prompted immediate pipeline execution, thereby integrating changes continuously.

Stage 'Checkout SCM'

🕒 Started 56 min ago

⌚ Queued 0 ms

⌚ Took 0.44 sec

✅ Success

🖥️ Running on [Jenkins](#)

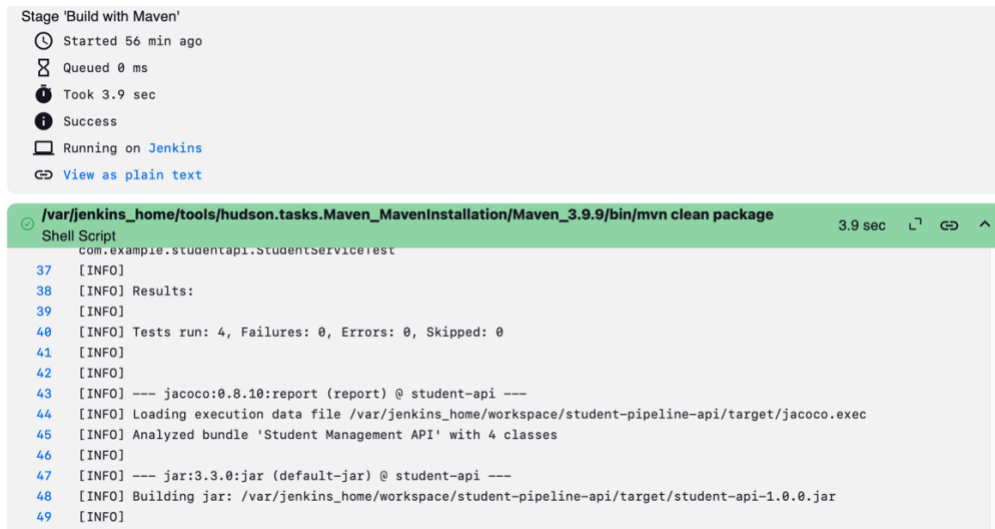
🔗 [View as plain text](#)

✅ Check out from version control

0.43 sec 🔍 ⌂ ⌵

2.2.2 Build Execution

Following the code checkout, the pipeline executed the build process using Maven, the chosen build automation tool. Maven compiled the source code and packaged the application into an executable JAR file. This process included resolving and downloading dependencies defined in the pom.xml file, ensuring that the build environment was self-contained and consistent across different development and production environments. The automation of this step eliminated the manual build process, reducing the potential for human error and ensuring that the build outputs were always consistent.



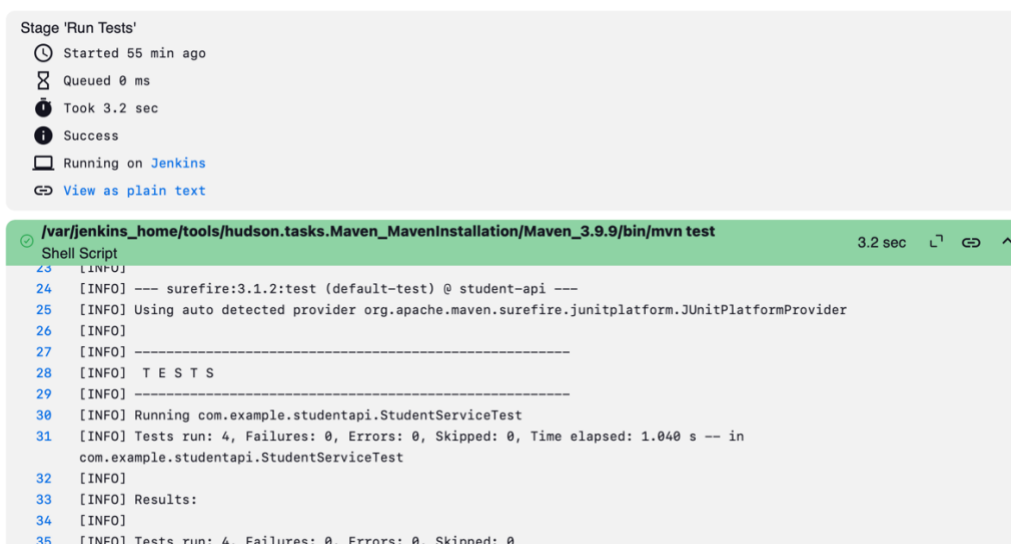
The screenshot shows the Jenkins build interface for a stage named 'Build with Maven'. The stage status is 'Success', having taken 3.9 seconds. Below the stage summary, the build log is displayed, showing the execution of 'mvn clean package'. The log output includes Maven version information, a successful compilation, and the creation of a JAR file named 'student-api-1.0.0.jar'.

```
Stage 'Build with Maven'
Started 56 min ago
Queued 0 ms
Took 3.9 sec
Success
Running on Jenkins
View as plain text

/var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/Maven_3.9.9/bin/mvn clean package 3.9 sec
Shell Script
37 [INFO] com.example.studentapi.StudentServiceTest
38 [INFO] Results:
39 [INFO]
40 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
41 [INFO]
42 [INFO]
43 [INFO] --- jacoco:0.8.10:report (report) @ student-api ---
44 [INFO] Loading execution data file /var/jenkins_home/workspace/student-pipeline-api/target/jacoco.exec
45 [INFO] Analyzed bundle 'Student Management API' with 4 classes
46 [INFO]
47 [INFO] --- jar:3.3.0:jar (default-jar) @ student-api ---
48 [INFO] Building jar: /var/jenkins_home/workspace/student-pipeline-api/target/student-api-1.0.0.jar
49 [INFO]
```

2.2.3 Testing

After the build, the pipeline focused on testing the application. This stage was critical for verifying the correctness of the application code before it moved further along in the pipeline. A comprehensive suite of unit tests, written using JUnit, was executed automatically. These tests were designed to cover various components of the application, particularly focusing on the business logic encapsulated in the service layer. The use of Mockito facilitated the mocking of dependencies, allowing for isolated testing of specific functionality without the need for actual database connections or external systems. Jenkins collected and reported the results of these tests, providing immediate feedback on their success or failure. This immediate feedback was vital for early detection of issues, enabling quick fixes that maintained the overall health of the application codebase.



The screenshot shows the Jenkins build interface for a stage named 'Run Tests'. The stage status is 'Success', having taken 3.2 seconds. Below the stage summary, the build log is displayed, showing the execution of 'mvn test'. The log output includes Maven version information, the use of the JUnit platform, and the successful execution of unit tests for 'StudentServiceTest'.

```
Stage 'Run Tests'
Started 55 min ago
Queued 0 ms
Took 3.2 sec
Success
Running on Jenkins
View as plain text

/var/jenkins_home/tools/hudson.tasks.Maven_MavenInstallation/Maven_3.9.9/bin/mvn test 3.2 sec
Shell Script
23 [INFO]
24 [INFO] --- surefire:3.1.2:test (default-test) @ student-api ---
25 [INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider
26 [INFO]
27 [INFO] -----
28 [INFO] T E S T S
29 [INFO] -----
30 [INFO] Running com.example.studentapi.StudentServiceTest
31 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.040 s -- in
com.example.studentapi.StudentServiceTest
32 [INFO]
33 [INFO] Results:
34 [INFO]
35 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

2.2.4 Static Code Analysis

Static code analysis was conducted using SonarQube, integrated into the pipeline following the testing stage. This tool analyzed the source code to identify potential bugs, vulnerabilities, code smells, and to ensure adherence to defined coding standards. SonarQube's analysis provided a detailed report on various quality metrics, including test coverage, complexity, duplicated code, and coding standard violations. The integration of SonarQube into the CI/CD pipeline ensured that code quality was assessed continuously, allowing developers to address issues promptly before merging changes into the main branch. This continuous quality control is essential in maintaining high standards throughout the project's lifecycle.

```
Stage 'Code Quality - SonarQube'
🕒 Started 55 min ago
⌚ Queued 0 ms
🕒 Took 6 sec
✅ Success
🖥 Running on Jenkins
🔗 View as plain text

Maven 3.9.9
Use a tool from a predefined Tool Installation 13 ms

mvn sonar:sonar -Dsonar.projectKey=com.example:student-api -Dsonar.host.url=http://107.21.51.131:9000 -... 5.6 sec
Shell Script
100 [INFO] 17:46:33.833 Sensor Zero Coverage Sensor
101 [INFO] 17:46:33.834 Sensor Zero Coverage Sensor (done) | time=0ms
102 [INFO] 17:46:33.834 Sensor Java CPD Block Indexer
103 [INFO] 17:46:33.840 Sensor Java CPD Block Indexer (done) | time=6ms
104 [INFO] 17:46:33.840 ----- Gather SCA dependencies on project
105 [INFO] 17:46:33.844 CPD Executor 2 files had no CPD blocks
106 [INFO] 17:46:33.844 CPD Executor Calculating CPD for 3 files
107 [INFO] 17:46:33.848 CPD Executor CPD calculation finished (done) | time=4ms
108 [INFO] 17:46:33.850 SCM revision ID '582a952e308fe91f943358edc2d52e268d3f136f'
109 [INFO] 17:46:33.891 Analysis report generated in 39ms, dir size=253.8 kB
110 [INFO] 17:46:33.901 Analysis report compressed in 9ms, zip size=39.0 kB
111 [INFO] 17:46:34.117 Analysis report uploaded in 214ms
```

2.2.5 Containerization

Docker played a pivotal role in the containerization stage of the pipeline. Once the application had been built and tested, a Dockerfile was used to build a Docker image containing the application and its runtime environment. This Docker image encapsulated everything needed to run the application, ensuring that it could be deployed consistently in any environment that supports Docker. This stage was crucial for abstracting the application from the underlying infrastructure, facilitating portability, and ensuring that the application ran identically in development, testing, and production environments.

```
Stage 'Build Docker Image'
🕒 Started 55 min ago
⌚ Queued 0 ms
🕒 Took 2.5 sec
✅ Success
🖥 Running on Jenkins
🔗 View as plain text

docker build -t atharvapaiwan7/student-api-cicd . 2.4 sec
Shell Script
22 #6 DONE 0.0s
23
24 #6 [internal] load build context
25 #6 sha256:1f1c16eadd305fa06a6f06bc748f2b915041ffd9e621332368ce58685bf41bd
26 #6 transferring context: 46.87MB 0.4s done
27 #6 DONE 0.4s
28
29 #7 [2/3] WORKDIR /app
30 #7 sha256:9afa9ac1386f35c5b20f0916db6a83db74c75492be10e460b6c34286fac8fb0e
31 #7 CACHED
32
33 #5 [3/3] COPY target/student-api-1.0.0.jar app.jar
34 #5 sha256:b3504064d1c6ed4153843768146b1197527cf831c5f33f6556471e0838926bc4
35 #5 DONE 0.1s
```

2.2.6 Deployment via Ansible

The final stage of the pipeline involved the deployment of the application, which was automated using Ansible. Ansible scripts were configured to deploy the Docker container to specified server environments. This automation included tasks such as pulling the latest Docker image from a registry, stopping any currently running containers, and starting the new container. This approach not only streamlined the deployment process but also minimized downtime and ensured a smooth transition between application versions. By automating deployment through Ansible, the project achieved a repeatable and error-free deployment process, crucial for maintaining the reliability and availability of the application in production.

Stage 'Deploy via Ansible'

🕒 Started 55 min ago

⌚ Queued 0 ms

🕒 Took 1.6 sec

🟢 Success

🖥️ Running on [Jenkins](#)

🔗 [View as plain text](#)

🟢 **ansible-playbook -i localhost, -c local deploy.yml --become --extra-vars "ansible_become_password=\$BECO...** 1.5 sec 🔍 ↻

Shell Script

```
0 + ansible-playbook -i localhost, -c local deploy.yml --become --extra-vars ansible_become_password=****
1
2 PLAY [Deploy Student Management API locally with Docker] *****
3
4 TASK [Gathering Facts] *****
5 ok: [localhost]
6
7 TASK [Check if the OS is macOS] *****
8 ok: [localhost]
9
10 TASK [Skip Docker installation on macOS] *****
11 skipping: [localhost]
12
13 TASK [Pull Docker image] *****
```

3. Testing and Coverage

Testing Approach: The project's testing strategy focused primarily on unit testing the service layer. A JUnit 5 test class, `StudentServiceTest`, was written to exercise the business logic in `StudentService`. Using Mockito, the `StudentRepository` was mocked so that the service methods could be tested in isolation without needing a real database. Each method of `StudentService` has a corresponding unit test:

- `testFindAllStudents()` creates a list of dummy `Student` objects, configures the repository mock to return this list for `findAll()`, and then asserts that the service returns the expected list and size.
- `testFindById_StudentExists()` simulates a repository `findById` returning a `Optional<Student>` for a given ID and verifies that the service correctly returns the student when present.
- `testSaveStudent()` simulates saving a student and ensures the returned object is not null and has the expected field values.
- `testDeleteById()` uses Mockito's `verify` to ensure `repository.deleteById()` was called when the service's delete method is invoked.

3.1 Code Coverage Results

Code coverage was measured using JaCoCo, which produces a report on how much of the code was executed by the tests. The overall coverage is relatively low – the JaCoCo report indicates about 28% instruction coverage (i.e., 28% of the bytecode instructions were executed during tests), which corresponds to roughly 34.1% line coverage. In other words, only about one-third of the lines of code are covered by tests. This is because the unit tests targeted only the service layer; the controller's methods and the main application (and some parts of the entity) were never invoked in tests. Below is a breakdown of coverage by class:

- `StudentService` – 100% of lines covered. All service methods were executed by the unit tests, so this class is fully tested.
- `Student` (Entity) – ~41% of lines covered. The entity is partially covered indirectly through the tests (the tests create `Student` objects and call a couple of getters). Many of the entity's simple getters/setters are not explicitly tested, which is generally acceptable since they are trivial, but they count toward uncovered lines.
- `StudentController` – 0% lines covered. None of the controller endpoints were tested, so this class's 14 lines of logic (mostly the endpoint methods) were never executed in the test run.
- `StudentApiApplication` – 0% lines covered. The main method and configuration were not tested (commonly, the main class isn't explicitly tested in unit tests).

Overall, the project's line coverage is ~34%, which is below typical industry recommendations (many projects aim for ~80% or higher). This low figure is a result of focusing on unit tests for one layer and the absence of integration tests. The test suite effectively validates the core service logic but does not ensure the controller endpoints work as expected. Despite the low coverage, the most critical logic (service methods) has been verified by tests, which gives some confidence in those parts of the code.

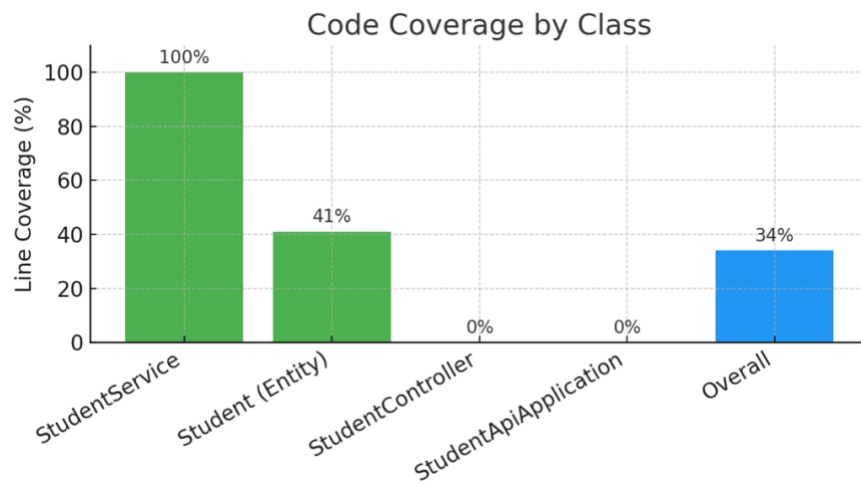


Figure : Code Coverage by Class.

This chart (generated from the JaCoCo report) shows the percentage of lines covered by tests for each major class in the application. As discussed, the service layer has 100% coverage, while the controller and the main application have 0%. The overall line coverage is ~34%. Improving these numbers would involve writing additional tests for the controller (and possibly repository integration), as outlined in the report.

4. Static Code Analysis using SonarQube

Static code analysis was performed using SonarQube, which was integrated into the CI pipeline. After each build, the code was analyzed for potential issues in reliability, security, and maintainability, as well as to compute duplicate code and coverage metrics. It was implemented on the AWS EC2 instance as shown in the figure below.

The screenshot displays the AWS Management Console interface. The top navigation bar shows the AWS logo, a search bar, and the user's profile (Atharva909). The left sidebar contains navigation links for EC2, Images, Elastic Block Store, and Network & Security. The main content area shows the 'Instances (1/1)' page for the 'sonarqube' instance (ID: i-0086ac198df534d24). The instance is in a 'Running' state, using the 't2.medium' instance type, and is located in the 'us-east-1d' availability zone. The instance's public IP address is 107.21.51.131. Below the instance details, there are tabs for 'Details', 'Status and alarms', 'Monitoring', 'Security', 'Networking', 'Storage', and 'Tags'. The 'Details' tab is selected, showing various instance attributes such as 'Instance ID', 'Public IPv4 address', 'Private IPv4 addresses', 'Instance state', 'Private IP DNS name (IPv4 only)', 'Instance type', 'VPC ID', 'Subnet ID', 'Hostname type', 'Answer private resource DNS name', 'Auto-assigned IP address', and 'IAM Role'. Below the instance details, there is a terminal window showing the output of system information and Docker commands. The terminal output includes system load, memory usage, and a list of running Docker containers. A notification banner at the bottom of the terminal window indicates that the instance is running and provides public and private IP addresses.

Instances (1/1)

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...
sonarqube	i-0086ac198df534d24	Running	t2.medium	2/2 checks passed	View alarms +	us-east-1d	ec2-107-21-51-131.co...	107.21.51.131

i-0086ac198df534d24 (sonarqube)

Instance summary

Attribute	Value
Instance ID	i-0086ac198df534d24
Public IPv4 address	107.21.51.131 open address
Private IPv4 addresses	172.31.27.26
Instance state	Running
Private IP DNS name (IPv4 only)	ip-172-31-27-26.ec2.internal open address
Instance type	t2.medium
VPC ID	vpc-0cd8d9771a8c12f77 open address
Subnet ID	subnet-0b6a0fe8189ad3d9 open address
Hostname type	IP name: ip-172-31-27-26.ec2.internal
Answer private resource DNS name	IPV4 (A)
Auto-assigned IP address	107.21.51.131 [Public IP]
IAM Role	-

CloudShell

```
Welcome to Ubuntu 22.04.5 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/pro

System information as of Mon Apr 7 17:48:14 UTC 2025

System load: 0.0          Processes: 149
Usage of /: 77.6% of 7.57GB Users logged in: 0
Memory usage: 69%        IPv4 address for eth0: 172.31.27.26
Swap usage: 0k

* Ubuntu Pro delivers the most comprehensive open source security and
  compliance features.
  https://ubuntu.com/aws/pro

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

1 additional security update can be applied with ESM Apps.
Learn more about enabling ESM Apps service at https://ubuntu.com/esm

New release '24.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Mon Apr 7 15:05:55 2025 from 18.206.107.28
ubuntu@ip-172-31-27-26:~$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED    STATUS    PORTS
3b9149a48241   sonarqube:community "/opt/sonarqube/dock-" 27 hours ago Up 27 hours 0.0.0.0:9000->9000/tcp, :::9000->9000/tcp
ad3c557937d2   postgres:13    "docker-entrypoint.s-" 27 hours ago Up 27 hours 5432/tcp

NAME          sonarqube
postgres
```

i-0086ac198df534d24 (sonarqube)

PublicIPs: 107.21.51.131 PrivateIPs: 172.31.27.26

The SonarQube dashboard for the Student Management API project shows that the application passed all configured quality gates. In particular, the project earned the following ratings and metrics:

- **Reliability:** A (Grade A, meaning zero bugs were detected by SonarQube) – The code has no known reliability issues. All functions appear to behave as intended without detecting any bug patterns.
- **Security:** A (no vulnerabilities found) – SonarQube did not identify any security vulnerabilities or hotspots in the code. This is expected given the simplicity of the application (no external inputs besides the REST API, and using Spring Boot which by default handles common vulnerabilities if used properly).
- **Maintainability:** A (zero code smells or very low technical debt) – The codebase is small and straightforward, resulting in no code smell issues reported. SonarQube calculates a maintainability rating based on technical debt; an A rating indicates that if there are any code smells, they are minor and would take very little time to fix.
- **Coverage:** 34.1% (**test coverage**) – This is the line coverage percentage imported from JaCoCo. While this number is relatively low, it still passed the quality gate in SonarQube (possibly the quality gate was configured to not fail below a certain threshold, or the focus was on ensuring *some* coverage and clean code rather than enforcing a high coverage minimum). This metric reinforces the earlier analysis that only about one-third of the code is covered by tests.
- **Duplications:** 0.0% – There is no duplicated code in the project according to SonarQube’s duplication detection. This is unsurprising given the project’s size; each piece of functionality is defined only once (no copy-pasted code). The presence of a service layer also means logic isn’t repeated in the controller, for example.

Because all these metrics meet the defined standards (e.g., no critical issues, coverage above 0%, etc.), the Quality Gate status is marked as Passed. In SonarQube’s terminology, a quality gate is a set of conditions that the project must fulfill (for instance, “Coverage > 80%” or “No new blocker issues”). The assignment likely used a default or custom quality gate that this project satisfied. Achieving an all-green dashboard was a key goal – it demonstrates that the code is of high quality (at least by automated checks). The static analysis process helped ensure that the team (or individual developer) adhered to good coding practices. For example, SonarQube would have flagged issues like unused variables, complexity, or bad practices if they existed; none of those appear in the final code. This implies that any such issues were corrected during development or none were introduced.

5. Challenges and Improvements

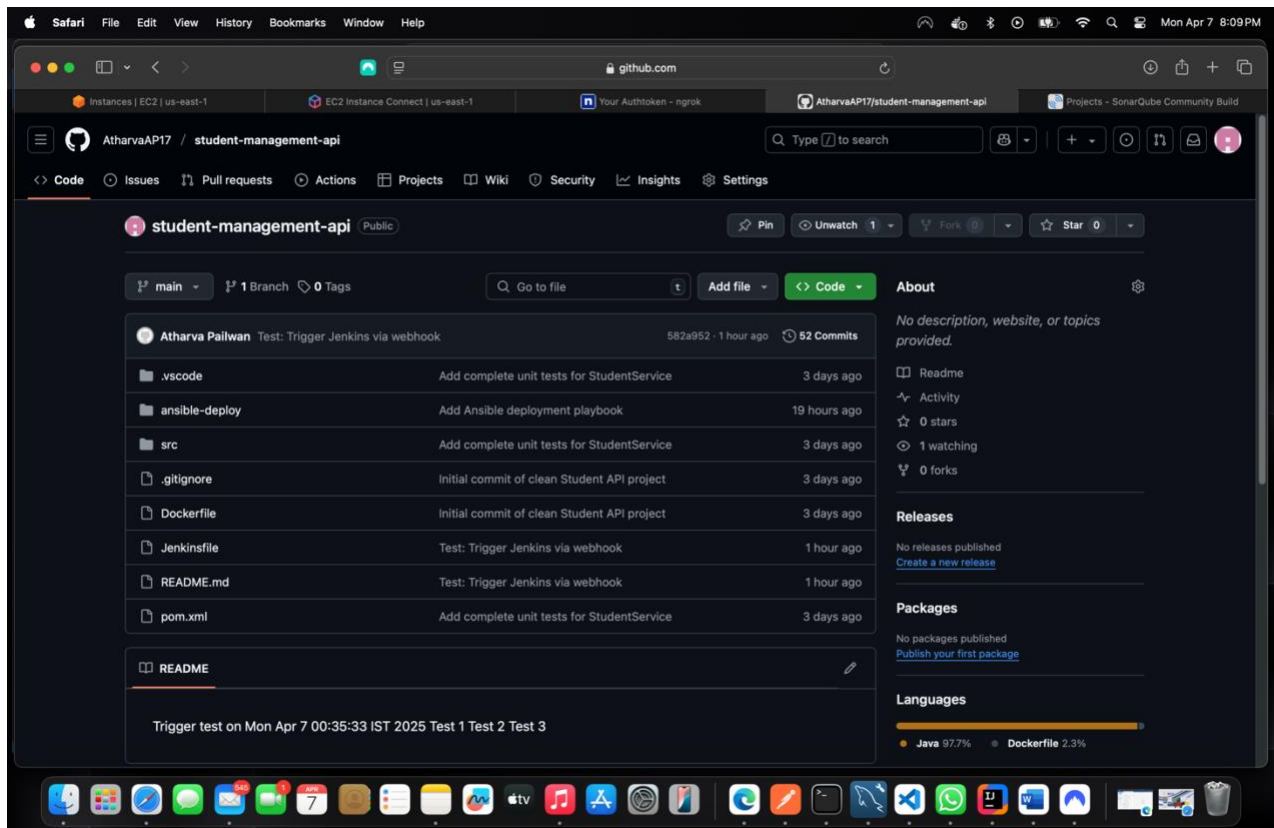
While the project met its primary objectives, there were several challenges encountered and areas for improvement identified during implementation:

- **Test Coverage and Depth:** Achieving comprehensive testing was challenging within the project timeline. As noted, only unit tests for the service layer were implemented, leading to modest coverage. **Improvement:** In the future, more tests should be added to increase coverage. This includes **integration tests** for the web layer (using Spring's MockMvc or an in-memory HTTP client) to test the StudentController endpoints. For example, tests could be written to simulate HTTP GET/POST requests and assert the correct HTTP responses and database state. Additionally, testing edge cases (such as requesting a non-existent student ID and expecting a 404 error) would improve reliability. If end-to-end testing is desired, one could use tools like Postman or Karate to test the running application as a black box. Increasing the coverage closer to the recommended 80% would better satisfy industry best practices and assignment expectations.
- **Exception Handling:** The current controller simply throws a runtime exception when a student is not found. This was a quick solution but not ideal, as it results in a 500 Internal Server Error. **Improvement:** Implement a custom exception (e.g., StudentNotFoundException) and a global exception handler to return a 404 Not Found status with a meaningful message. This would make the API more user-friendly and align with RESTful principles.
- **Continuous Integration Setup:** Setting up Jenkins and SonarQube integration involved dealing with credentials and configurations (for example, generating a SonarQube token, configuring Jenkins agents with Docker and Ansible, etc.). Ensuring the Jenkins pipeline worked end-to-end required troubleshooting environment issues (like installing the correct Maven version on the Jenkins node, Docker daemon access, etc.). **Improvement:** One possible enhancement is to add a SonarQube **quality gate check** in the pipeline. Currently, the pipeline runs SonarQube analysis but does not explicitly stop the build if the quality gate fails. Jenkins has a way to wait for SonarQube to compute the quality gate result and then mark the build failed if it didn't pass. Integrating that would enforce quality standards strictly (e.g., if in the future we set a higher coverage threshold or other conditions).
- **Pipeline Performance:** The CI/CD pipeline, while functional, could be optimized. For instance, the build and test stages could potentially be combined (since mvn package already runs tests by default). Alternatively, one could use Maven's parallel test execution to speed up the test stage (not crucial here due to few tests). Caching Maven dependencies on the Jenkins agent is another improvement – this would avoid downloading the same libraries on each build, thus speeding up the build process.
- **Docker and Deployment:** Containerizing the application was straightforward with a multi-stage Dockerfile, but deploying via Ansible required the target environment to have Docker and the correct configuration. One challenge was ensuring the Ansible playbook could run on the Jenkins host (using localhost in this case) with appropriate permissions. Using Ansible's **Docker modules** or even Docker Compose could simplify the deployment. **Improvement:** In a more advanced scenario, the deployment could be improved by deploying to a Kubernetes cluster for scalability, but that was beyond the scope. Another small improvement is to use versioning tags for Docker images (e.g., tagging images with the Git commit or version number instead of always "latest") for better traceability of deployments.
- **Security Considerations:** Although SonarQube didn't find security issues, real-world improvements could include adding authentication/authorization to the API (e.g., using Spring Security with JWT). This wasn't required in the assignment, but it's an area to consider for making the service production-ready. Also, currently the H2 database is used without credentials (H2 console enabled in dev mode), which is fine for a demo but would be locked down in a production environment.
- **Project Structure:** All code resides in one package (com.example.studentapi). For better organization, especially as the project grows, the code could be refactored into sub-packages (e.g., .controller, .service, .repository, .model). This doesn't change runtime behavior but improves maintainability by clearly separating layers. It's a minor structural improvement

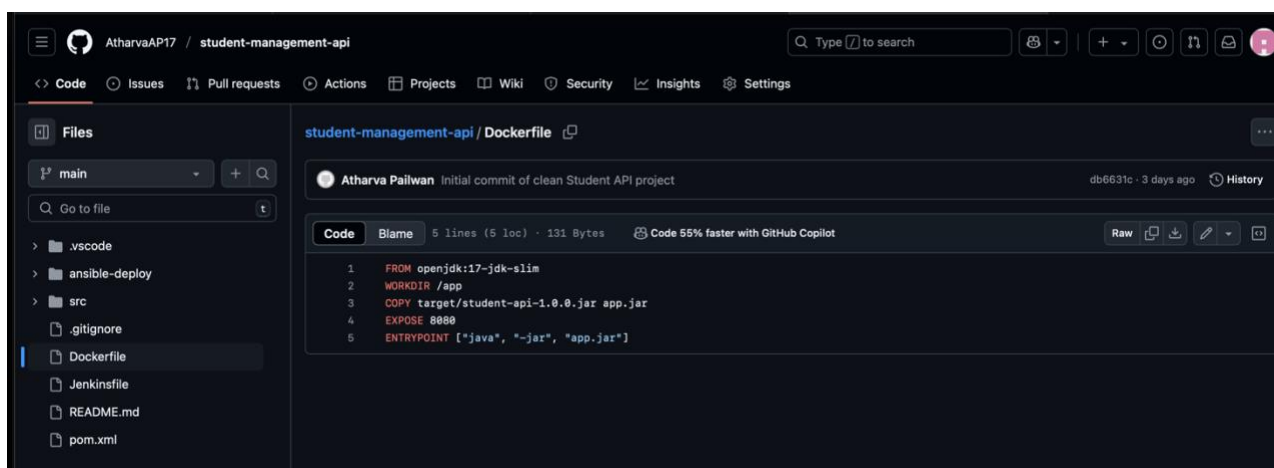
6. Screenshots

Code file pushed to the git repository containing all the files(Jenkinsfile, Dockerfile, Ansible playbook, etc.):

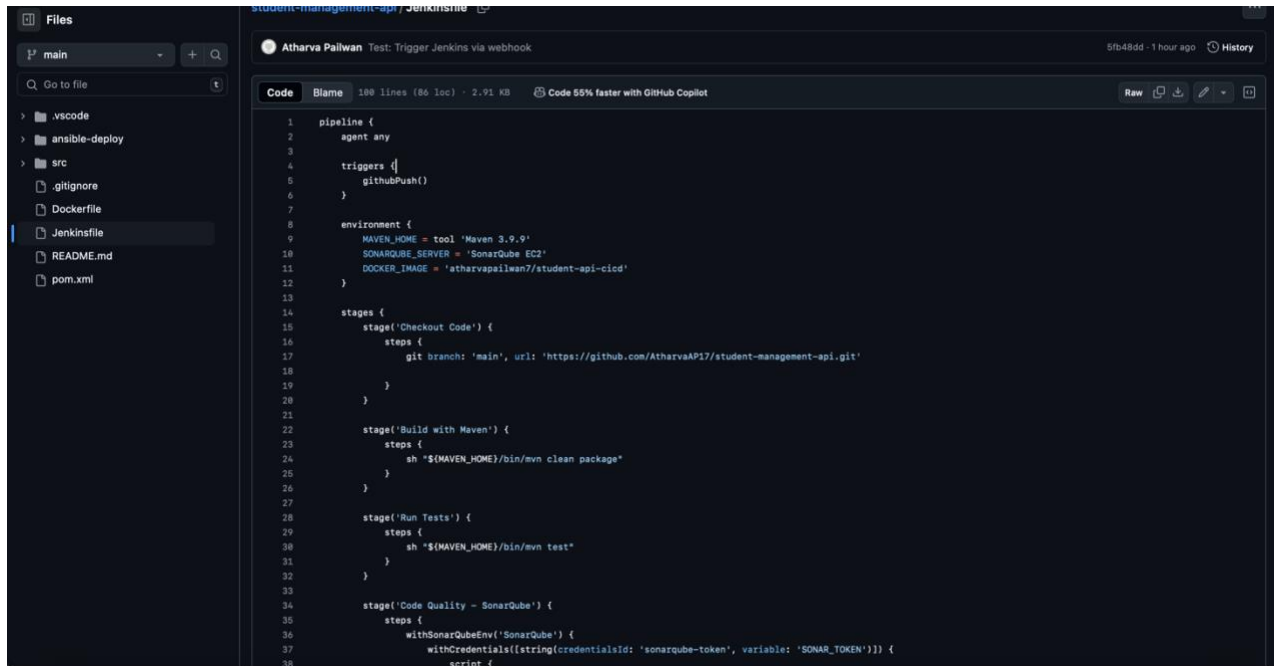
GitHub Repository link: <https://github.com/AtharvaAPI7>



Dockerfile:



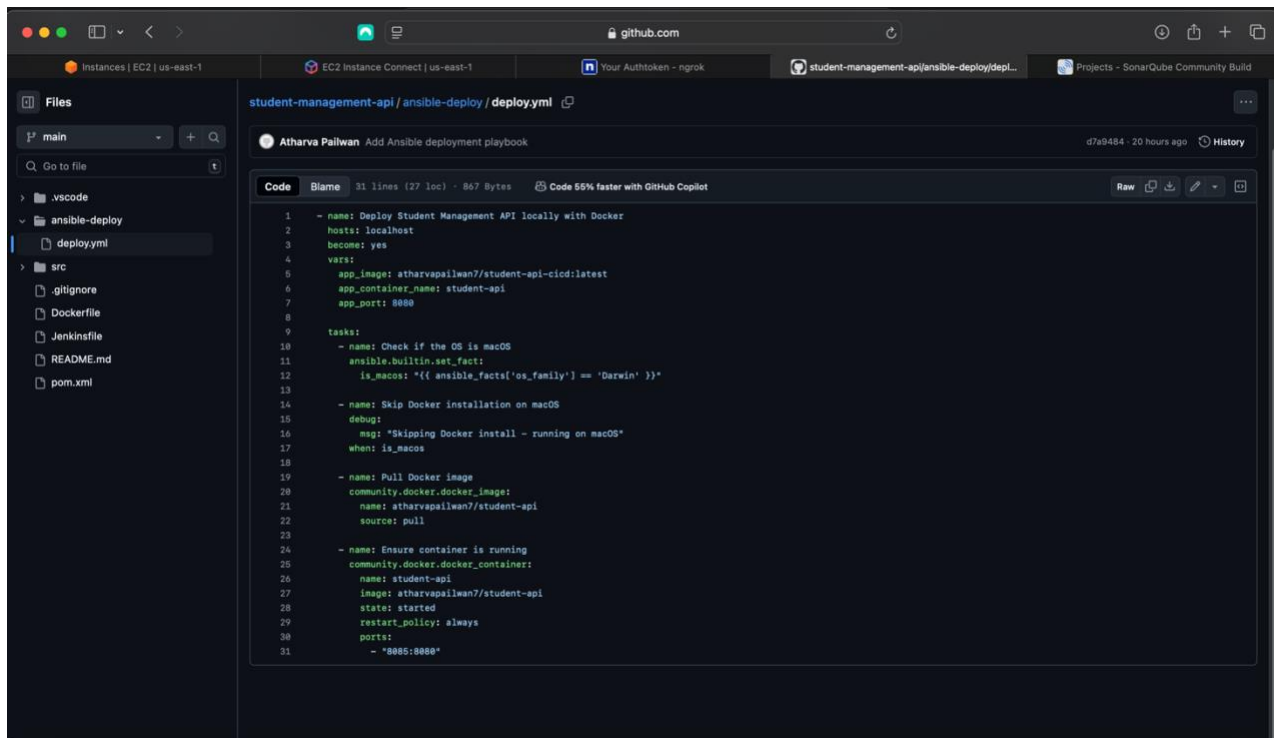
Jenkinsfile:



The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like .vscode, ansible-deploy, src, .gitignore, Dockerfile, Jenkinsfile, README.md, and pom.xml. The Jenkinsfile is selected. The code editor shows the Jenkinsfile content, which is a pipeline script. The pipeline has a name 'Test: Trigger Jenkins via webhook' and a trigger 'githubPush()'. It has an environment section with variables for MAVEN_HOME, SONARQUBE_SERVER, and DOCKER_IMAGE. The pipeline has four stages: 'Checkout Code', 'Build with Maven', 'Run Tests', and 'Code Quality - SonarQube'. The 'Code Quality - SonarQube' stage uses the 'withSonarQubeEnv' plugin to run SonarQube scans.

```
1 pipeline {
2   agent any
3
4   triggers {
5     githubPush()
6   }
7
8   environment {
9     MAVEN_HOME = tool 'Maven 3.9.9'
10    SONARQUBE_SERVER = 'SonarQube EC2'
11    DOCKER_IMAGE = 'atharvapailwan7/student-api-cicd'
12  }
13
14  stages {
15    stage('Checkout Code') {
16      steps {
17        git branch: 'main', url: 'https://github.com/AtharvaAP17/student-management-api.git'
18      }
19    }
20
21    stage('Build with Maven') {
22      steps {
23        sh "${MAVEN_HOME}/bin/mvn clean package"
24      }
25    }
26
27    stage('Run Tests') {
28      steps {
29        sh "${MAVEN_HOME}/bin/mvn test"
30      }
31    }
32
33    stage('Code Quality - SonarQube') {
34      steps {
35        withSonarQubeEnv('SonarQube') {
36          withCredentials([string(credentialsId: 'sonarqube-token', variable: 'SONAR_TOKEN')]) {
37            script {
38
```

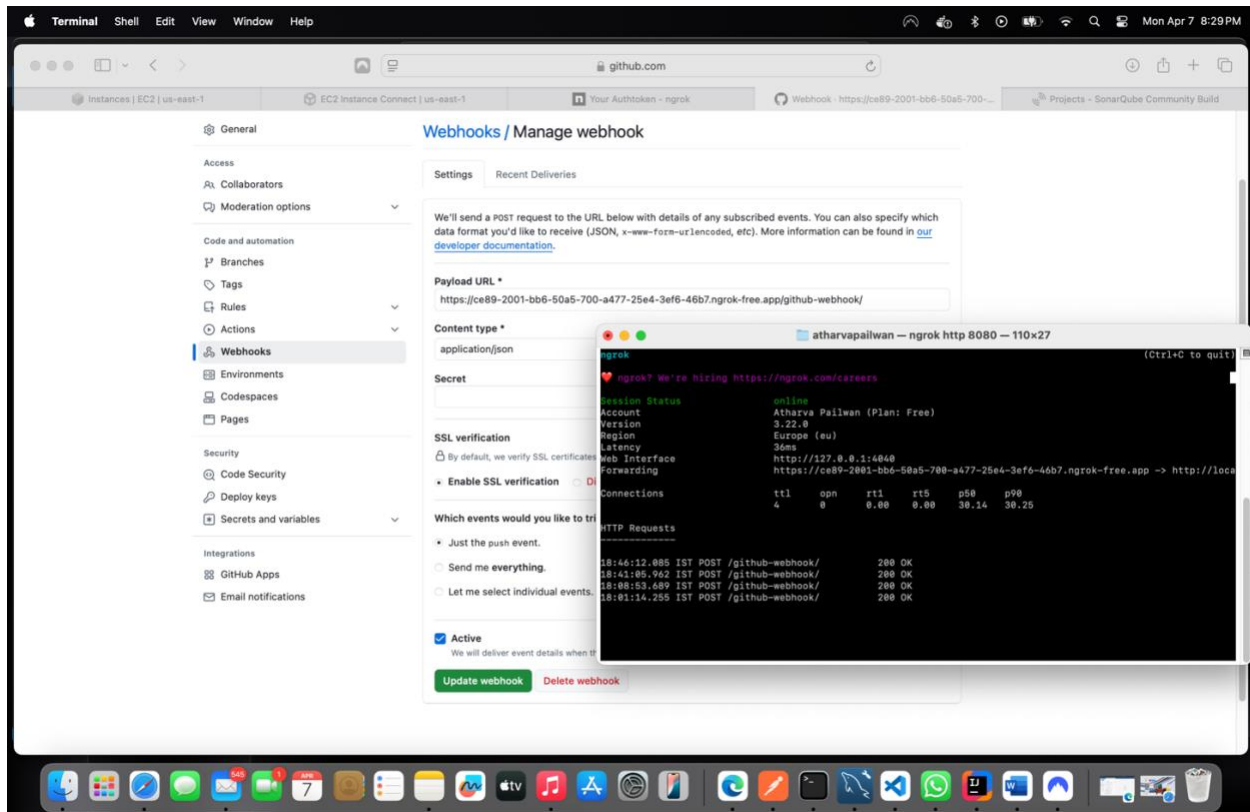
Ansible deploy.yml file:



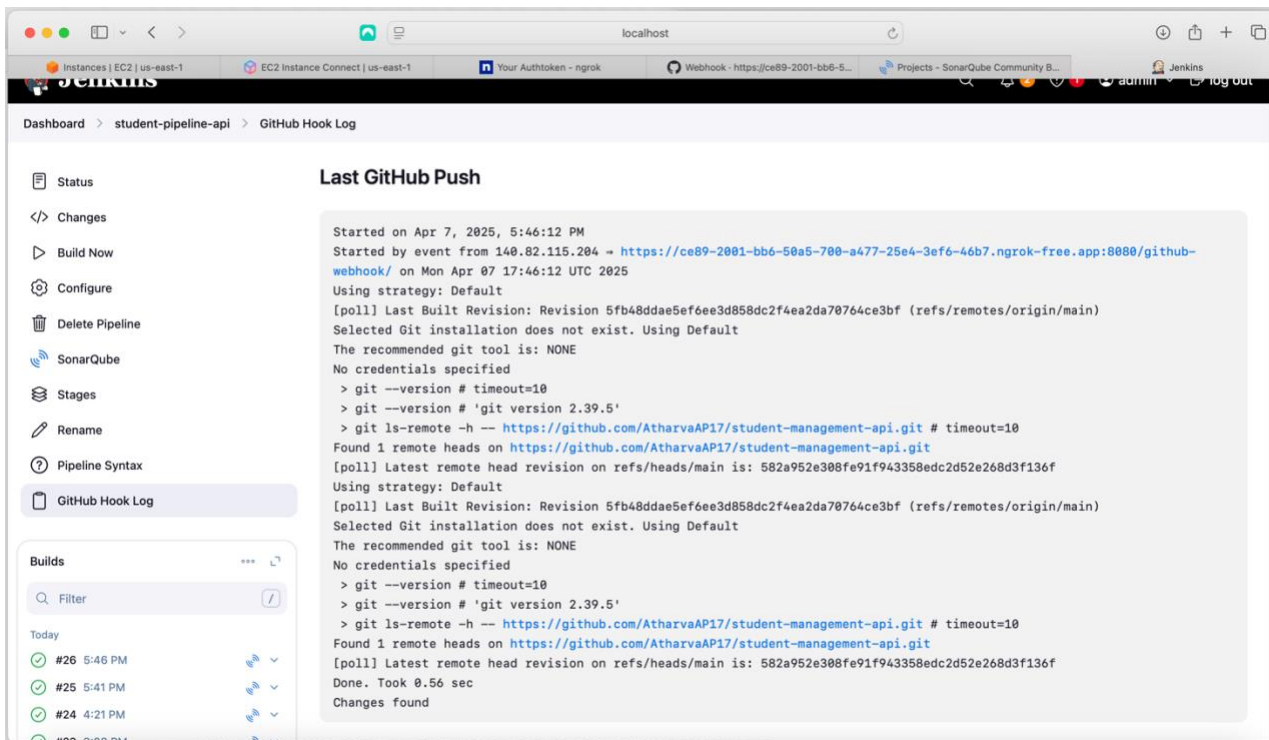
The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with files like .vscode, ansible-deploy, src, .gitignore, Dockerfile, Jenkinsfile, README.md, and pom.xml. The ansible-deploy directory is selected, and the deploy.yml file is open. The code editor shows the Ansible playbook content. The playbook has a name 'Deploy Student Management API locally with Docker' and a host 'localhost'. It has a 'become: yes' option and a 'vars' section with variables for app_image, app_container_name, and app_port. The tasks section contains four tasks: 'Check if the OS is macOS', 'Skip Docker installation on macOS', 'Pull Docker image', and 'Ensure container is running'.

```
1 - name: Deploy Student Management API locally with Docker
2   hosts: localhost
3   become: yes
4   vars:
5     app_image: atharvapailwan7/student-api-cicd:latest
6     app_container_name: student-api
7     app_port: 8080
8
9   tasks:
10    - name: Check if the OS is macOS
11      ansible.builtin.set_fact:
12        is_macos: "{{ ansible_facts['os_family'] == 'Darwin' }}"
13
14    - name: Skip Docker installation on macOS
15      debug:
16        msg: "Skipping Docker install - running on macOS"
17      when: is_macos
18
19    - name: Pull Docker image
20      community.docker.docker_image:
21        name: atharvapailwan7/student-api
22        source: pull
23
24    - name: Ensure container is running
25      community.docker.docker_container:
26        name: student-api
27        image: atharvapailwan7/student-api
28        state: started
29        restart_policy: always
30        ports:
31          - "8085:8080"
```

For creating an automated pipeline trigger, github webhook was created with ngrok's port forwarding which can be seen in the screenshot below:



Github Webhook log:



Pushing code to the github repo which triggers the pipeline:

```
> TERMINAL
atharvapaiwan@Atharvas-MacBook-Pro student-management-api % git commit -m "Test: Trigger Jenkins via webhook"
[main 582a952] Test: Trigger Jenkins via webhook
Committer: Atharva Paiwan <atharvapaiwan@mac.home>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 2 insertions(+), 1 deletion(-)
atharvapaiwan@Atharvas-MacBook-Pro student-management-api % git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 11 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/AtharvaAPI7/student-management-api.git
5fb48dd..582a952 main -> main
atharvapaiwan@Atharvas-MacBook-Pro student-management-api % git add README.md
atharvapaiwan@Atharvas-MacBook-Pro student-management-api % git commit -m "Test: Trigger Jenkins via webhook"
[main 7a67822] Test: Trigger Jenkins via webhook
Committer: Atharva Paiwan <atharvapaiwan@mac.home>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

1 file changed, 1 insertion(+), 1 deletion(-)
atharvapaiwan@Atharvas-MacBook-Pro student-management-api % git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 11 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 303 bytes | 303.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
```

Pipeline starts automatically:

The screenshot shows a web browser window displaying the Jenkins dashboard for a pipeline named 'student-pipeline-api'. The dashboard includes a sidebar with navigation options like 'Configure', 'Delete Pipeline', 'SonarQube', 'Stages', 'Rename', 'Pipeline Syntax', and 'GitHub Hook Log'. The main content area shows the pipeline's status as 'Passed' with a green bar chart indicating successful builds. A 'Latest Test Result' section shows 'no failures'. A 'Permalinks' section lists recent builds with their status and timestamps. A 'Builds' list on the left shows a series of builds from #21 to #28, with #28 being the most recent and successful.

Dashboard > student-pipeline-api >

Configure
Delete Pipeline
SonarQube
Stages
Rename
Pipeline Syntax
GitHub Hook Log

Student Management API **Passed**
server-side processing: **Success**

Latest Test Result (no failures)

Permalinks

- Last build (#27), 1 min 28 sec ago
- Last stable build (#27), 1 min 28 sec ago
- Last successful build (#27), 1 min 28 sec ago
- Last failed build (#22), 4 hr 44 min ago
- Last unsuccessful build (#22), 4 hr 44 min ago
- Last completed build (#27), 1 min 28 sec ago

Builds

Filter

Started 5.2 sec ago
Estimated remaining time: 33 sec

Today

- #28 7:46 PM
- #27 7:44 PM
- #26 5:46 PM
- #25 5:41 PM
- #24 4:21 PM
- #23 3:08 PM
- #22 3:02 PM
- #21 1:14 AM

Github Webhook's recent deliverables confirming the trigger response:

AtharvaAP17 / student-management-api

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Rules

Actions

Webhooks

Environments

Codespaces

Pages

Security

Code Security

Deploy keys

Secrets and variables

Integrations

GitHub Apps

Email notifications

Webhooks / Manage webhook

Settings Recent Deliveries

✓ 08e80e26-13e9-11f0-8d42-b91af5d5319d push 2025-04-07 20:46:45 ...

Request **Response** 200 Redeliver Completed in 0.42 seconds.

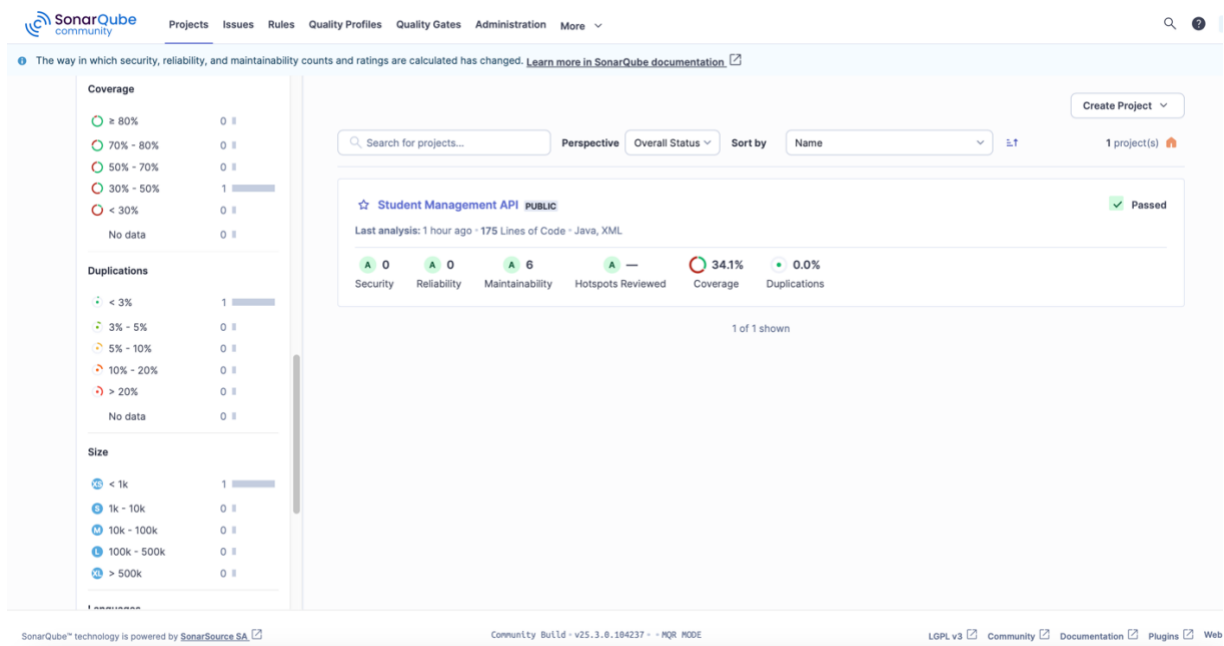
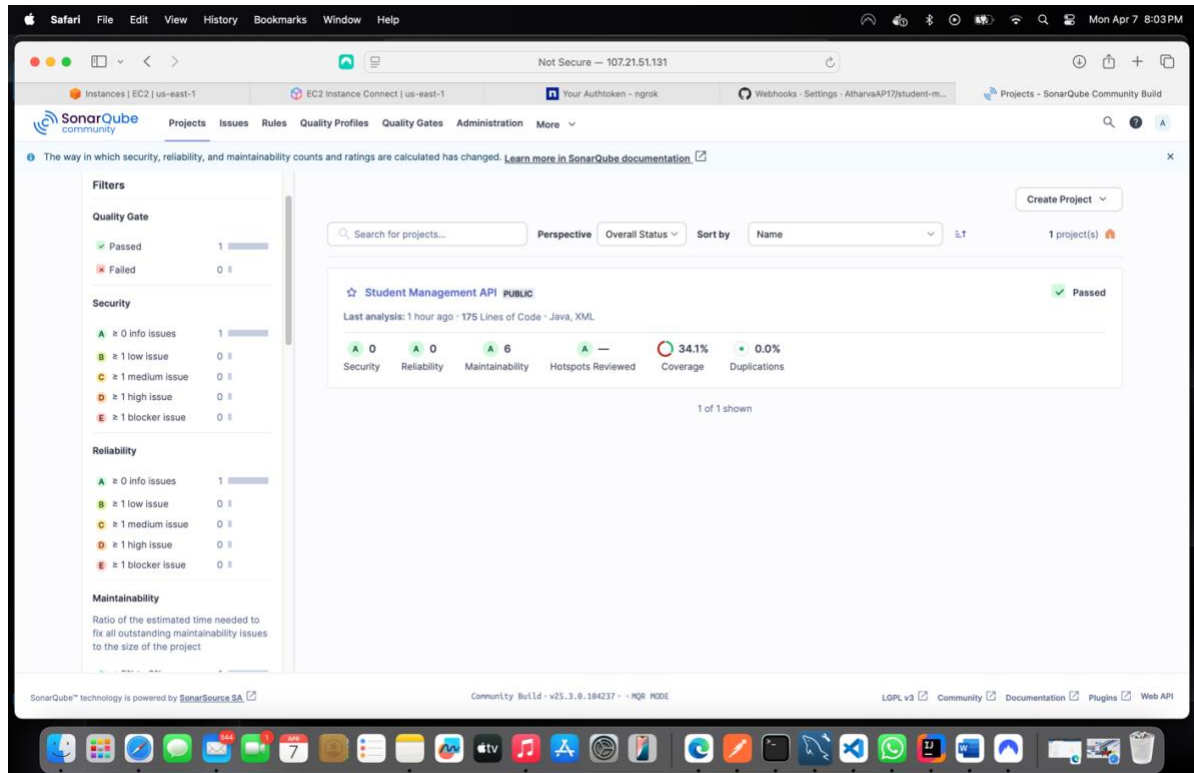
Headers

Request URL: https://ce89-2001-bb6-50a5-700-a477-25e4-3ef6-46b7.ngrok-free.app/github-webhook/
Request method: POST
Accept: */*
Content-Type: application/json
User-Agent: GitHub-Hookshot/21a6bf0
X-GitHub-Delivery: 08e80e26-13e9-11f0-8d42-b91af5d5319d
X-GitHub-Event: push
X-GitHub-Hook-ID: 539555292
X-GitHub-Hook-Installation-Target-ID: 960536454
X-GitHub-Hook-Installation-Target-Type: repository

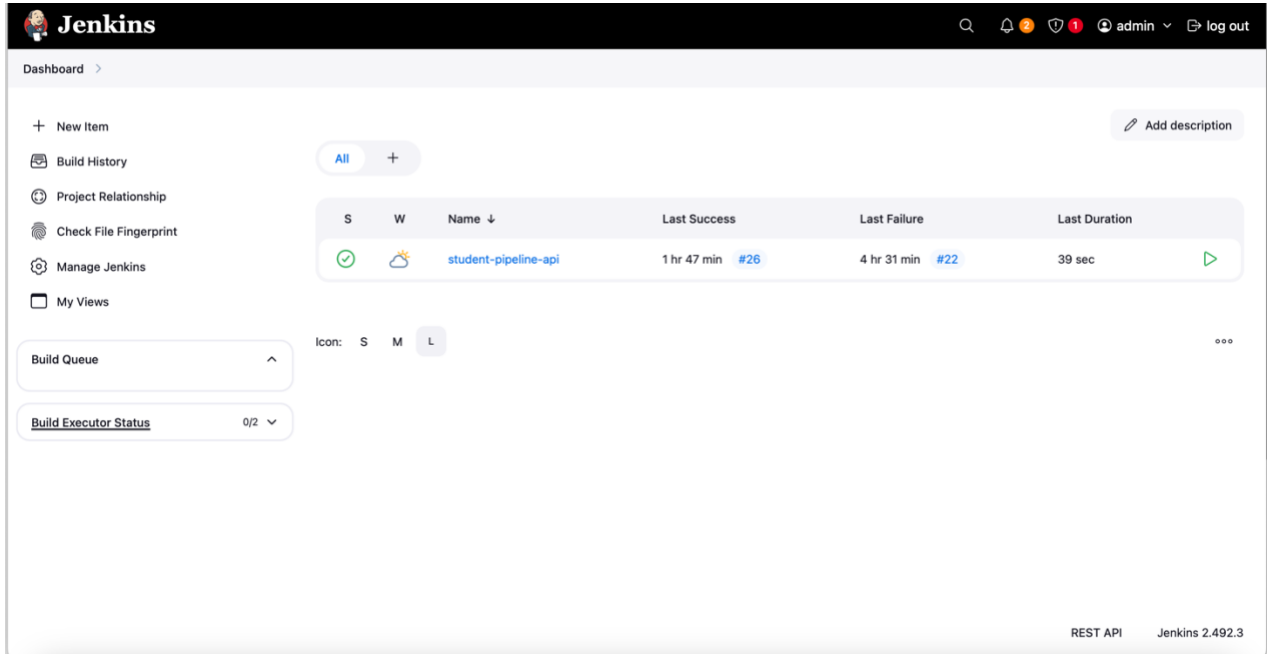
Payload

```
{
  "ref": "refs/heads/main",
  "before": "7a6782264210af8018b3b20ebfdbfc01e01c98b",
  "after": "2f64927baa36998edcd1a5f13882e0eb57c23418",
  "repository": {
    "id": 960536454,
    "node_id": "R_kgDOUCfhg",
    "name": "student-management-api",
    "full_name": "AtharvaAP17/student-management-api",
    "private": false,
    "owner": {
      "name": "AtharvaAP17",
      "email": "106423494+AtharvaAP17@users.noreply.github.com",
```

The SonarQube screenshots below illustrates the summary of these metrics and the quality gate result. Notably, the A-grade ratings in reliability, security, and maintainability indicate a clean bill of health in those categories, which is a strong positive outcome of the static analysis. The main area of improvement highlighted by SonarQube is the test coverage (which is marked, but still green in the context of the quality gate). Overall, the static analysis confirms that the Student Management API code is well-written and maintainable for the scope of the project.



Jenkins Dashboard hosted on port 8080 showing the latest job status and last failure:

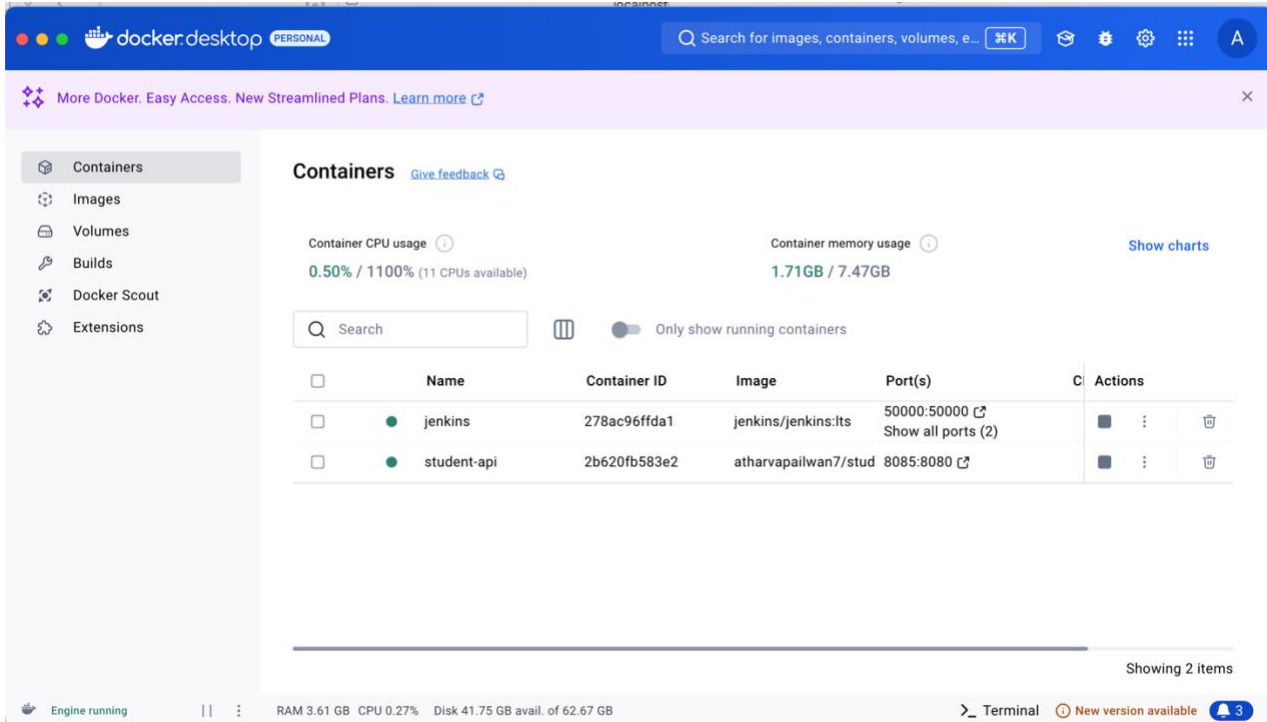


The Jenkins Dashboard is shown with the following components:

- Header:** Jenkins logo, search icon, notification bell, shield icon, and user 'admin' with a 'log out' button.
- Left Sidebar:** Navigation menu with 'New Item', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', and 'My Views'.
- Main Content Area:**
 - Build Queue:** A dropdown menu showing '0/2'.
 - Build Executor Status:** A dropdown menu showing '0/2'.
 - Job List Table:**

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓	☁	student-pipeline-api	1 hr 47 min #26	4 hr 31 min #22	39 sec
 - Footer:** REST API and Jenkins 2.492.3.

Running Jenkins and student-api Containers running on Docker desktop:



The Docker Desktop interface shows the following details:

- Header:** Docker Desktop logo, search bar, and user 'A'.
- Left Sidebar:** Navigation menu with 'Containers', 'Images', 'Volumes', 'Builds', 'Docker Scout', and 'Extensions'.
- Main Content Area:**
 - Containers Section:**
 - CPU Usage:** 0.50% / 1100% (11 CPUs available)
 - Memory Usage:** 1.71GB / 7.47GB
 - Show charts** button
 - Search** bar and **Only show running containers** toggle.
 - Container List Table:**

	Name	Container ID	Image	Port(s)	C	Actions
<input type="checkbox"/>	jenkins	278ac96ffda1	jenkins/jenkins:its	50000:50000 ⚙ Show all ports (2)	■	⋮ ⚡
<input type="checkbox"/>	student-api	2b620fb583e2	atharvapaiwan7/stud	8085:8080 ⚙	■	⋮ ⚡
- Footer:** Engine running, RAM 3.61 GB, CPU 0.27%, Disk 41.75 GB avail. of 62.67 GB, Terminal icon, New version available notification, and 3 notifications.

Code coverage using Jacoco:

The first screenshot shows the Jacoco index.html report for the 'Student Management API'. The report displays overall coverage statistics for the entire project.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.studentapi	110 of 154	28%	0 of 0	n/a	20	27	27	41	20	27	2	4
Total	110 of 154	28%	0 of 0	n/a	20	27	27	41	20	27	2	4

The second screenshot shows the Jacoco report for the 'com.example.studentapi' package. The report displays detailed coverage statistics for each class in the package.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
StudentController	0%		n/a		8	8	14	14	8	8	1	1
Student	33%		n/a		10	12	10	17	10	12	0	1
StudentApiApplication	0%		n/a		2	2	3	3	2	2	1	1
StudentService	100%		n/a		0	5	0	7	0	5	0	1
Total	110 of 154	28%	0 of 0	n/a	20	27	27	41	20	27	2	4

StudentService

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
StudentService(StudentRepository)		100%		n/a	0	1	0	3	0	1
save(Student)		100%		n/a	0	1	0	1	0	1
findById(Long)		100%		n/a	0	1	0	1	0	1
deleteById(Long)		100%		n/a	0	1	0	1	0	1
findAll()		100%		n/a	0	1	0	1	0	1
Total	0 of 26	100%	0 of 0	n/a	0	5	0	7	0	5

Created with JaCoCo 0.8.10.202304240956

StudentService.java

```

1. package com.example.studentapi;
2.
3. import org.springframework.stereotype.Service;
4. import java.util.List;
5. import java.util.Optional;
6.
7. @Service
8. public class StudentService {
9.     private final StudentRepository repository;
10.
11.     public StudentService(StudentRepository repository) {
12.         this.repository = repository;
13.     }
14.
15.     public List<Student> findAll() { return repository.findAll(); }
16.
17.     public Optional<Student> findById(Long id) { return repository.findById(id); }
18.
19.     public Student save(Student student) { return repository.save(student); }
20.
21.     public void deleteById(Long id) { repository.deleteById(id); }
22. }

```

Created with JaCoCo 0.8.10.202304240956

7. Conclusion and Evaluation:

The Student Management API project successfully demonstrates the implementation of a CI/CD pipeline for a Spring Boot microservice. All key deliverables of the assignment were achieved: the application code was developed with clear structure and layered architecture, and a fully automated pipeline was set up to build, test, analyze, containerize, and deploy the service. As a result, the project can be built and released at the push of a code commit, which is a significant outcome for continuous delivery practices. In terms of deliverables:

- The project's source is managed in Git and builds reproducibly with Maven, producing an executable JAR.
- A Jenkins pipeline automates the build and runs the test suite and static analysis on each push, ensuring continuous integration.
- Quality assurance is embedded into the process via SonarQube, and the project passed all quality gate metrics (with an A-rating in all categories and no critical issues).
- The service is containerized with Docker, and the pipeline automatically pushes the image to a registry.
- Deployment is handled through an Ansible script, demonstrating infrastructure-as-code and automated deployment to a server environment.
- Evidence of testing (JUnit results and a JaCoCo coverage report) and static analysis (SonarQube dashboard) have been provided, satisfying the assignment's reporting requirements.

Through this project, the following learning outcomes were realized: The importance of automated testing and early bug detection was underscored by the CI process, as even this small project benefited from having tests catch regressions immediately. Integrating SonarQube showed how maintaining code quality is an ongoing activity, not a one-time task. Working with Docker and Ansible in concert with Jenkins provided practical experience in deployment automation and the DevOps toolchain. Perhaps most importantly, the exercise reinforced the value of a coherent CI/CD pipeline — it increases confidence in each code change and lays the groundwork for rapid, reliable releases.

In conclusion, the Student Management API project met its objectives by delivering both a functional microservice and a robust continuous delivery pipeline. While there remains room for improvement (especially in expanding the test suite and refining deployment), the project as delivered provides a strong foundation. It illustrates how modern software engineering practices (CI/CD, automated testing, static analysis, and containerization) can be applied even to a relatively simple application to enhance its quality and maintainability. The experience gained from this assignment can be applied to more complex projects in the future, ensuring they too can achieve fast, high-quality, and continuous software delivery.