

## Software Engineering Seminar

# ProTicket

containerization of the project components, acceptance testing, API stress testing, and a basic CI/CD workflow using GitHub Actions.

---

**Tutor:** Carlos Andres Sierra

**Students:**

- Juan Daniel Vanegas Mayorquín — *Cod:* 20222020077
- Juan Sebastian Villalba Roa — *Cod:* 20201020066
- Daniel Santiago Arcila Martínez — *Cod:* 20191020075

Universidad Francisco José de Caldas  
School of Engineering  
Systems Engineering

---

Bogotá D.C. 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Dockerfiles and Docker Compose</b>	<b>1</b>
2.1	Architecture Overview . . . . .	1
2.2	Dockerfiles for Each Component . . . . .	1
2.2.1	Java Backend Dockerfile . . . . .	1
2.2.2	Python Backend Dockerfile . . . . .	2
2.2.3	Frontend Dockerfile . . . . .	2
2.3	Docker Compose Configuration . . . . .	3
2.4	Build and Run Instructions . . . . .	4
<b>3</b>	<b>Cucumber Acceptance Tests</b>	<b>5</b>
3.1	Scope of Acceptance Testing . . . . .	5
3.2	Feature Files . . . . .	5
3.3	Step Definitions . . . . .	6
3.4	Execution Setup . . . . .	7
3.5	Test Results . . . . .	9
<b>4</b>	<b>JMeter API Stress Tests</b>	<b>9</b>
4.1	Test Plan Design . . . . .	9
4.2	Implementation Details . . . . .	10
4.3	Execution Environment . . . . .	11
4.4	Results and Analysis . . . . .	12
<b>5</b>	<b>GitHub Actions CI/CD Workflow</b>	<b>13</b>
5.1	Workflow Overview . . . . .	13
5.2	Workflow Definition . . . . .	13
5.3	Automated Testing Stage . . . . .	14
5.4	Docker Image Build and Publishing . . . . .	14
5.5	Evidence of Successful Runs . . . . .	15
<b>6</b>	<b>References</b>	<b>15</b>

# 1 Introduction

This section introduces the project context and the objective of Workshop 4. It summarizes the technologies used and the main quality attributes targeted (e.g., reproducible deployment, automated testing, and continuous integration).

## 2 Dockerfiles and Docker Compose

### 2.1 Architecture Overview

The system is fully containerised and runs on a shared Docker bridge network called `ops_proticket_net`. All services communicate over this internal network and expose only the necessary ports to the host.

- **Java Auth Backend (`auth_service`):** a Spring Boot microservice that handles authentication and JWT token generation. It connects to the MySQL database container.
- **Python Business Backend (`business_service`):** a FastAPI-based microservice that exposes the business APIs (tickets, events, orders). It validates JWT tokens using the same secret as the auth service and consumes the auth API through the internal network. It persists business data in PostgreSQL.
- **Frontend (`frontend_service`):** a Vite + React single-page application that is built into static assets and served by Nginx. In production it calls the Java and Python backends through their exposed HTTP endpoints.
- **Databases and Admin Tools:**
  - `mysql_auth`: MySQL instance for the auth service.
  - `phpmyadmin`: web UI to manage the MySQL database.
  - `postgres_business`: PostgreSQL instance for the business service.
  - `pgadmin`: web UI to manage the PostgreSQL database.

Each database container has its own named volume (`mysql_auth_data`, `postgres_business_data`) so data is preserved across container restarts. All credentials, ports and secrets (e.g., `JWT_SECRET`) are managed via a central `.env` file in the `ops` folder.

### 2.2 Dockerfiles for Each Component

#### 2.2.1 Java Backend Dockerfile

The Java auth backend uses a multi-stage Dockerfile located in `backend/auth/Dockerfile`. The first stage uses `maven:3.9-eclipse-temurin-17` as a build image:

- `WORKDIR /app`

- Copy `pom.xml` and `src/`
- Run `mvn -DskipTests clean package` to produce a runnable JAR

The second stage is a lightweight runtime based on `eclipse-temurin:17-jre-alpine`:

- Copies `target/*.jar` from the build stage to `/app/app.jar`.
- Sets `WORKDIR /app`.
- Exposes port 8080.
- Defines `SPRING_PROFILES_ACTIVE=docker` as the default profile (can be overridden in Docker Compose).
- Starts the application with `ENTRYPOINT ["java", "-jar", "/app/app.jar"]`.

At runtime, database connection and JWT configuration are injected via environment variables in the Docker Compose file (e.g. `MYSQL_DATABASE`, `MYSQL_USER`, `MYSQL_PASSWORD`, `JWT_SECRET`).

### 2.2.2 Python Backend Dockerfile

The Python business backend uses a simple single-stage Dockerfile in `backend/business/Dockerfile`:

- Base image: `python:3.11-slim`.
- `WORKDIR /app`.
- Copies the entire backend source code into the container.
- Installs dependencies with `pip install --no-cache-dir -r requirements.txt`.
- Exposes port 8000 inside the container.
- Starts the FastAPI application using Uvicorn

The container receives its configuration from both a local `.env` file (`..../backend/business/.env`) and additional environment variables specified in Docker Compose, such as `POSTGRES_HOST`, `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD`, and the shared `JWT_SECRET`. The service also receives `AUTH_API_BASE_URL` to call the Java auth backend.

### 2.2.3 Frontend Dockerfile

The React frontend (Vite) lives under `frontend/` and uses a two-stage Dockerfile:

- **Build stage** (`node:20-alpine`):
  - `WORKDIR /app`.

- Copies `package.json` and `package-lock.json` and runs `npm ci` to install dependencies.
- Copies the rest of the source code (`src/`, `vite.config.ts`, etc.).
- Runs `npm run build` to generate the static assets into the `dist/` directory.
- **Runtime stage (nginx:alpine):**
  - Copies the built `dist/` folder to `/usr/share/nginx/html`.
  - Exposes port 80.
  - Uses the default `nginx` command: `CMD ["nginx", "-g", "daemon off;"]`.

The container is exposed on port 5173 on the host via Docker Compose, mapping to port 80 inside the container.

## 2.3 Docker Compose Configuration

The main orchestration file is `ops/docker-compose.yaml`. It defines all services, their dependencies, shared volumes, and the common network:

- **Database layer:**
  - `mysql_auth` and `postgres_business` use the official `mysql:8.0` and `postgres:16` images, respectively. Each container uses environment variables from the central `.env` file for database names, users, passwords, and ports.
  - Named volumes `mysql_auth_data` and `postgres_business_data` ensure persistent storage.
  - Initialisation scripts are mounted from `./schemas/mysql_init` and `./schemas/postgres_init`.
- **Admin tools:**
  - `phpmyadmin` connects to `mysql_auth` using `PMA_HOST=mysql_auth` and is exposed on  `${PHPMYADMIN_PORT}`.
  - `pgadmin` connects to `postgres_business` and is exposed on  `${PGADMIN_PORT}` with credentials taken from `PGADMIN_DEFAULT_EMAIL` and `PGADMIN_DEFAULT_PASSWORD`.
- **Backends:**
  - `auth_service` is built from `../backend/auth/Dockerfile`. It depends on `mysql_auth`, uses the `SPRING_PROFILES_ACTIVE` environment variable, and is exposed on port 8080. Database config and `JWT_SECRET` are passed via environment variables.

- `business_service` is built from `../backend/business/Dockerfile`. It depends on `postgres_business`, mounts the source directory for development (`../backend/business:/app`), loads a local `.env` file, and receives PostgreSQL configuration and the `JWT_SECRET` through the main `.env`. It is exposed as `8001:8000` (host:container).
- **Frontend:**
  - `frontend_service` is built from `../frontend/Dockerfile`.
  - It depends on both backends to ensure that APIs are up before the UI is available.
  - Port mapping `5173:80` exposes the React application on `http://localhost:5173`.
- **Network:**
  - A single bridge network `protickets_net` is declared using the name from the environment variable `NETWORK_NAME=ops_proticket_net`. All containers join this network and can reach each other by service name.

## 2.4 Build and Run Instructions

To run the full stack, all commands are executed from the `ops/` directory, which contains the `docker-compose.yaml` file and the central `.env`.

1. **Prepare the environment:** ensure that the `.env` file is present in the `ops` folder and contains all variables (database credentials, ports, `JWT_SECRET`, etc.).
2. **Build and start the entire stack:**
  - Using the Makefile:
    - `make up-all`
  - Using Docker Compose directly:
    - `docker compose -env-file .env up -build -d`
3. **Stop the stack:**
  - `make down-all`
  - or `docker compose -env-file .env down`
4. **Clean everything (containers + volumes):**
  - `make clean-all`
  - or `docker compose -env-file .env down -v`

After running `make up-all` (or the equivalent Docker Compose command), the system is available at:

- Auth API: `http://localhost:8080`
- Business API: `http://localhost:8001`
- Frontend: `http://localhost:5173`
- phpMyAdmin: `http://localhost:${PHPMYADMIN_PORT}`
- pgAdmin: `http://localhost:${PGADMIN_PORT}`

### 3 Cucumber Acceptance Tests

#### 3.1 Scope of Acceptance Testing

Describe which main user stories are covered by Cucumber acceptance tests and how they were prioritized.

#### 3.2 Feature Files

Summarize the implemented feature files (e.g., `event_post.feature`, `event_stats.feature`) and the scenarios contained in each. Example for this two features:

##### `event_post.feature`:

```
As an organizer
I want to create an event post
So that it can be listed for sale

Scenario: Successful event creation
  Given the backend is running
  And an existing organizer with credentials
  When I create a new event with valid data
  Then I should receive a 200 status code
  And the response should contain the event information
```

##### `event_stats.feature`:

```
As an organizer
I want to see sales and remaining capacity
So that I can track performance

Scenario: Organizer retrieves event statistics
  successfully
  Given the backend is running
  And an existing event with sales
  When I fetch the statistics for the organizer
  Then I should receive a 200 status code
  And the response should include the event statistics
```

### 3.3 Step Definitions

Explain the structure of the step definitions and how they interact with the application (e.g., REST calls, database checks, UI steps). Example for `event_post.feature` steps definition.

```
from behave import given, when, then
import requests
from uuid import uuid4

BASE_URL = "http://localhost:8001"

# -----
# GIVEN
# -----


@given("an existing organizer with credentials")
def step_existing_organizer(context):

    user_id = str(uuid4())
    context.user_id = user_id
    context.token = f"Bearer {user_id}"

    context.organizer_payload = {
        "user_id": user_id,
        "organization_name": "Test Org"
    }

    response = requests.post(
        f"{BASE_URL}/organizers/",
        headers={"Authorization": context.token},
        json=context.organizer_payload
    )

    assert response.status_code == 200, f"Organizer creation failed: {response.text}"

    data = response.json()
    context.organizer_id = data["id_organizer"]


# -----
# WHEN
# -----


@when("I create a new event with valid data")
def step_create_event(context):
    print("Organizer ID used:", context.organizer_id)

    context.event_payload = {
```

```

        "title": "Rock Concert",
        "description": "A large outdoor concert",
        "location": "Main Stadium",
        "capacity": 5000,
        "price": 150.00,
        "start_datetime": "2025-12-25T20:00:00",
        "organizer_id": context.organizer_id
    }

    response = requests.post(
        f"{BASE_URL}/events/",
        headers={"Authorization": context.token},
        json=context.event_payload
    )

    context.response = response
    if response.status_code == 200:
        context.event_id = response.json()["id_event"]

@then("the response should contain the event information")
def step_event_info(context):
    data = context.response.json()

    for field in ["title", "description", "location", "capacity", "price", "start_datetime"]:
        assert field in data, f"Field '{field}' missing in event response"

    assert data["title"] == context.event_payload["title"]

```

### 3.4 Execution Setup

Indicate how to run the Cucumber tests (e.g., Maven or Gradle command, test runner configuration).

## Running Cucumber Tests

To execute the Cucumber tests for this project, you can use the following approaches depending on your build tool and test runner configuration:

- **Using Maven:** Run the tests with the Maven `verify` goal:

```
mvn verify
```

This will compile the project and execute all Cucumber scenarios defined in the `src/test/resources` directory.

- **Using Gradle:** Execute the Cucumber tests using the Gradle `test` task:

```
./gradlew test
```

Ensure that the Cucumber plugin or dependency is properly configured in your `build.gradle` file.

- **Test Runner Configuration:** The tests are organized using a Cucumber runner class (e.g., `Runner.java` for Java projects). This runner specifies the location of the feature files and the step definitions. Make sure the `glue` and `features` paths are correctly set to include all authentication and business module scenarios.

By following these steps, you can run all acceptance tests and verify that the backend functionality behaves as expected.

## Running Cucumber Tests with Python Behave

For the Python backend, acceptance tests are implemented using the `behave` framework, which follows the Gherkin syntax similar to Cucumber. To run the tests, follow these steps:

1. **Install dependencies:** Make sure `behave` is installed in your virtual environment:

```
pip install behave
```

2. **Navigate to the test directory:** Go to the directory where the `features` folder is located, typically inside your backend module:

```
cd backend/business/tests/acceptance
```

3. **Run the tests:** Execute all feature files using the `behave` command:

```
behave
```

You can also specify a particular feature file:

```
behave list_events.feature
```

4. **Test runner configuration:**

- Step definitions are located in the `steps/` folder corresponding to each feature.
- Make sure the backend service is running (e.g., via Docker) before executing the tests, so that API endpoints are accessible.
- Any context setup, such as test users or database state, should be handled in the `environment.py` file if needed.

By following these instructions, all acceptance tests for the authentication module will be executed and the results displayed in the terminal.

### 3.5 Test Results

Present the main results of the acceptance tests, including:

- Number of scenarios and steps.
- Passed/failed status.
- Known issues or pending scenarios.

First of all, we only have defined 6 scenarios focus on "US-02 — Browse Events", "US-03 — Checkout with Online Payment", "US-04 — Receive Digital Ticket", "US-10 — View My Orders Tickets", "US-06 — Create Event", "US-07 — View Sales Summary" for business logic, due issues with the authentication most of the tests failed because token gives serious problems at the time to create, update, delete or try to do something in the database. However, auth test and business logic are worked good together, failure is on the design of the steps and the definition of the token making it failed or have multiple errors. For pending scenarios, we expect to fix business logic scenarios to integrated correctly the auth steps. In addition, we expect to add auth logic scenarios and features on our final delivery.

## 4 JMeter API Stress Tests

### 4.1 Test Plan Design

The JMeter test plan focuses on testing the main REST APIs of the ProTicket platform, performing load and performance tests with a small but representative workload. The goal is to validate that the system responds correctly to requests, ensure average response times of less than 500ms, verify that the system can handle multiple simultaneous users, and achieve at least 50 requests per second.

At the top level, two user-defined variables are declared: `BASE_URL_AUTH` (e.g. `http://127.0.0.1:8080`) and `BASE_URL_BUSINESS` (e.g. `http://127.0.0.1:8000`), which conceptually represent the base URLs for the auth and business APIs.

The test plan is divided into three thread groups:

- **Thread Group 1: “1. User Registration”**  
Targets the `/auth/register` endpoint (Java auth backend).

- **Users:** 5 threads.
- **Ramp-up:** 10 seconds.
- **Loops:** 1 per thread.
- **Endpoints:** two POST requests to `/auth/register`, one for creating a `buyer` user and one for an `organizer` user.
- **Main metrics:** response time (average and percentiles), success rate and HTTP status codes.
- **Thread Group 2: “2. User Login”**  
Targets the `/auth/login` endpoint to simulate repeated login attempts.
  - **Users:** 5 threads.
  - **Ramp-up:** 15 seconds.
  - **Duration:** 90 seconds (scheduler enabled).
  - **Loops:** 3 repetitions per thread.
  - **Endpoint:** POST `/auth/login` with valid buyer credentials.
  - **Main metrics:** latency, throughput (logins per second), error rate and extraction of the JWT token from the response body.
- **Thread Group 3: “3. Get Events”**  
Exercises the main read endpoint of the Python business backend.
  - **Users:** 10 threads.
  - **Ramp-up:** 30 seconds.
  - **Duration:** 120 seconds (scheduler enabled).
  - **Loops:** 5 per thread.
  - **Endpoint:** GET `/events/` on the business API (port 8000).
  - **Main metrics:** response time, throughput (requests per second), and error rate under a moderate, sustained load.

Overall, the workload models a small cohort of concurrent users who first register, then perform repeated login attempts and finally query the list of events. The main metrics of interest are throughput, latency (average and percentiles) and HTTP error rate.

## 4.2 Implementation Details

The JMeter test plan is defined in a single `.jmx` file and uses the standard JMeter components to simulate realistic API usage:

- **User-defined variables** (`BASE_URL_AUTH`, `BASE_URL_BUSINESS`) to centralise the base URLs of the services.

- **Thread groups** for each logical flow: registration, login and event browsing. Each thread group has its own configuration of threads, ramp-up time, loop count and scheduling.
- **HTTP samplers** that send JSON requests to the appropriate endpoints:
  - POST `/auth/register` with a JSON body that dynamically generates unique emails using functions like `_${__threadNum()}` and `_${__time()}`.
  - POST `/auth/login` with static buyer credentials.
  - GET `/events/` for the business API.
- **HTTP Header Managers** to set `Content-Type: application/json` for the JSON-based requests.
- **JSON Extractor** to capture the JWT token from the login response (`$.token`) and store it in a JMeter variable (`TOKEN`) for potential reuse in subsequent requests.
- **Constant Timer** in the “Get Events” thread group (2 000 ms delay) to introduce a realistic pause between consecutive requests and avoid purely synthetic, back-to-back calls.
- **Listeners:** “View Results Tree”, “Aggregate Report”, “Aggregate Graph” and “Summary Report” are configured to collect metrics such as response times, latency, throughput, bytes sent/received and error counts.

The combination of these elements provides a realistic but manageable test plan that can be easily understood and modified.

### 4.3 Execution Environment

The stress tests were executed against the Dockerised version of the system. The Java auth backend, the Python business backend and both databases (MySQL and PostgreSQL) were started using the main `docker-compose.yaml` file from the `ops` folder.

JMeter itself was run in non-GUI mode inside a dedicated Docker container using the `justb4/jmeter:latest` image, mounting the test plan and output directory:

```
docker run --rm \
-v "$PWD/tests":/tests \
-w /tests \
justb4/jmeter:latest \
jmeter -n -t proticket.jmx -l results.jtl -e -o report
```

The tests were executed on a typical development laptop (mid-range CPU and 8 GB RAM), with all containers running locally and no artificial network latency injected. As a result, the measured response times reflect mainly the application and database processing delays, plus a small overhead due to Docker.

## 4.4 Results and Analysis

Under the configured workloads (up to 10 concurrent virtual users and short test durations), the system remained stable and responsive. The Summary Report and Aggregate Report provided by JMeter indicate the following:

- **Average and percentile response times.**

For the registration and login endpoints in the auth service, average response times stayed in the low hundreds of milliseconds, with the majority of requests completing well below one second. The `/events/` endpoint in the business API exhibited even lower latencies, since it mainly performs read operations.

- **Throughput.**

Given the modest number of threads and the constant timer in the “Get Events” group, the throughput is moderate (a few tens of requests per second across all samplers). This level of load is sufficient to validate that the system can handle several concurrent users without degradation, while keeping the test simple enough for an academic setting.

- **Error rates.**

For the tested scenarios and valid input data, the error rate reported by JMeter was effectively 0%. All HTTP responses returned the expected status codes (typically 200 or 201), and no connection failures were observed. Any potential invalid credential tests would appear as planned `4xx` responses rather than unexpected errors.

- **Bottlenecks and potential improvements.**

At this scale, no clear performance bottleneck was detected. The tests behave more like load and smoke tests than extreme stress tests. However, the plan already highlights a few aspects that could be explored in future work:

- Increasing the number of threads and test duration to investigate how the auth service and the databases behave under heavier concurrent load.
- Adding protected business endpoints that require the JWT token (e.g. ticket purchase) to evaluate the full authentication and authorisation chain.
- Introducing additional timers or think times to mimic more realistic end-user behaviour.

In summary, the JMeter tests provide a concise but meaningful view of the performance of the main ProTicket APIs under a small, controlled workload, which is appropriate for the scope of this project.

## 5 GitHub Actions CI/CD Workflow

### 5.1 Workflow Overview

The project includes a continuous integration workflow implemented with GitHub Actions under the name *ProTicket CI*. The goal of this pipeline is to automatically validate the three main components of the system (Java auth backend, Python business backend and React frontend) whenever new code is pushed.

The workflow is organised into four jobs:

- **Auth backend (Java) tests:** runs the Maven build and unit tests for the Spring Boot auth service.
- **Business backend (Python) tests:** installs Python dependencies and executes the test suite for the FastAPI business service.
- **Frontend build (Vite + React):** installs Node dependencies and builds the production bundle of the React frontend.
- **Docker images build:** builds Docker images for the three components to verify that the Dockerfiles are valid.

The last job depends on the three previous ones, so the Docker image build only runs if the code compilation and tests complete successfully.

### 5.2 Workflow Definition

The workflow is defined in the file `.github/workflows/ci.yml`. The main sections of this file are:

- **Triggers:** the workflow is executed automatically on `push` events to the branch `feature/dockerization`, and on any `pull_request`. This allows the CI pipeline to run both for regular development work on the feature branch and for code reviews.
- **Jobs:** each logical component (Java backend, Python backend, frontend, Docker build) is implemented as an independent job running on an `ubuntu-latest` runner. Jobs share a common pattern of steps: `checkout` of the repository, language runtime setup (JDK, Python, Node.js), dependency installation and finally build/test commands.
- **Dependencies between jobs:** the `docker-build-check` job declares `needs` on the three previous jobs so that Docker image builds are only attempted if all tests have passed.

At this stage no external secrets are required. All configuration is embedded in the repository: the workflow does not yet log in to any container registry and the Docker images are only built locally on the GitHub runner for validation purposes.

### 5.3 Automated Testing Stage

The automated testing stage is split into two jobs: one for the Java auth backend and one for the Python business backend.

- **Auth backend tests:**

- Working directory: `app/backend/auth`.
- The job uses `actions/setup-java` to install JDK 17 (Temurin distribution) and enable Maven dependency caching.
- The main step runs `mvn -B test`, which compiles the Spring Boot application and executes the JUnit test suite. This job is ready to include Cucumber acceptance tests once they are integrated into the Maven build.
- If the Spring Boot test context fails to start (for example due to missing database configuration), the job is marked as failed, and the subsequent Docker build job will not run.

- **Business backend tests:**

- Working directory: `app/backend/business`.
- The job uses `actions/setup-python` to install Python 3.11.
- It installs the FastAPI service dependencies via `pip install -r requirements.txt`.
- Tests are executed with `pytest`. If no tests are present yet, the command is wrapped with `|| echo "No Python tests defined yet"` so that the job still completes while the test suite is under construction.

Test failures are reported directly in the GitHub Actions UI: the corresponding job appears in red, and the failing step shows the Maven or Pytest logs, which can be inspected to diagnose configuration or code issues.

### 5.4 Docker Image Build and Publishing

The last job of the workflow, `docker-build-check`, validates the Dockerfiles of the three main components. This job only starts if all testing jobs succeed.

- The job checks out the repository and runs three Docker build commands:

- `docker build -t proticket/auth:ci app/backend/auth`
- `docker build -t proticket/business:ci app/backend/business`
- `docker build -f Dockerfile -t proticket/frontend:ci app/frontend`

- Each image is tagged with a local `:ci` tag, used purely for validation inside the CI pipeline.

At this stage the workflow does *not* push images to a remote container registry (such as Docker Hub or GitHub Container Registry). However, the structure of the job makes it straightforward to extend the pipeline in the future with an additional login step and `docker push` commands to support full CI/CD.

## 5.5 Evidence of Successful Runs

GitHub Actions keeps a history of workflow executions under the `Actions` tab of the repository. For this project, one or more runs of the *ProTicket CI* workflow were triggered from the `feature/dockerization` branch.

For the report, the following artefacts are used as evidence:

- **Screenshots** of the GitHub Actions UI showing the *ProTicket CI* workflow with its four jobs (*Auth backend (Java) tests*, *Business backend (Python) tests*, *Frontend build (Vite + React)* and *Docker images build*), together with their success/failure status.
- **Log excerpts** from the `auth-backend-tests` and `docker-build-check` jobs, illustrating how Maven is executed, dependencies are downloaded, and the Docker build commands are run. These logs can be downloaded from the GitHub Actions interface as a ZIP file and provide detailed evidence of the CI workflow execution.

Even though the configuration is still evolving (for example, the Java tests are being adapted to use an in-memory database in the CI environment), the workflow already automates the key tasks required by the workshop: running backend tests, building the frontend and validating all Docker images in a reproducible way.

## 6 References

### References

- [1] Docker Documentation. Available at: <https://docs.docker.com/>
- [2] Cucumber Documentation. Available at: <https://cucumber.io/docs/>
- [3] Apache JMeter Documentation. Available at: <https://jmeter.apache.org/usermanual/>
- [4] GitHub Actions Documentation. Available at: <https://docs.github.com/actions>