UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

BACHELOR THESIS

TITLE: TESTBENCH: Automated Management and Evaluation Platform of Code Projects in Education

DEGREE: Bachelor's Degree in Network Engineering

AUTHOR: Antoni Boté Ribalta

DIRECTOR: Juan López Rubio

SUBMISSION DATE: 20/06/2025

TITLE: TESTBENCH: Automated Management and Evaluation Platform of Code Projects in Education

DEGREE: Bachelor's Degree in Network Engineering

AUTHOR: Antoni Boté Ribalta

DIRECTOR: Juan López Rubio

SUBMISSION DATE: 20/06/2025

Abstract

The goal of this thesis is to create testBench, a platform designed to support educators in managing, testing, and grading student coding assignments more effectively. As programming courses grow in size and complexity, instructors often struggle with the overhead of setting up environments, reviewing code manually, and ensuring fair, consistent grading. testBench addresses these challenges by offering a system that automates repetitive tasks while supporting clear, objective evaluation.

From an educational perspective, the tool encourages student independence and active learning by providing immediate feedback in a controlled, reproducible environment. It allows students to test their code, learn from mistakes, and iterate-without needing to navigate complex setup processes. Technically, testBench combines containerized environments, automated testing frameworks, and a user-friendly interface to streamline the entire workflow. This thesis explores the development of the tool and its potential to enhance the learning experience in modern computer science education.

Als meus pares i a la meva germana, per la vostra confiança, paciència i suport incondicional. Sense vosaltres, arribar fins aquí no hauria estat possible.

Als amics amb qui he compartit aquests anys, gràcies per ser-hi tant en els bons moments com en els difícils. Espero que seguim fent camí plegats durant molts anys.

Als companys de feina, per tot el que m'heu ensenyat i per l'empenta i la dedicació que sempre heu transmès.

Al meu tutor, per ser una font constant d'idees, entusiasme i passió pels projectes que compartim, i per la confiança dipositada durant tot aquest procés.

A tots plegats, gràcies per acompanyar-me en aquest viatge. Aquest treball també és vostre.

Table of Contents

List of Figures

List of Tables

Glossary

API	Application Programming Interface
ASGI	Asynchronous Server Gateway Interface
CD	Continuos Deployment
CI	Continuos Integration
CLI	Command-Line Interface
CSV	Comma-Separated Values
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
IDE	Integrated Development Enviroment
JSON	JavaScript Object Notation
JUnit	Java Unit Testing Framework
JWT	JSON Web Token
LMS	Learning Management System
ORM	Object-Relational Mapping
REST	Representational State Transfer
UI	User Interface
UML	Unified Modeling Language
XML	eXtensible Markup Language

1 Introduction

In the growing landscape of software development and computer science education, managing student code projects efficiently, fairly, and reproducibly has become increasingly important. Traditional methods of collecting assignments, setting up development environments, and manually grading code not only consume significant time and effort, but are also prone to inconsistencies.

This gap between pedagogical needs and available tools calls for a solution that streamlines the entire workflow, from project management to automated assessment.

The purpose of *testBench* is to be a code project management tool designed with educators in mind. It enables teachers to import and manage student projects, automatically deploy standardized development environments, and run automated tests to assess functionality and correctness. The platform then compiles the results into final grades, facilitating objective and repeatable evaluation.

By combining deployment, testing, and grading into a single platform, *testBench* aims to reduce the overhead associated with manual assessment while promoting consistent evaluation criteria across all student submissions. It is particularly suited for programming courses where reproducibility, automation, and traceability are critical to managing increasing class sizes and maintaining academic integrity.

This thesis outlines the design, development and implementation of *testBench*, discusses the technical decisions behind its architecture, and evaluates its effectiveness as an educational support tool.

1.1 Motivation

When I began my degree in Network Engineering, I wasn't entirely sure what I was getting into. I had always been interested in technology, but the development side of it felt distant and abstract. That changed during my first programming course: something clicked, and I unexpectedly found myself drawn to coding and problem solving in a way I hadn't anticipated.

Over time, and especially through the hands-on experience I gained during my internship, that interest matured into a clear direction. I came to value not just the technical side of software development, but its potential to improve real-world systems — including those within education.

The idea for *testBench* grew out of that realization. While much emphasis is placed on easing the burden of grading for instructors, I became more interested in how an automatic and asynchronous feedback system could directly benefit students. A tool like this hopes to make it easier for students to practice on their own, outside the classroom, and to receive instant feedback - encouraging more frequent self-driven learning.

It also supports broader teaching models, such as virtual or hybrid learning environments, where traditional face-to-face feedback may be limited. Today's students often ask fewer questions in class and increasingly rely on platforms like YouTube or ChatGPT to guide their learning. A system like *testBench* meets them where they are, but still requires them to engage actively with the material — submitting code, testing it, and learning from their own mistakes. It helps shift programming exercises from passive observation to hands-on participation.

Another major motivation was the desire to eliminate common technical barriers. Many students struggle to set up local development environments, especially on underpowered or restricted devices. *testBench* avoids these issues entirely by providing a standardized, containerized environment — no need for local tools, no risk of version mismatches, and no performance bottlenecks. This also makes it ideal for controlled situations like exams, where internet access may be re-

stricted and system configuration must remain uniform.

Developing *testBench* was an opportunity to apply what I've learned to a real educational challenge — and to give something back to the academic environment that helped shape my journey.

1.2 Project's goals

An overview of the goals that were set for this project:

- Design and implement a backend architecture capable of managing multiple student coding projects efficiently and reliably.
- Develop a user-friendly frontend interface that simplifies the interaction for instructors and provides clear feedback.
- Integrate DevContainers to ensure consistent, reproducible development environments across all student submissions.
- Enable automated testing using frameworks like JUnit to assess the correctness and functionality of student code.
- Automate the grading process and compile results into final scores, supporting objective and scalable evaluation.
- Provide tools for easy project import, allowing integration with existing course workflows.
- Ensure the platform is modular, extensible, and maintainable for future enhancements or adaptations.
- Address security, performance, and usability considerations to support real-world classroom use.

2 Context

This chapter sets the stage for the project by exploring the background and challenges associated with managing and grading student programming projects.

It begins by highlighting the key problems faced by educators in this area, followed by a review of existing tools. The chapter then introduces the concept and architecture of *testBench* as a solution.

2.1 Background of the problem

Software development in computer science education has increasingly presented challenges in managing and evaluating student projects. Traditionally, instructors have relied on manual processes to collect, configure, and grade programming assignments. These processes often involve handling various project submissions, setting up individualized development environments, and performing manual tests, tasks that are time consuming and error-prone [1].

In addition, the growing class sizes in programming courses exacerbate these challenges, making it difficult to maintain consistent grading standards and ensure fairness. Therefore, the need for a comprehensive solution that automates and standardizes the workflow from project import to environment deployment, testing, and grading has thus become apparent [2]. Addressing this gap is critical to improving the efficiency, objectivity, and scalability of programming education, ultimately improving both instructor and student experiences.

testBench was conceived to meet this need, providing a centralized platform that supports reproducible development environments and the automated and consistent evaluation of student code projects.

2.2 Overview of existing solutions

As previously noted, several existing solutions offer functionalities similar to those of *testBench*. In fact, *testBench* draws inspiration from these platforms, making it important to briefly review their features and limitations. This section outlines some of these existing solutions.

2.2.1 LeetCode

LeetCode is a popular on-line platform primarily designed to help people practice coding problems and prepare for technical interviews [3]. It offers a vast collection of algorithmic challenges with instant automated feedback on correctness and performance. LeetCode excels at individual coding exercises and automated evaluation, but is not tailored to managing full student projects or providing reproducible development environments.

2.2.2 GitHub Classroom

GitHub Classroom integrates with the GitHub platform to facilitate assignment distribution, student repository management, and collaboration [4]. It enables educators to create and distribute coding projects through version-controlled repositories, providing a familiar workflow for students. In addition, it supports integration with CI tools to automate testing and classification.

2.2.3 Gradescope

Gradescope is an assessment platform used in academia that supports a variety of assignment types, including programming projects [5]. It offers features such as automated code grading, manual grading assistance, and detailed feedback tools. Gradescope improves grading efficiency and consistency.

2.2.4 Repl.it Classroom

Repl.it Classroom is a cloud-based coding environment that allows educators to assign coding projects and exercises with instant access to interactive, browser-based development environments [6]. It provides real-time collaboration and automated grading features. Its cloud-native approach simplifies environment setup by abstracting configuration details away from the user.

2.3 Introducing *testBench*: concept & architecture

The central concept behind *testBench* is to provide educators with a tool that reduces the administrative burden of managing numerous student submissions, while ensuring that projects are executed in standardized, isolated environments. This approach enhances fairness and consistency by eliminating the variability in local setups that often leads to discrepancies during evaluation.

testBench is designed as an integrated platform that supports the entire programming course workflow. It spans from course and user management to project handling, code editing, and automated testing. By unifying these functions, it streamlines the responsibilities of educators, allowing them to focus more on teaching rather than logistic tasks.

Instructors can create courses, assign exercises, monitor student progress, and review code submissions all within a single application. The automated testing features provide timely, objective feedback, enabling scalable and reproducible assessment. Meanwhile, students can access the exercises assigned to their courses, edit and save their submissions, and deploy their code to receive automatic grading.

The following figure illustrates the key features and workflow possibilities offered by *testBench*.

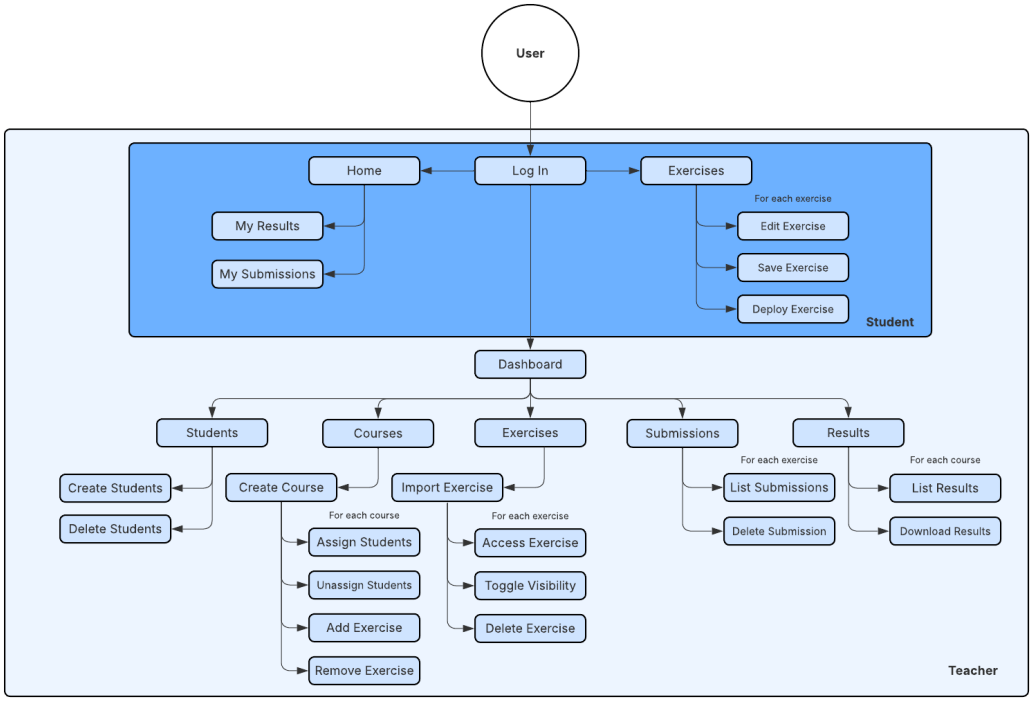

Figure 2.1: Structural overview of *testBench*

Figure 2.1 summarizes the overall structure and intended workflow of *testBench*, clearly distinguishing between the roles of students and teachers. On the student side, the system focuses on usability and autonomy, allowing users to log in, access exercises, submit code, and receive feedback in a streamlined and self-directed way. Teachers, meanwhile, are given comprehensive

tools to manage students, courses, and exercises, as well as to track submissions and results through a centralized dashboard.

This combination of features reflects the platform's core purpose: to reduce friction for both students and instructors, while supporting scalable, asynchronous programming education. By minimizing environment setup issues, standardizing evaluation, and encouraging active student participation, *testBench* aims to improve learning outcomes and make practical programming assignments more accessible in modern academic contexts.

With this context established, the following chapters will explore the design, implementation, and deployment of the system in more detail.

3 Design & Architecture

In this chapter, we will explore the design of *testBench* in greater detail, along with the architecture that connects the backend and frontend components.

Additionally, we will examine the UML diagram to explain how the app's data is structured and interrelated. Finally, we will take a closer look at the various services offered by the application.

3.1 Architecture Overview

As shown in the figure below, *testBench* follows a three-tier architecture [7]. On the left, the client tier is represented by a laptop, which signifies the frontend interface through which users interact with the system. This interface sends requests to and receives responses from the backend.

On the right, the backend and the database are represented. The first section represents the API layer, which serves as the entry point for client requests, handling routing, validation, and interfacing with internal services. The second section encompasses the application server and database, responsible for executing business logic and managing persistent data.

The specific technologies chosen for each component of this architecture will be discussed in detail in later chapters.

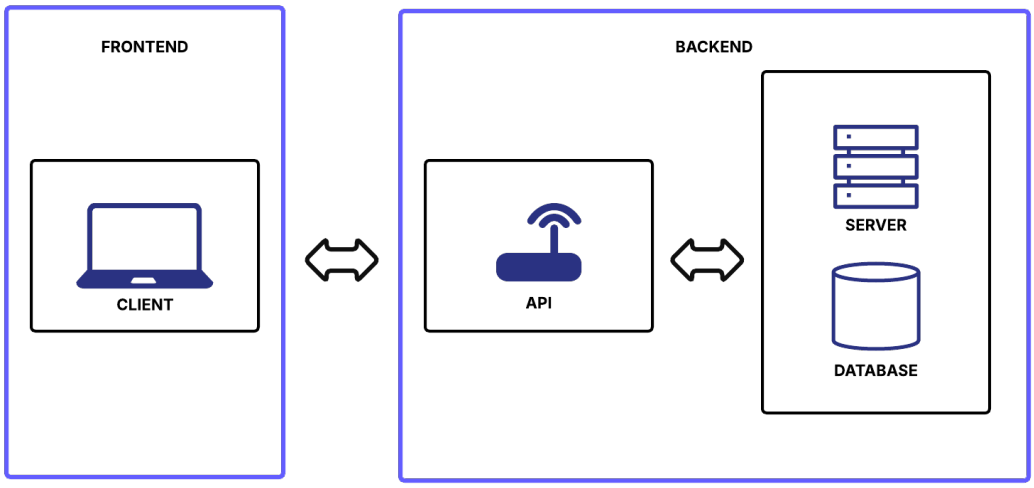

Table 3.1: Architecture of *testBench*

3.2 UML Diagram

Up next is the UML Class Diagram [8], which illustrates the structural design of the application's core data models and the relationships that connect them.

This diagram serves as a blueprint of the backend domain, providing a high-level overview of how the system organizes, stores, and interacts with data.

The diagram visualizes the primary entities within the system—`Course`, `Exercise`, `Submission`, and `User`—each represented as a class with its respective attributes. It also highlights the associations between these entities, such as the way users interact with exercises, how exercises belong to specific courses, and how submissions are linked to both users and exercises.

Furthermore, it includes supporting data structures like `ExerciseFile`, `ExerciseTest`, `SubmissionFile`,

and `TestResult`, which are tightly coupled with their parent entities through composition relationships.

By clearly laying out these components and their interconnections, the UML Class Diagram provides a structured and coherent representation of the data model. This aids in understanding the system's design, ensures consistency during development, and serves as a reference point for maintaining and extending the application in the future.

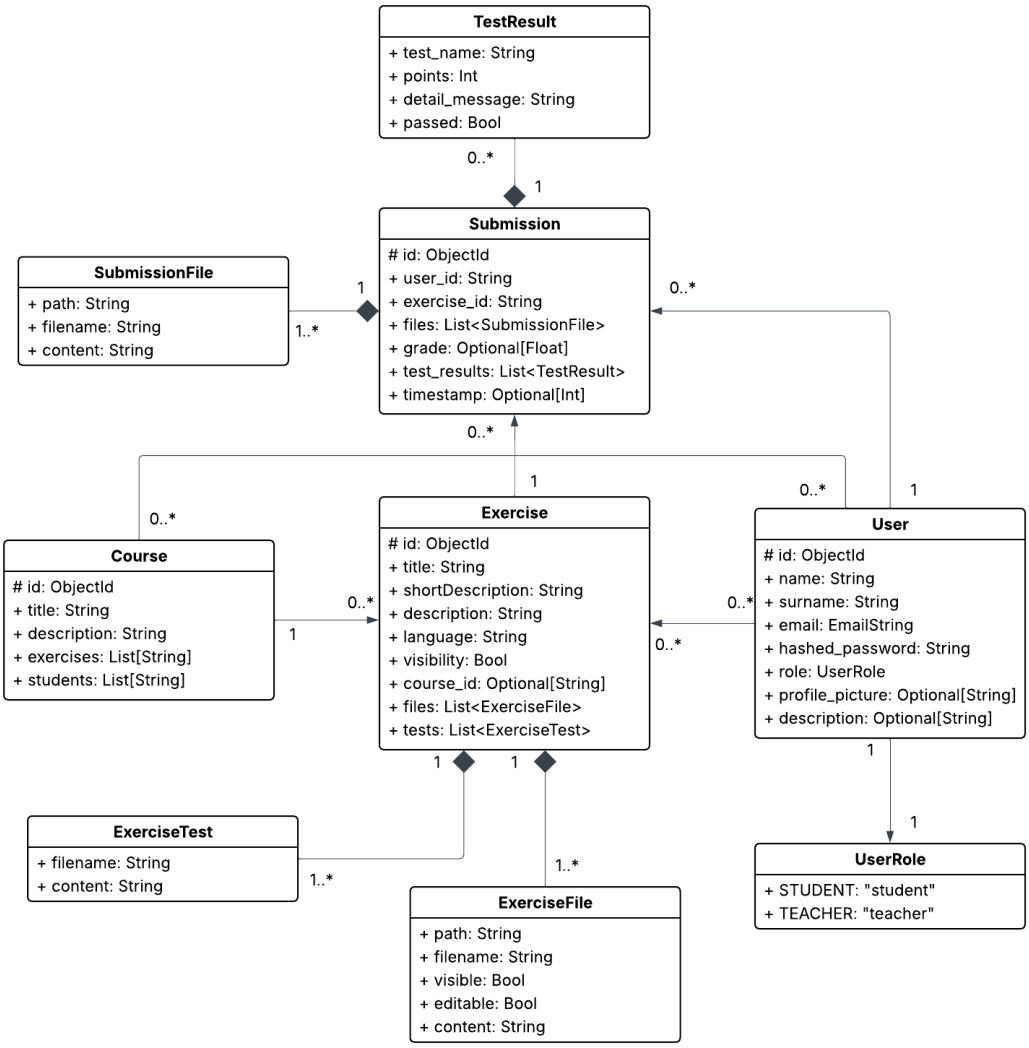

Table 3.2: UML Class Diagram

3.3 Services

The next step in this chapter is to enumerate the primary services offered by the application. These services are exposed through a REST API [9] —a set of conventions that facilitate communication between systems over HTTP.

In the following sections, the services will be organized according to the data models they pertain to or interact with most directly. For each group, the most relevant functions are outlined, along with the corresponding routes they utilize.

3.3.1 Users

Firstly, we will go over the services corresponding to the User. These include the functions responsible for user registration, authentication, profile management, and user-specific operations such as retrieving user details or deleting student accounts.

These endpoints facilitate core user interactions with the application and are protected via authentication tokens [10] where appropriate.

POST	/register	Registers a new user, ensuring email uniqueness, storing hashed credentials, and returning an access token.
POST	/login	Authenticates a user by verifying their credentials and returns an access token if successful.
POST	/create_student	Allows a teacher to create a new student account. Sends the student an email with their login credentials.
POST	/upload-profile-picture	Allows users to upload a profile picture.
GET	/me	Retrieves the current authenticated user's information based on the provided access token.
GET	/students	Returns a list of all users with the "student" role. Accessible only by users with the "teacher" role.
PUT	/update-profile	Updates the authenticated user's profile data, such as name, surname, password, and description.
DELETE	/delete/{student_id}	Deletes a student account and associated submissions. Operation restricted to users with the "teacher" role.

Table 3.3: Routes for user operations

3.3.2 Exercises & Submissions

Next, we examine the services corresponding to the Exercise and Submission models.

These endpoints handle the creation, retrieval, updating, and deletion of exercises, as well as the management of code submissions by students. Also, the routes are secured and role-restricted where necessary to ensure appropriate access and maintain data integrity across users.

POST	/import_project	Imports an exercise project from a locally stored ZIP file.
POST	/import_github	Imports an exercise project from a GitHub repository URL.
POST	/save/{exercise_id}	Saves or updates a submission for a specific exercise.
GET	/	Retrieves all exercises.
GET	/{exercise_id}	Retrieves details for a specific exercise.
GET	/my_submissions	Retrieves all submissions made by the authenticated user.
GET	/submission/{exercise_id}	Retrieves the current user's submission for a specific exercise.
GET	/search	Searches exercises by title using a case-insensitive match.
GET	/search_courses	Searches courses by title using a case-insensitive match.
PUT	/update/{exercise_id}	Updates an existing exercise. Only accessible by users with the "teacher" role.
PUT	/{exercise_id}/remove_course_id	Removes the course_id from the specified exercise.
DELETE	/delete/{exercise_id}	Deletes an exercise and its submissions.
DELETE	/submission/delete/{submission_id}	Deletes a specific submission. Only for teachers.
PATCH	/{exercise_id}/visibility	Updates the visibility of an exercise.

Table 3.4: Routes for exercise and submission operations

3.3.3 Courses

We now turn our attention to the services related to the Course model. These endpoints support the creation, management, and association of courses within the platform.

Teachers can create and update courses, assign exercises and students, and retrieve detailed information about enrolled students or linked exercises. In addition, courses can be modified or deleted, and exercises or students can be dynamically assigned or unassigned.

These operations enable a structured and scalable approach to managing academic content and

participation found in *testBench*.

POST	/create	Creates a new course.
POST	/{course_id}/assign_exercise/{exercise_id}	Assigns an exercise to a course and updates the exercise's {course_id}.
POST	/{course_id}/assign_students	Assigns a list of students to the specified course.
GET	/	Retrieves a list of all courses and their data.
GET	/{course_id}	Retrieves details for a specific course.
GET	/{course_id}/exercises	Retrieves all exercises assigned to a specific course.
GET	/{course_id}/students	Retrieves the list of students assigned to the course.
GET	/results/by_course/{course_id}	Retrieves students' grades for all exercises in a course.
GET	/results/me/{course_id}	Retrieves the authenticated student's grades for a course.
PUT	/update/{course_id}	Updates the title and/or description of a course.
DELETE	/delete/{course_id}	Deletes a specific course by ID.
DELETE	/{course_id}/remove_exercise	Removes a specific exercise from a course using a query parameter.
DELETE	/{course_id}/students/{student_id}	Unassigns a specific student from a course.

Table 3.5: Routes for course operations

3.3.4 Deployment

This section addresses a specialized subset of services responsible for automating the execution and testing of code submissions.

Unlike previous routes that map directly to persistent data models, these endpoints set up the environment in which student submissions are compiled, executed, tested, and graded. It is important to note that services such as GUI container support are still in development and may not be fully enabled in production deployments.

POST	/{exercise_id}	Executes a submission inside a devcontainer, runs tests, and grades the result.
POST	/stop/{exercise_id}	Stops a running devcontainer instance for a submission.
POST	/start_gui/{exercise_id}	Launches a GUI-enabled devcontainer environment for the given exercise. (experimental)
POST	/stop_gui/{exercise_id}	Stops and removes the active GUI container linked to a submission. (experimental)
WEBSOCKET	/ws/logs/{exercise_id}	Establishes a WebSocket connection to stream real-time logs during execution.

Table 3.6: Routes for deployment operations

This chapter has outlined the main structural elements of *testBench*, including its overall architecture, data model, and service organization. Each part has been designed to support the core workflows required by the application—handling users, exercises, submissions, and course management, as well as enabling automated deployment and testing.

With the system design in place, the next step is to look at how these ideas were implemented in practice, and how the different components come together in the working application.

4 Implementation

This chapter outlines the practical implementation of *testBench*, covering the technologies and tools used throughout the backend, frontend, and deployment stages. Each section highlights the implementation decisions made to support functionality, maintainability, and scalability within the system.

4.1 Backend

First, we delve into the backend, the core component responsible for managing the application's logic and orchestrating all the services outlined in the previous chapter. This section aims to justify the selection criteria behind each of the technologies used in its implementation.

These decisions were guided by several key factors, including performance, scalability, maintainability, and ease of integration within an asynchronous and microservice-friendly environment.

4.1.1 Choosing the Right Framework

Selecting the appropriate technology stack was a critical step in the development of the project. For *testBench*, the goal was to adopt a framework that offered simplicity in implementation, clarity for long-term maintainability, and strong compatibility with the libraries and tools anticipated for use later in the development process.

A framework is a reusable, structured set of tools, libraries, and conventions that provides a foundation for developing software applications [11]. It defines the architecture and flow of control for a particular type of application—such as web services, mobile apps, or desktop programs—while allowing developers to focus on implementing the specific functionality of their project.

4.1.2 FastAPI

FastAPI [12] is a modern, high-performance web framework for building APIs with Python [13], based on standard Python type hints. Developed by Sebastián Ramírez, FastAPI is built atop two foundational components: Starlette [14], which provides the asynchronous web toolkit, and Pydantic [15], which handles data validation and serialization using Python type annotations.

FastAPI offers several key features that distinguish it from traditional web frameworks:

1. **Fast to run:** It is one of the fastest Python frameworks available, with performance on par with Node.js and Go, thanks to its asynchronous capabilities via ASGI (Asynchronous Server Gateway Interface) [16].
2. **Fast to code:** It reduces development time by 200–300% through concise syntax, automatic documentation, and intelligent validation.
3. **Automatic interactive documentation:** It automatically generates OpenAPI-compliant docs via Swagger UI and ReDoc, allowing developers and users to test endpoints directly from the browser.
4. **Type-based validation and serialization:** FastAPI uses standard Python type hints to validate input data and serialize output responses, providing immediate feedback to developers and consistent behavior in production.
5. **Editor support and developer ergonomics:** Because it is built on type hints, FastAPI supports auto-completion, type checks, and in-line documentation in IDEs like VS Code and PyCharm, which enhances developer productivity and reduces bugs.
6. **Security and authentication tools:** It includes built-in utilities for handling OAuth2, JWT

tokens, API keys, and other forms of secure access control.

7. **Dependency injection system:** FastAPI includes a powerful and extensible dependency injection system that enables clean separation of concerns, modular design, and testability.
8. **Built-in support for WebSockets and background tasks:** FastAPI provides first-class support for real-time communication via WebSockets and the execution of background jobs during request handling.

While FastAPI offers many advantages, it is important to also consider its limitations during the technology selection process. The following are some of the most notable disadvantages associated with using FastAPI:

1. **Relatively young ecosystem:** Compared to more established frameworks like Django or Flask, FastAPI has a smaller ecosystem and a shorter track record, which can lead to fewer third-party extensions and community-driven tools.
2. **Steeper learning curve for beginners:** While FastAPI simplifies many tasks through type annotations and dependency injection, these concepts—especially asynchronous programming and Pydantic models—can be challenging for developers new to Python or web development.
3. **Complexity with advanced features:** Features such as custom dependency injection, middleware, or complex async workflows can become difficult to manage in larger applications without careful architectural planning.
4. **Limited support for synchronous tooling:** FastAPI is optimized for asynchronous operation. Developers using libraries or tools that are strictly synchronous may encounter compatibility challenges or reduced performance.
5. **No built-in ORM or admin interface:** Unlike Django, FastAPI does not come with an integrated ORM or admin dashboard. This provides flexibility but also means that developers must choose, integrate, and configure such components manually.
6. **Less mature documentation for edge cases:** While the official documentation is excellent for common use cases, coverage of edge cases and advanced usage patterns can sometimes be limited, requiring developers to explore source code or community discussions.

Taking into consideration all the factors discussed above, FastAPI emerged as the most suitable choice for the backend development framework. Its widespread adoption, rapid development capabilities, high performance, and robust dependency injection system aligned closely with the project's requirements. While some drawbacks exist, none were significant enough to warrant choosing an alternative framework. No other framework was seriously considered as an alternative.

4.1.3 Database

In terms of database selection, the choice of MongoDB [17] was straightforward. This decision was primarily driven by extensive prior experience with the technology, which ensured a high level of familiarity and confidence during the development of *testBench*.

Beyond familiarity, MongoDB offers a robust set of features that align well with the needs of the platform. As a document-oriented NoSQL database, it provides a flexible, schema-less data model that accommodates the varied and nested data structures required by *testBench*. Each exercise, for instance, includes multiple files—such as test cases, configuration files, and code statements—organized hierarchically through paths and filenames. MongoDB's ability to efficiently store and retrieve this type of structured data without complex joins or rigid schemas proved to be a natural fit.

Furthermore, MongoDB is designed to handle large volumes of data, an important consideration for an educational platform expected to manage potentially hundreds of student submissions, each with unique file structures and metadata. Although the project does not require advanced features such as geospatial indexing or full-text search, it benefits significantly from MongoDB's powerful query engine, horizontal scalability, and built-in replication mechanisms.

Another relevant factor is the need to manage interdependencies between collections. For example, deleting a course must also remove references to that course in related exercises. While MongoDB does not enforce referential integrity natively, its flexibility allows these relationships to be maintained through well-structured application logic, ensuring data consistency across the platform.

It is also worth noting that, instead of installing MongoDB locally, the decision was made to use its official Docker image, which is actively maintained and well-documented. This approach proved to be more efficient, portable, and easier to integrate into the development workflow. Additionally, in cases of database corruption or misconfiguration during development, the Dockerized setup allowed for quick and seamless resets by simply rebuilding the container, greatly improving the development experience and reducing downtime.It was consistently applied across both development and production environments.

4.1.4 Docker

Docker [18], which plays a central role across the app, was chosen for *testBench* as a modern and reliable tool for containerizing applications. Its core advantage lies in its ability to create consistent, isolated environments that package an application together with all its dependencies, eliminating the variability often introduced by differences between development and production systems. This consistency was especially valuable for a project like *testBench*, which depends on reproducible behavior across different setups and contributors.

Docker also contributes significantly to the maintainability and long-term sustainability of the project. Configuration files such as Dockerfile and docker-compose.yml serve as executable documentation for the entire application environment. This not only streamlines onboarding for future developers but also reduces the likelihood of configuration errors, especially in complex setups involving multiple services. As the project evolves, these files make it easier to update or extend the system in a controlled, repeatable manner. Moreover, their compatibility with CI/CD pipelines allows for seamless automation in testing, deployment, and version management—making Docker a future-proof choice for ongoing development and scaling of *testBench*.

While specific implementation details such as base images used, networking configuration, and service interconnection are discussed in later sections, the decision to adopt Docker was ultimately driven by its ability to streamline development, increase reliability, and support the goals of reproducibility, modularity, and automation that are central to *testBench*.

4.1.5 Development Containers

One of the most critical technology choices in the development of *testBench* was selecting the mechanism used to automatically deploy, test, and grade student code submissions in isolated environments. Early in the design phase, Development Containers emerged as a natural fit—not only for local development consistency, but also as a foundation for the automated runtime environments required by the platform.

DevContainers are a specification and tooling standard that allows developers to define complete, reproducible development environments using simple configuration files [19]. By describing dependencies, language runtimes, tools, and editor extensions in a devcontainer.json file, these environments can be instantiated on-demand in a controlled and isolated way. While initially designed for improving developer experience in editors like Visual Studio Code, their capabilities go

far beyond that.

In the context of *testBench*, DevContainers provide a secure, containerized workspace that mirrors real-world development environments. This made them ideal for executing student code: they offer system-level isolation, reproducibility, and the flexibility to support multiple languages and build systems—key requirements for a platform that aims to automate testing and grading across diverse assignments. Their integration with Docker ensures compatibility with the rest of the system architecture, and their ability to define complex configurations declaratively aligns well with the platform's focus on traceability and automation.

The decision to adopt DevContainers was further reinforced by their developer-friendly nature, modularity, and future extensibility. As a result, they became a cornerstone in the implementation of *testBench*'s core feature: reliably executing and evaluating user-submitted code in standardized, isolated containers.

Further details on how DevContainers are integrated into the system will be presented in the implementation sections that follow.

4.2 Frontend

Similar to the previous section, this one focuses on the key factors that influenced the selection of the frontend development framework.

In this case, the chosen technology is a cross-platform solution capable of generating applications that function on both mobile devices and web browsers.

Although *testBench* was initially intended for use on desktop browsers, completely ruling out the possibility of future support for mobile platforms felt like an unnecessarily limiting decision at this stage of development.

4.2.1 Flutter

Flutter is an open-source UI software development framework created by Google [20]. It enables developers to build natively compiled applications for mobile, web, desktop, and embedded devices from a single codebase, using the Dart programming language.

Dart, is an object-oriented, class-based programming language also developed by Google [21]. Its syntax is clean and familiar to developers experienced in JavaScript or Java, and includes modern language features such as null safety, asynchronous programming with async/await, and strong static typing, which makes it particularly well suited for building high performance maintainable user interfaces.

Like any framework, Flutter comes with a few trade-offs that were carefully considered during the decision-making process. The most notable advantages Flutter provides are:

1. **Cross-platform development:** A single codebase supports Android, iOS, web, and desktop platforms, reducing development time and maintenance effort.
2. **Fast development with hot reload:** Developers can instantly view changes in the UI, significantly speeding up testing.
3. **Deep widget catalog:** Flutter offers a comprehensive collection of pre-built, composable widgets for layout, input, animation, and platform-specific UI elements, enabling rapid UI development with high flexibility and full control over design.
4. **High Performance:** Flutter uses its own custom rendering engine, which draws every pixel on the screen, giving developers full control over the UI and ensuring consistent behavior across platforms.
5. **Ecosystem support from Google:** Flutter and Dart are actively developed and maintained

by Google, ensuring long-term viability, regular updates, robust documentation, and integration with other tools.

Among the disadvantages that impacted the choice:

1. **Low adoption of Dart:** Compared to more established languages like JavaScript, Dart has a smaller ecosystem, which raised questions about long-term support, community resources, and the availability of third-party libraries.
2. **Weighty applications:** Another factor weighed during the evaluation phase was Flutter's tendency to produce larger application binaries, especially when compared to natively developed apps. This is largely due to the inclusion of the Flutter engine and rendering framework in every build. While this trade-off comes with benefits like consistent performance and UI across platforms, it was recognized as a potential limitation, particularly if *testBench* were to be deployed on mobile devices or in environments where download size and storage space are critical concerns.

After carefully weighing both the advantages and disadvantages, the decision to adopt Flutter as the frontend development framework for *testBench* became evident. The framework's extensive widget catalog and high-performance rendering engine were key factors, offering the flexibility and efficiency needed for building a responsive and maintainable UI. Additionally, Flutter's cross-platform capabilities aligned well with the long-term goal of potentially expanding *testBench* to mobile platforms without the need to rewrite the codebase. Prior experience with Flutter also contributed to the decision, as it enabled faster development and aligned with the broader objective of maintaining a consistent technological stack across related projects.

4.3 Component Overview

This section provides a detailed breakdown of the functional components that make up the operational workflow of *testBench*. Given the process-rich architecture of the platform, analyzing each component in isolation facilitates a clearer understanding of the system as a whole. The following subsections are organized to mirror the typical user journey, from course and exercise management to code submission, execution, testing, and grading, highlighting the responsibilities and interactions of each component within that sequence.

4.3.1 Course, User, Exercise & Submission Management

Before diving into the more technical components of the workflow, it is essential to understand how the platform organizes and manages its core entities: users, courses, exercises and submissions. These four data models form the structural foundation of the system and define how users interact with the platform.

This management layer is responsible for handling user accounts and roles, structuring exercises within courses, and providing instructors with tools to create, edit, and organize learning content.

4.3.1.1 User Management

testBench implements role-based access control to differentiate between two types of users: teachers and students. Each role has specific permissions that define what actions the user is authorized to perform.

For better management, the only way a student can access the platform is if its teacher has created an account for him. That can be done by importing a csv with the students emails, at that point an account will be generated for them, and the credentials sent to their email. Once there they will be able to log in, and even change their password if the wish to do so.

To ensure structured access and maintain control over the learning environment, students can

only join the platform through teacher initiation. Specifically, teachers are responsible for creating student accounts. This is typically done by importing a CSV file containing the students' email addresses. Upon import, the system automatically generates individual accounts and credentials for each student. These credentials are then sent to the respective email addresses, allowing students to log in for the first time. After logging in, students have the option to change their password to one of their choosing, ensuring both security and user autonomy.

Role-based access is enforced throughout the platform, with clear distinctions between the permissions granted to teachers and students. While teachers inherit all the capabilities available to students, they also have access to an exclusive administrative dashboard. This dashboard allows them to perform a variety of privileged actions, including importing exercises, creating and managing student accounts, assigning exercises to courses, and enrolling students into those courses.

Students, in contrast, have restricted access. They can only view the courses they have been explicitly assigned to and interact with the exercises within those courses. This separation of privileges ensures a clear boundary between content consumers (students) and content managers (teachers).

The figure below illustrates the user management use cases within *testBench*, highlighting the actions available to each user role.

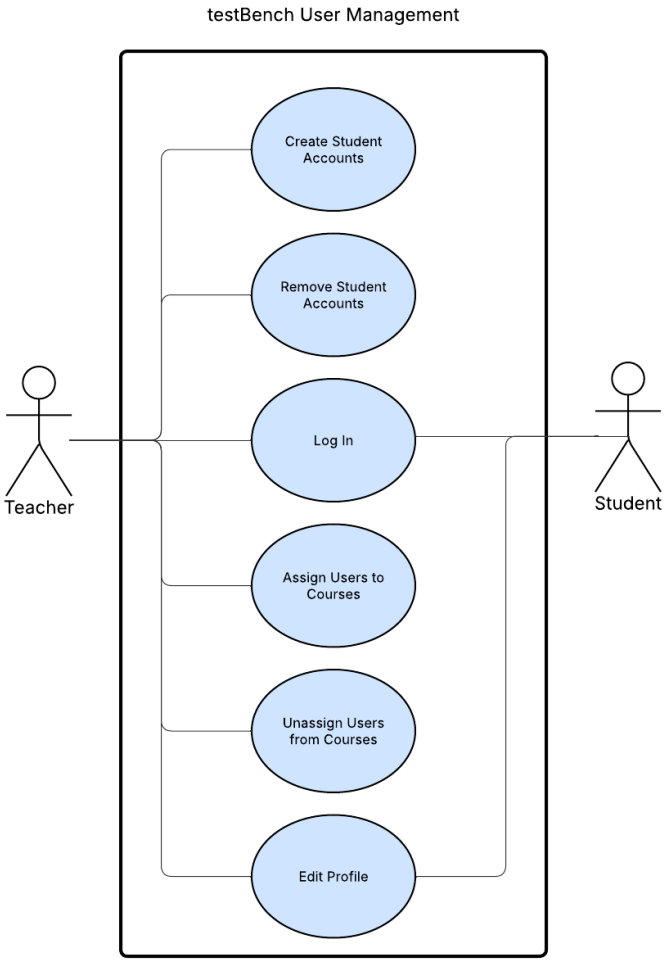

Figure 4.1: Use Case Diagram: User management

4.3.1.2 Exercise Management

The exercise management component governs the creation, organization, and interaction with programming exercises in *testBench*. It serves as the interface through which learning content is defined, structured, and accessed. The design of this component incorporates role-based access control, ensuring that users interact with exercises according to their assigned permissions.

The following subsections outline the key capabilities of the exercise management system.

Importing exercises

Exercises can only be added to the platform through two supported import mechanisms:

- **ZIP archive import:** Teachers can upload a compressed archive containing all the necessary exercise files, including the configuration and metadata.
- **GitHub import:** Teachers can link to a public GitHub repository, which is cloned and parsed by the system.

In both cases, *testBench* extracts the directory structure, reads key configuration files (e.g. exercise.json, devcontainer.json, and Makefile), and stores both metadata and file content in the database. This ensures that every exercise follows a consistent structure suitable for automated execution and grading.

Detail and Editor

Once an exercise has been successfully imported, it becomes accessible through a detailed interface. Initially, exercises appear in a list format, showing their title and short description. Clicking on a specific entry opens the exercise detail page, which serves as a preview.

This detail view presents the core metadata of the exercise, including:

- **Title:** The display name of the exercise.
- **Description:** A short description that contextualizes the exercise's goal.
- **Course:** The course to which the exercise is assigned, if any.
- **Language:** The programming language specified during import.
- **Statement:** The file `statement.md`, which is rendered from Markdown into HTML for easy reading [22].

From this view, the user can click on *"Start Exercise"* to open the code editing interface. This environment mirrors the experience a student would have when solving the exercise.

The editor screen includes several key components:

- **File Navigator Sidebar:** A tree view that replicates the directory structure of the imported exercise. All paths, folders, and files are displayed and can be clicked to open the corresponding file.
- **Code Editor:** A text editor where file contents can be reviewed and edited. Editable files (typically source code files) are fully modifiable, while configuration or metadata files may be read-only.
- **Save Button:** Allows changes to be persisted to the database. When a user saves their work, the current state of the files is stored as a *submission*, uniquely tied to both the `exercise_id` and the `user_id`. This submission is generated from the original exercise files and represents the user's working solution. If a submission already exists for that user-exercise pair, it is updated; otherwise, a new submission is created. The system merges updated files with existing ones to preserve continuity and prevent accidental data loss.
- **Run Code Button:** Triggers the execution pipeline, running the student (or teacher) submission inside a containerized environment.
- **Terminal View:** A live console output area that shows the results of compilation, test execution, or other scripted output from the container.

Assignment to courses

To make exercises available to students, they must be assigned to one or more courses. This is handled through the dashboard, where teachers can:

- Select a course and assign any existing exercises.
- Reassign or remove exercises as needed.

This creates a link between the exercise and the course, allowing students enrolled in that course to access the corresponding content.

Assignment to courses

Teachers can toggle the visibility of an exercise at any time. A visible exercise is accessible to students in the courses to which it has been assigned. When marked as invisible, the exercise remains stored in the system but is hidden from student views. This enables teachers to prepare material in advance, archive outdated exercises, or temporarily disable access during updates.

Access Control and Role-Based Behavior.

Access to exercise management functionalities is governed by user roles:

- **Teachers** can import, delete, and assign exercises, as well as control their visibility.
- **Students** can only view and submit exercises that are assigned to their courses and marked as visible.

This separation of responsibilities ensures that students interact only with relevant content, while teachers maintain full control over the lifecycle of each exercise.

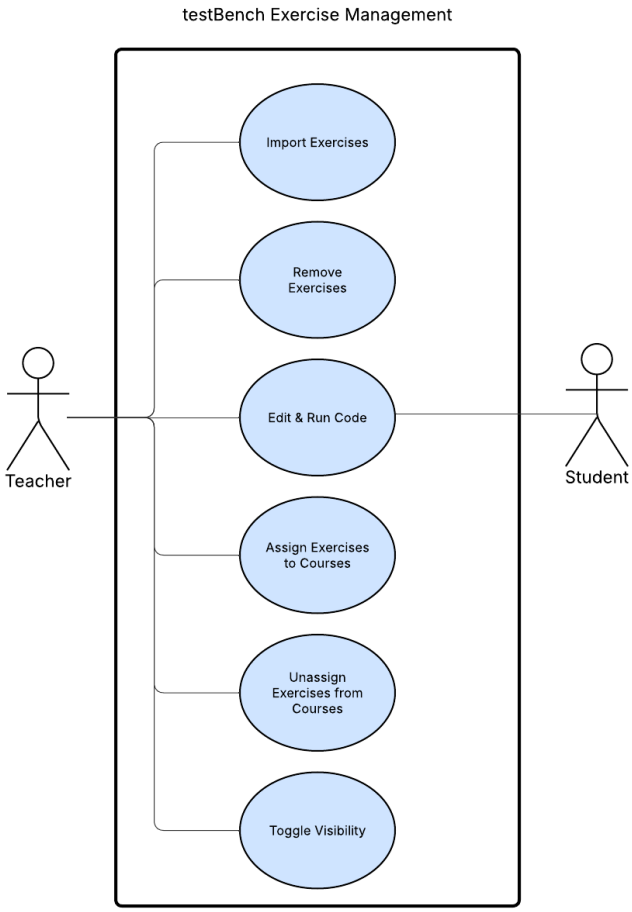

Figure 4.2: Use Case Diagram: Exercise management

4.3.1.3 Course Management

The course management component in *testBench* provides the foundational structure for organizing educational content and access control. Courses act as containers that group exercises and students, defining which users can interact with which exercises. Only users with the teacher role are permitted to create, modify, and assign courses.

Course Creation

Teachers can create new courses by specifying a title and description. Each course is stored with an initially empty list of associated exercises and students.

Exercise Assignment

After exercises have been imported into the platform (as described in the Exercise Management section), they can be assigned to courses. This action establishes a many-to-one relationship: multiple exercises can be linked to a single course, and an exercise may be reassigned. Assigning an exercise automatically updates both the course's list of exercises and the exercise's `course_id` reference.

Exercises can also be removed from a course. This detaches the association without deleting the exercise from the system.

Student Enrollment

Teachers can assign students to a course by selecting them from a list of all existing students in the system. This enrollment process updates the course's internal list of student IDs, defining who will have access to the exercises linked to that course. A student can belong to multiple courses, and unassigning a student from a course is supported via a dedicated endpoint.

Course Retrieval and Listings

Users can retrieve course information depending on their role:

- **Teachers** can view all courses, along with associated students and exercises.
- **Students** can only view the courses to which they are assigned.

Course Deletion

Courses can be deleted by teachers. This operation removes the course from the system and implicitly severs links to all assigned exercises and students. However, it does not delete the exercises themselves or the student accounts.

This modular structure for course management provides clear boundaries of responsibility and enables scalable coordination of exercises across different cohorts or academic terms.

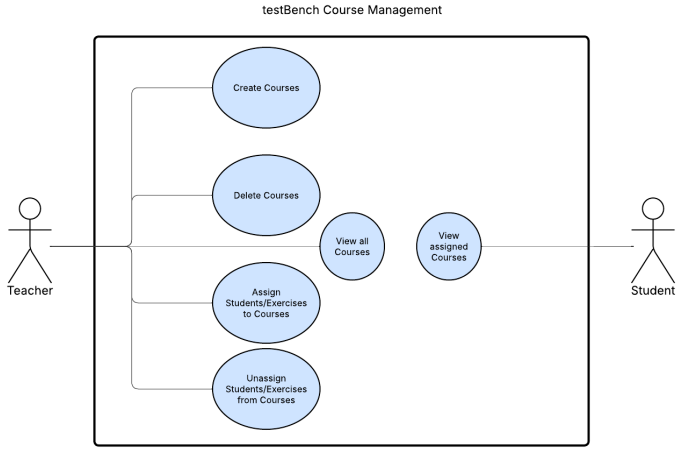

Figure 4.3: Use Case Diagram: Course management

4.3.1.4 Submissions and Results Management

The management of submissions and results is the final step in the exercise workflow and a key feature of *testBench*'s automation capabilities. This component handles the persistence of user submissions, the execution and grading of those submissions, and the display of results for both students and instructors.

Submission Creation

When a user (typically a student) edits and saves their solution to an exercise, a submission is created or updated in the system. Each submission is uniquely identified by a combination of `user_id` and `exercise_id`, and stores:

- The set of submitted files.
- The timestamp of the submission.
- The resulting grade (once executed).

Execution and Grading

Submissions can be executed inside an isolated container environment. During execution, the student's code is compiled and tested against the specifications defined in the exercise (via Makefile and JUnit tests [23]). Each test assigns a score based on a point system defined in the exercise configuration.

Result Access

After execution, students can view the outcome of their submission, including:

- Test case names and descriptions.
- Pass/fail status.
- Points awarded for each test.
- Overall grade.

Instructor Overview

Teachers have access to aggregated results for all students within a course. Through a dedicated results view, they can:

- View all submissions per exercise.
- See student grades and timestamps.
- Filter results by course and exercise.
- Delete individual submissions if necessary.

An additional summary feature calculates the average grade per exercise, allowing instructors to gauge difficulty and student progress.

Privacy and Isolation

Students cannot see each other's submissions or results. All submission data is scoped to the user who created it and is accessible only by them.

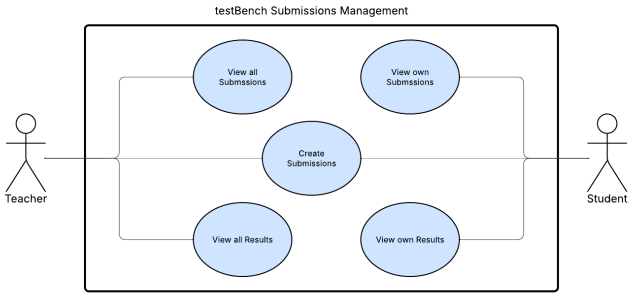

Figure 4.4: Use Case Diagram: Submission management

4.3.2 Exercise Specification

Although not a direct part of the application's source code, the exercises imported into *testBench* are a fundamental component of its functionality. For an exercise to be compatible with the system, it must conform to a specific structure and include certain required files. These files define the behavior, presentation, and assessment logic of the exercise within the platform.

An exercise is expected to be imported as a compressed ZIP archive or via a GitHub repository. In both cases, the directory should contain the following core files:

exercise.json

This file, which has been ideated specifically for *testBench* contains the primary metadata and grading structure for the exercise. It defines:

- `title` – A human-readable title for the exercise.
- `shortDescription` – A brief summary, used in list views.
- `description` – A longer description or problem statement.
- `tests` – An array of test definitions, each specifying:
  - `name`: The name of the test case, expected to match a JUnit test.
  - `points`: The number of points assigned if the test passes.

```
{
  "title": "testBench␣Minimal",
  "description": "This␣exercise␣tests␣the␣Fibonacci␣function␣
     implementation.",
  "shortDescription": "Test␣the␣Fibonacci␣implementation␣in␣Java.",
  "tests": [
    {
      "name": "testFibonacciZero",
      "points": 2,
      "expected_result": "assertEquals(0,␣Main.fibonacci(0));"
    },
    {
      "name": "testFibonacciOne",
      "points": 3,
      "expected_result": "assertEquals(1,␣Main.fibonacci(1));"
    },
    {
      "name": "testFibonacciTwo",
      "points": 5,
      "expected_result": "assertEquals(1,␣Main.fibonacci(2));"
    }
  ]
}
```

Figure 4.5: Sample `exercise.json`

The example shown in Figure 4.5 defines a minimal exercise that tests the accuracy of a Fibonacci function implementation in Java. It includes three test cases— `testFibonacciZero`, `testFibonacciOne`, and `testFibonacciTwo`—each associated with a call to `Main.fibonacci()()` and a corresponding expected output. Points are assigned to each test, enabling partial grading based on how many tests pass during execution.

devcontainer.json

The `devcontainer.json` file defines the containerized environment in which the exercise will be built and tested. It conforms to the official specification provided by Microsoft's Development Containers framework [19], which enables reproducible and isolated environments tailored for development and automation.

This file allows *testBench* to reliably spin up a pre-configured container with all the necessary tools, ensuring consistent execution regardless of the host system. Instructors can customize this environment to match the needs of each exercise.

Key configuration fields include:

- `image` or `dockerFile` – Specifies the base container environment. `image` refers to a prebuilt container (e.g., a Java environment from Microsoft's container registry), while `dockerFile` allows building a custom image locally with fine-grained control over dependencies.
- `mounts` – Defines shared volumes between the host and the container. This is typically used to mount the workspace folder, enabling persistent file access and synchronization.
- `postCreateCommand` – An optional shell command that runs after the container is initialized. It is commonly used to install additional packages, initialize environments, or configure settings needed for the exercise.
- `customizations` – Allows configuration of editor and IDE features, such as preferred extensions, themes, or startup behaviors (especially relevant when used with Visual Studio Code).

```
{
  "name": "Java",
  "image": "mcr.microsoft.com/devcontainers/java:1-21-bullseye",
  "features": {
    "ghcr.io/devcontainers/features/java:1": {
      "version": "none",
      "installMaven": "true",
      "installGradle": "true"
    }
  },
  "workspaceMount": "source=${localWorkspaceFolder},target=/
     workspace,type=bind",
  "workspaceFolder": "/workspace"
}
```

Figure 4.6: Sample `devcontainer.json`

The example shown in Figure 4.6 is taken from the Fibonacci exercise. It uses a prebuilt Java 21 image from Microsoft's Dev Container registry and declares support for Maven and Gradle using reusable features provided via GitHub Container Registry. Binds the local workspace to the container and sets the work directory to `/workspace`. This setup ensures that the environment is fully prepared for compilation and testing, without the need for additional configuration by students or instructors.

Makefile

Each exercise must include a `Makefile` defining how the code is compiled, tested, and optionally deployed. The system expects a `test` target to be defined, as it is invoked during grading to run unit tests and collect results. Below is a sample `Makefile` from the Fibonacci exercise:

Dockerfile (optional)

If a custom image is required, the `devcontainer.json` may reference a `Dockerfile`. This allows the instructor to define a precise development environment by installing additional tools, setting environment variables, or customizing runtime behavior.

statement.md

A Markdown-formatted problem statement that is rendered to HTML within the application. This file typically includes the exercise instructions, examples, input/output formats, and any additional

```
# Makefile

# Default target to run tests
.PHONY: all test clean build deploy

# Build the project using Maven
build:
        mvn clean install

# Run the tests
test:
        mvn test

# Clean the project
clean:
        mvn clean

# Custom deploy target (optional)
deploy:
        echo "Deploying␣application..."
        # Deployment logic here (e.g., Docker, Kubernetes)
```

Figure 4.7: Sample `Makefile`

context needed for the student to complete the task.

Compatibility Requirements

To ensure consistency and automation, *testBench* imposes a specific structure on imported exercises. While the platform is designed to support multiple languages and configurations, full compatibility depends on adherence to a standard layout and the presence of key control files.

Each exercise must include a fixed set of files and folders with defined purposes:

- exercise.json, Makefile, and devcontainer.json are considered core specification files. These define metadata, grading structure, build instructions, and the execution environment. As such, they are marked as non-editable within the platform to prevent accidental misconfiguration.
- Source files (e.g., Java classes, Python scripts) are editable by the user and are intended to contain the student's solution.
- Test files are typically provided by the instructor and should output results in a machine-readable format such as JUnit-compatible XML. These files are visible but not editable by students.
- Additional configuration files (e.g., `Dockerfile`, `.vscode/settings.json`) may be present to support advanced use cases, but their structure must not conflict with the platform's expectations.

The internal file manager distinguishes between student-facing and system-critical files by marking them as either editable or locked. This ensures a clear separation of concerns: students focus on solving the problem, while instructors control the structure and grading logic.

By enforcing this layout and respecting file roles, *testBench* can reliably execute, evaluate, and iterate on exercises in a controlled, reproducible manner.

4.3.3 Submission Handler

The submission handler is the component in *testBench* responsible for capturing, storing, and organizing all student interactions with an exercise. It ensures that each student's work is persistently recorded and contextually linked to its original template, allowing for safe execution, testing,

and grading in isolated environments.

Its responsibilities span three main areas: saving edited code, maintaining internal file structures, and adapting imported exercises for submission compatibility.

GitHub and ZIP Imports

The submission handler plays a role in processing exercises at the moment of import. Instructors can import exercises through a public GitHub repository or a ZIP archive. In both cases, the handler normalizes the incoming folder structure and validates the presence of required files.

- When importing from GitHub, the latest commit from the default branch is downloaded and unpacked.
- When importing a ZIP archive, unnecessary outer folders are removed, and common platform-specific artifacts (e.g., `__MACOSX` or hidden files) are ignored.

After validation, the imported exercise is frozen as a template. All future submissions created from it are based on this version, ensuring consistency even if the source repository or ZIP file changes later.

Saving Student Code

Every time a student clicks the *Save* button while editing an exercise, the submission handler creates a new submission instance. This submission is stored in the database and includes all files the student has permission to modify. Each submission is tied to both the `user_id` and `exercise_id`, ensuring that each student maintains a unique and independent workspace per exercise.

The system avoids modifying the original exercise files. Instead, it derives a copy of the original template and overlays it with the student's edits. This isolation preserves the integrity of the base exercise and allows multiple students to work on the same content without interference.

File Structure Enforcement

To support reproducibility and compatibility with automated testing, *testBench* enforces a strict internal file structure. Upon submission, files are categorized and stored according to their intended role:

- **Editable Files** – These include source files that students are allowed to modify.
- **Locked Files** – These are system-critical files such as Makefile, exercise.json, devcontainer.json, and test cases. These files are protected and cannot be modified by the student to avoid compromising the execution pipeline.

This classification ensures that only permitted edits are saved and that the system maintains a consistent directory hierarchy required by the build tools.

The figure below represents the internal steps how the system processes submissions:

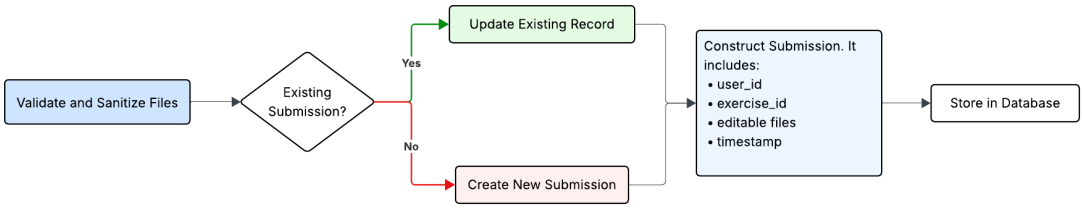

Figure 4.8: Submission component overview

4.3.4 DevContainer Orchestrator

The DevContainer Orchestrator is the subsystem within *testBench* responsible for managing isolated, containerized environments used to run, test, and grade student submissions. Using the Dev Container specification [19] to ensure that each exercise is executed within a consistent and reproducible runtime, regardless of the host machine or the underlying operating system.

This orchestrator encapsulates the full lifecycle of a containerized session, including environment setup, workspace mounting, execution of user commands (such as test or run), and eventual teardown.

Lifecycle Overview

Each time a student triggers execution (e.g., by pressing *Run Code*), the orchestrator initiates the following process:

1. **Build/Fetch Image:** Based on the `devcontainer.json` or `Dockerfile`, the system either builds a custom container or pulls a predefined image from a registry.
2. **Mount Workspace:** The student's submission is mounted into the container using a Docker volume or bind mount, typically at `/workspace`, as specified by the `workspaceFolder` field in the configuration.
3. **Environment Setup:** Optional hooks such as `postCreateCommand` are executed inside the container to install dependencies, configure tools, or prepare the testing environment.
4. **Command Execution:** The orchestrator runs specific commands (e.g., `make test`) inside the container, capturing output and test results. These results are parsed and stored for feedback and grading.
5. **Cleanup:** After execution, the container is stopped and removed to ensure stateless operation. Volumes may be retained temporarily for debugging or discarded immediately, depending on system policy.

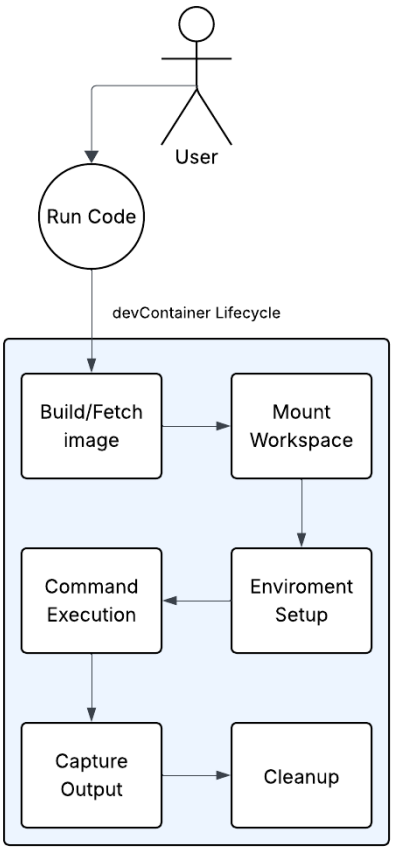

Figure 4.9: devContainer lifecycle overview

Mounting and File System Isolation

Each container is provisioned with its own isolated file system. Student code is injected via a volume or a mounted bind directory to `/workspace`, matching the expectations of most Dev Container configurations.

All locked or read-only files (e.g., `Makefile`, test cases) are injected from the frozen exercise template. Editable files are overlaid from the student submission. This ensures that the environment is consistent across attempts, but still reflects the student's individual code at runtime.

Image and Build Handling

The orchestrator supports two modes of container setup:

- **Prebuilt Image Mode:** If `devcontainer.json` specifies an `image`, it is pulled and reused directly. This allows fast startup and ensures predictable environments.
- **Dockerfile Mode:** If a `Dockerfile` is provided, the orchestrator builds a custom image before launching the container. This provides flexibility at the cost of increased build time.

Build caching is employed to speed up repeated invocations for the same exercise template, though containers are still created per submission to preserve isolation.

Execution Coordination

After the container is initialized and the workspace is mounted, control is handed over to the execution engine. That subsystem is responsible for invoking the build and test processes, capturing output, and initiating the grading phase.

Security and Isolation
All containers are executed with restricted permissions using Docker's default security profiles. By isolating builds inside containers and disallowing privileged operations, the orchestrator prevents unsafe or malicious code from affecting the host system.

Network access is disabled by default, and execution timeouts are enforced to prevent infinite loops or denial-of-service conditions.

Reproducibility and Cleanup
A key design principle of the orchestrator is statelessness. Once execution is complete and the results are collected, the container is destroyed. This ensures that each submission is evaluated in a clean environment and that stale data or state cannot affect results.

To further promote reproducibility, versioned exercise templates and immutable submission snapshots ensure that re-evaluation of past submissions yields consistent outcomes.

4.3.5 Execution Engine

The execution engine is responsible for coordinating and executing student submissions within the containerized environment prepared by the DevContainer Orchestrator. It serves as the runtime driver that triggers the testing process, manages execution flow, and collects relevant outputs for grading and feedback.

While the Orchestrator ensures the environment is properly initialized, the execution engine operates within that environment, focusing solely on the logic of running the student's code.

Submission Execution
Once the student's workspace has been mounted into the container, the execution engine initiates the predefined execution procedure. This typically involves running a test command, as specified in the exercise's configuration. The engine ensures that all editable files are accessible, and any required preparation steps — such as permission adjustments or dependency checks — are completed before execution begins.

Output Handling
As the student's code runs, all generated output — including success messages, compilation errors, and test results — is collected and relayed back to the platform in real time. This feedback is used both for user display and for subsequent grading. To preserve execution logs, the engine records all outputs for future review and troubleshooting, especially in the case of failed runs.

Error Management
The execution engine includes safeguards to detect and handle common failure scenarios such as syntax errors, missing files, or failed builds. These errors are caught and reported cleanly to the user interface, enabling students to iterate on their code with immediate insight into what went wrong.

Resource Constraints
In multi-user deployments, resource control is essential for maintaining system stability and fairness. Although the execution engine itself does not enforce resource limits, it operates within the constraints defined by the underlying container runtime or orchestrator. These constraints — such as CPU, memory, and execution time limits — are can be configured at the infrastructure level to prevent misuse and ensure predictable performance across concurrent executions.

Integration with Grading Pipeline
Once execution is complete, the results — including test reports and output logs — are passed to the test runner and grading component. The execution engine's role concludes at this stage, enabling a clean separation between running student code and evaluating its correctness.

The figure below summarizes the core responsibilities of the execution engine:

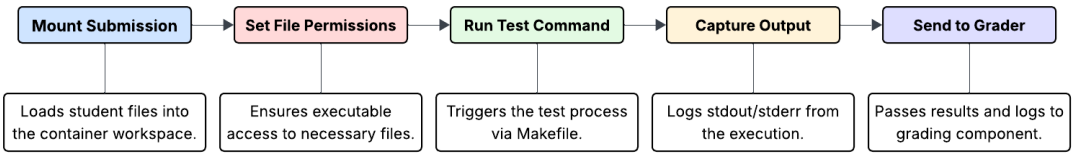

Figure 4.10: Execution component overview

4.3.6 Test Runner & Grader

Following the execution of a submission, the responsibility shifts to the test runner and grading component. Its core function is to interpret test results produced during the execution phase, calculate the corresponding grade based on predefined rules, and persist the outcome in a structured format.

This subsystem forms the final step of the submission lifecycle and directly impacts both student feedback and performance tracking.

Test Report Parsing

Each exercise is expected to produce machine-readable test reports, typically in the form of XML files following the JUnit schema [23]. These files are generated by the testing framework inside the container and stored in a designated results directory.

The test runner parses these XML documents to extract:

- The names of test cases executed.
- Their pass/fail status.
- Any failure messages or stack traces (for diagnostics).

This information is structured into internal data models for grading and feedback purposes.

Grading Logic

Each test case defined in the exercise's `exercise.json` is associated with a specific name and point value. The grader compares the parsed results with this schema to determine:

- Which test cases passed successfully.
- How many points should be awarded.
- Whether partial credit is applicable.

The final score is computed as the sum of all passed test cases, with optional scaling or weighting depending on the course configuration. This enables fine-grained control over how correctness is evaluated and how feedback is delivered to the student.

Feedback Generation

Once grading is complete, a detailed report is generated for each submission. This includes:

- A pass/fail indicator for each test.
- The total score earned and maximum possible.
- Any relevant diagnostic output (such as error logs or stack traces).

This report is stored alongside the submission and made accessible to both students and instructors for review. It enables transparent evaluation and supports iterative learning.

Persistence & Analytics

All grading outcomes are stored in the platform's database, linked to the user and exercise. This allows for historical tracking of progress, automated analytics, and the generation of performance summaries at both individual and course levels.

Instructors can use this data to identify common mistakes, monitor engagement, and assess learning outcomes across cohorts.

The diagram in Figure 4.11 shows the internal flow of the grading component, from test execution output to result storage.

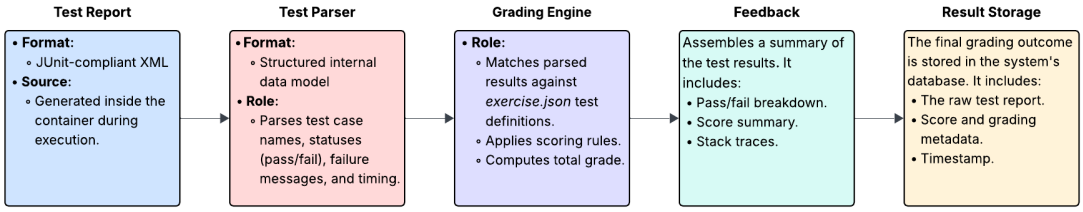

Figure 4.11: Grading component overview

4.3.7 WebSocket Logger

One of the core usability features within the *testBench* code editor is the ability to provide real-time feedback to students as their code is executed. To support this, the backend includes a WebSocket-based logging system that streams output from the execution container directly to the frontend interface [24]. This mechanism not only improves transparency but also supports iterative debugging, especially in cases where students encounter compilation errors or runtime failures.

Real-Time Feedback

Unlike traditional request/response models where the user must wait for a job to finish before seeing results, the WebSocket Logger enables a continuous stream of information during the execution process. As soon as a student triggers execution (e.g., by pressing *Run Code*), a WebSocket connection is established between the frontend and backend. The backend then begins sending log events to the connected client, updating the terminal view line by line as output is generated within the container.

This approach closely mirrors the developer experience in integrated development environments (IDEs), where feedback is immediate and interactive. It also helps students better understand their program's behavior by providing access to standard output and standard error streams in real time.

Logging Architecture

Internally, the logger is structured as a multi-producer, single-consumer pipeline. Execution routines running inside the container send log messages into a queue, which is asynchronously consumed by a background task responsible for dispatching the output to the WebSocket channel.

Each log message is structured as a JSON object containing metadata such as:

- **timestamp**: When the log was generated.
- **level**: Log severity (e.g., info, error).
- **origin**: Which subsystem produced the log (e.g., orchestrator, execution engine).
- **message**: The textual content of the log line.

This format allows frontend clients to render log entries consistently, apply filters, and even store logs for later inspection.

Error Tracing and Diagnostics

In the event of a failure, such as a compilation error, infinite loop, or container crash, the WebSocket Logger ensures that relevant error messages are transmitted in full to the user. This includes stack traces, exit codes, and interpreted error descriptions when available.

By exposing these failure signals directly through the interface, students are empowered to resolve issues autonomously without instructor intervention. It also enables instructors to diagnose platform-level issues more easily, as logs are preserved alongside each submission for later analysis.

Disconnection and Resilience

To support long-running jobs and intermittent connectivity, the WebSocket channel is designed with resilience in mind. If a connection is lost (e.g., due to a page refresh), logs are still preserved on the backend. Upon reconnection, the client may request a replay of the latest buffered output. Additionally, once execution completes, the full log stream is persisted and attached to the submission record for later retrieval.

To better illustrate how log data is propagated through the system, Figure 4.12 provides an architectural overview of the WebSocket Logger.

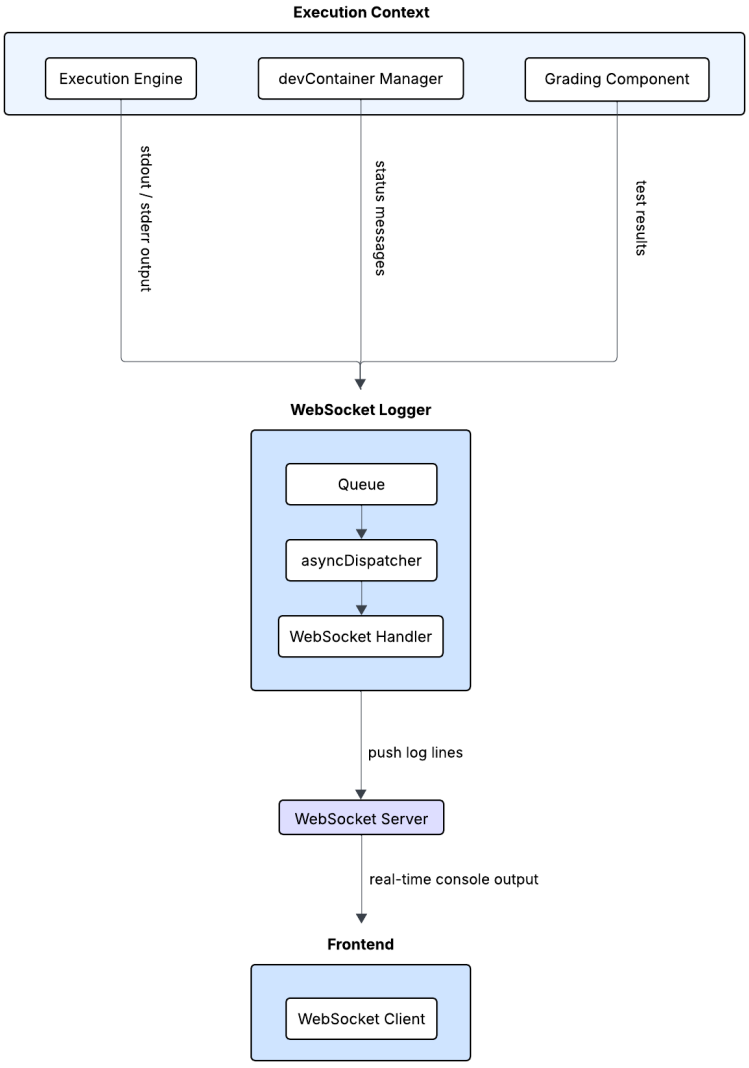

Figure 4.12: WebSocket logger overview

4.3.8 GUI Session Manager

Certain programming projects, especially those involving web applications, visual simulations, or other forms of interactive output, require more than standard terminal-based execution. To support this, *testBench* includes an experimental feature that allows students and instructors to launch the graphical component of a project inside an isolated container, accessible via a web browser.

This feature is distinct from providing a graphical development environment. Instead, it focuses on enabling users to run their projects as live applications and interact with them visually, using the browser as a window into the containerized interface.

Experimental Nature and Intended Use

This feature is currently under active development and is not yet considered production-ready. It is intended for:

- Projects that include web frontends or GUI-based interfaces.
- Courses where visual feedback is essential.
- Instructors who want to let students run their applications "as users," not just as code under test.

Examples include launching a Flask or React app, running simulations that render visual output, or viewing dashboards and GUI widgets.

Architecture and Access Flow

When a user clicks a *Launch GUI* button, the following sequence occurs:

1. The user's saved submission files are copied into a container workspace.
2. The platform reads the exercise's `devcontainer.json` and, if provided, a `Dockerfile`.
3. A container is started using `devcontainer up`, with appropriate port exposure configured.
4. The application runs inside the container, and the user is redirected to a URL such as:

   `https://<submission_id>.testbench.upc.edu`

Hostname-based routing is managed by a reverse proxy (Traefik), which dynamically connects the domain name to the running container instance.

Developer-Defined Behavior

Unlike the headless execution mode, where the system controls which command is run, the GUI session relies on the exercise defining its own runtime behavior. Instructors can configure containers to:

- Start a web server (e.g., `npm run start`, `python app.py`).
- Serve a graphical desktop application via a browser-compatible protocol.
- Launch apps that bind to predefined ports.

This behavior is specified using the `postCreateCommand` in `devcontainer.json`, or directly in the container's `Dockerfile`.

Container Lifecycle and Isolation

Each GUI container is:

- Spawned on demand for a specific submission.
- Isolated at the Docker level (one per student/exercise).
- Automatically assigned a hostname and tracked in-memory for session control.

Users can manually stop the session via a dedicated endpoint, or it will be automatically terminated after a configurable period of inactivity.

	Headless Mode	GUI Session
Output Type	Test Results	Graphical Interface
Execution Trigger	Run Code	Launch App
Purpose	Grading	Visual Interaction
Port Exposure	Not Required	Required (via proxy)
Entry Point	System-controlled	Exercise-defined

Table 4.1: Comparison between headless and GUI execution modes

Comparison to Headless Execution

Limitations and Roadmap

As of now, this feature has several limitations:

- No persistent storage inside GUI containers.
- No authentication layer between the user and the launched app.
- Static, manual port exposure.
- Basic error handling and resource control.

Future improvements under consideration include:

- Secure WebSocket and port forwarding.
- Automatic resource limits and session quotas.
- Expiring access URLs and signed tokens.

This component lays the foundation for supporting richer and more interactive student projects while maintaining the principles of isolation, automation, and reproducibility that define the *testBench* architecture.

4.4 Operational Workflow

After exploring the individual components that make up the architecture of *testBench*, we now turn our attention to how these elements come together in practice. This section walks through the complete lifecycle of interaction from both the teacher's and the student's perspective. By outlining typical workflows, we illustrate how the platform functions as a cohesive system to support automated, containerized, and structured programming exercises.

4.4.1 Teacher Workflow

This section presents the teacher's journey through *testBench*, following a typical sequence of actions taken to prepare and manage a course. Rather than focusing on internal implementation details, this walkthrough emphasizes the practical flow from the teacher's perspective—from initial setup to reviewing student performance.

1. **Logging into the Platform** — The teacher begins by logging into the platform using their credentials. Upon successful login, they access a dashboard summarizing their courses, exercises, and student activity.
2. **Creating a New Course** — A new course can be created by entering a name and optional description. This course acts as a container for exercises and students.
3. **Enrolling Students** — By uploading a CSV file with student email addresses, the teacher can automatically create student accounts and distribute login credentials via email.
4. **Importing Exercises** — Exercises are added by uploading a ZIP file or linking a public

GitHub repository. The platform processes the files, extracts configuration and metadata, and prepares the exercise for use.

5. **Assigning Exercises to Courses** — Imported exercises are assigned to one or more courses, determining which students can access them.
6. **Exercises Visibility** — Exercises remain visible until the teacher explicitly marks them as hidden. Only visible exercises can be viewed and attempted by students.
7. **Monitoring Student Progress** — Teachers can track submissions and grades in real time through the results dashboard, gaining insight into each student's performance.
8. **Previewing Graphical Output (Optional)** — For GUI-based or web-oriented assignments, teachers can launch a live preview of a student's project in the browser without setup.
9. **Iterating on Content** — Teachers can revise exercises, toggle their visibility, or reassign them as needed. This flexibility supports both iterative content development and responsive teaching.

This streamlined workflow enables instructors to manage course content, onboard students, and evaluate performance without needing to interact with the underlying infrastructure. The goal is to keep the teaching experience focused and efficient.

To better represent the typical journey a teacher follows while using *testBench*, the following diagram presents a high-level overview of the workflow.

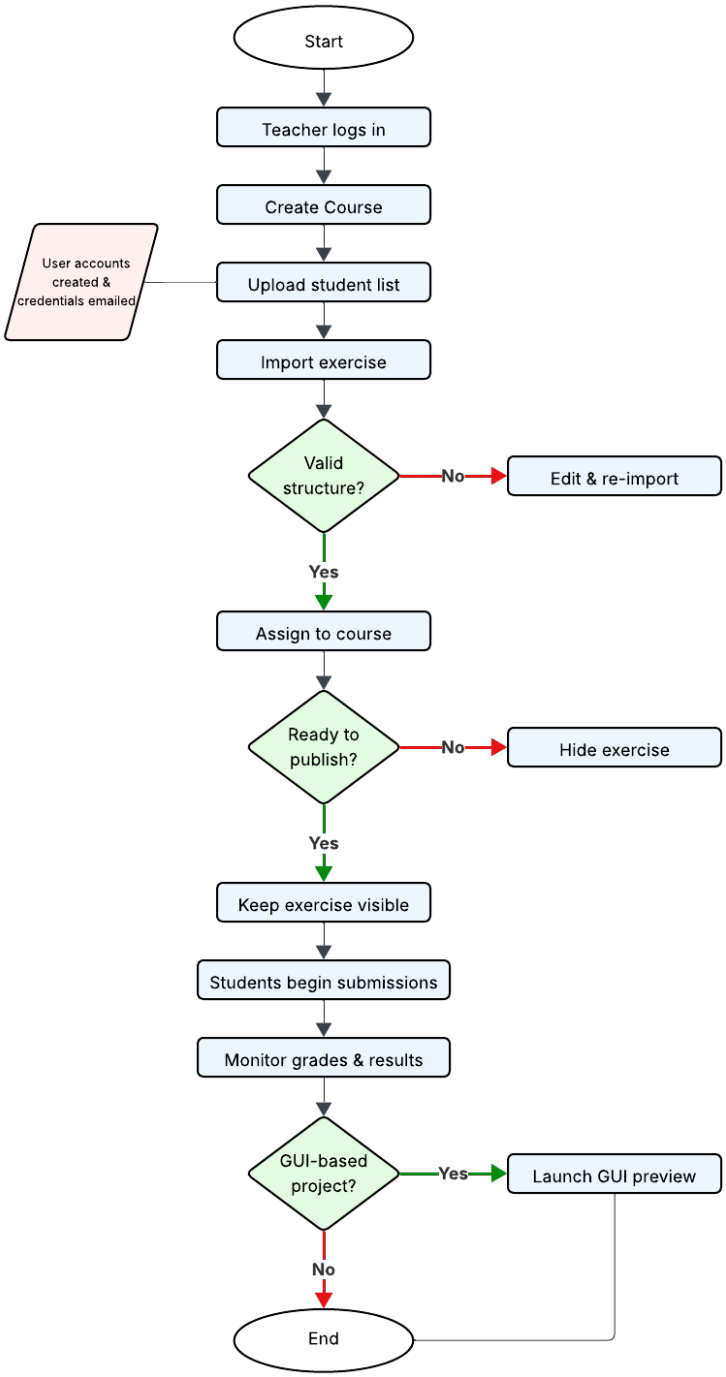

Figure 4.13: Flow Diagram: Teacher workflow

4.4.2 Student Workflow

From the student's perspective, interacting with *testBench* is centered on solving the programming exercises assigned within a course. The platform is designed to minimize friction and present a streamlined interface for coding, submitting, and reviewing results. The typical workflow follows the steps listed below:

1. **Logging In** — Students receive credentials via email and log in to access their personalized dashboard, which shows only the courses and exercises they've been assigned.

2. **Viewing Available Exercises** — Within each course, students can see a list of visible exercises. Each exercise entry includes a title, description, and link to open the coding interface.
3. **Reading the Statement** - When entering an exercise, the platform renders the Markdown Statement file, providing the student with instructions, goals and requirements for the task.
4. **Editing Code** — Students work on their solution in the built-in code editor. The interface replicates a typical development environment with a file tree, syntax highlighting, and autosave.
5. **Saving a Submission** — Once ready, students can save their progress. This action creates or updates a submission that is tied to the current user and exercise.
6. **Running Code** — Students can run their submission inside a containerized environment. This process compiles the code, executes tests, and returns real-time feedback via a terminal view.
7. **Reviewing Results** — After execution, the student receives detailed feedback from the test, including pass / fail status, points awarded, and overall grade.
8. **Iterating** — Students may update and re-submit their solution as many times as needed before the exercise is locked or the deadline passes.

The following diagram visually summarizes this workflow, capturing each major interaction and illustrating the iterative nature of student submissions.

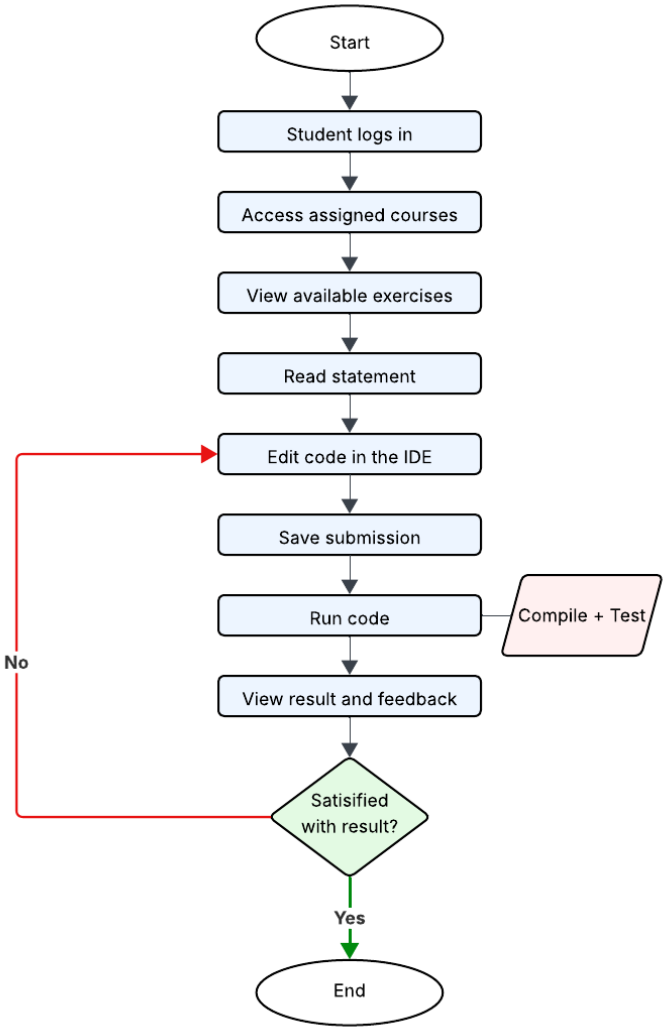

Figure 4.14: Flow Diagram: Student workflow

4.5 Deployment

This section outlines the deployment architecture of *testBench*, distinguishing between the development and production environments. Each environment is configured to prioritize reproducibility, isolation, and ease of use, while accommodating the different needs of local testing and live operation.

4.5.1 Development

To support local development, *testBench* relies on a modular environment composed of a manually launched frontend and backend, combined with a containerized database service. This approach was chosen to provide maximum flexibility during development, while ensuring consistency in service behavior through containerization and automation scripts.

The application's source code is maintained in a private Git repository, allowing for version control and rollback capabilities when needed. The repository contains the full implementation of the backend, frontend, configuration files, test suites, and container definitions.

4.5.1.1 Backend and Frontend Execution.

The backend, implemented in FastAPI, is executed directly through a Python runtime. It requires a working virtual environment and a set of dependencies listed in `requirements.txt`. The frontend, built with Flutter, is similarly run locally through the Flutter CLI, targeting web or desktop during development. Both services are launched manually to allow targeted debugging and partial testing without the overhead of a full container stack.

4.5.1.2 Database

The MongoDB database is the only component containerized during development. It is orchestrated via a dedicated Docker Compose configuration file, `docker-compose-local.yml`, which provisions a development database container with the following properties:

- Uses the official MongoDB 5.0 image.
- Exposes port `27017` to the host, allowing local clients to connect without special configuration.
- Initializes the container with a root username and password, defined via environment variables for security and reusability.
- Attaches a persistent named volume (`mongo_data`) to store the database files across restarts.

```
version: "3.9"
services:
  mongodb:
    image: mongo:5.0
    container_name: exam-manager-db
    ports:
      - "27017:27017"
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_INITDB_ROOT_USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_INITDB_ROOT_PASSWORD}
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Figure 4.15: Local Docker Compose configuration for MongoDB

This setup ensures that developers do not need to install MongoDB locally while maintaining a predictable and isolated database environment consistent with production.

4.5.1.3 Automated Environment Setup.

To reduce setup friction and ensure the host environment is correctly prepared, a script named `setup-dev.sh` is included in the repository. This script performs several preparatory tasks:

1. Installs Python dependencies via `pip`, ensuring the backend can be run immediately.
2. Installs the DevContainer CLI globally using `npm`, which is essential for managing containerized student code execution environments.
3. Verifies the presence of Docker on the host system, and exits with an error message if it is missing.
4. Creates the `/var/submissions` directory with full permissions. This path is used by the system to mount and execute student code in isolated containers, and must be accessible to both the main application and the Docker daemon.

```
#!/bin/bash

echo "Setting up the environment..."

# Install Python dependencies
echo "Installing Python dependencies..."
pip install -r requirements.txt

# Install Devcontainer CLI
echo "Installing Devcontainer CLI..."
sudo npm install -g @devcontainers/cli

# Verify Devcontainer CLI installation
if ! command -v devcontainer &> /dev/null; then
    echo "Error: Devcontainer CLI installation failed!"
    exit 1
fi

# Ensure Docker is installed
if ! command -v docker &> /dev/null; then
    echo "Error: Docker is not installed. Please install Docker and
       restart."
    exit 1
fi

# Create the /var/submissions directory with correct permissions
echo "Creating /var/submissions directory..."
sudo mkdir -p /var/submissions
sudo chmod 777 /var/submissions
sudo chown $(whoami):$(whoami) /var/submissions

echo "Setup completed successfully!"
```

Figure 4.16: `setup-dev.sh`: Local environment initialization script

This script standardizes the local environment and reduces the number of manual configuration steps developers need to perform. Since containerized execution plays a central role in the platform's grading logic, the correct creation of the `/var/submissions` mount point is critical. Without it, container orchestration and test execution would fail.

Overall, the development environment is intentionally designed to be simple, predictable, and scriptable. By containerizing only the components that benefit most from isolation (e.g., the database), and leaving the backend and frontend manually controlled, developers retain fine-grained control over each part of the system while minimizing environmental drift between contributors.

4.5.2 Production Deployment

In contrast to the modular and partially manual setup used during development, the production deployment of *testBench* adopts a fully containerized architecture. This design ensures consistent operation, scalability, and ease of maintenance when running the application in a live environment.

The platform is deployed on a dedicated server within the UPC institutional network, using Docker Compose [25] to orchestrate all services. The setup comprises four main components: the frontend, backend, MongoDB database, and a reverse proxy (Traefik) [26] responsible for managing network traffic and TLS termination.

Each service is defined as a separate container in the deployment configuration, with the backend

and frontend having dedicated Dockerfiles and versioned images published to Docker Hub [25].

4.5.2.1 Backend

The backend remains implemented in FastAPI, but in production, it is encapsulated in a custom Docker image built from a dedicated Dockerfile. This container is optimized for production use and includes only the necessary runtime and dependencies.

Unlike the local setup where the backend is run manually, the production instance is started automatically through Docker Compose. It is configured to read from an environment file (`.env`) and expose its API over port 8000, which is routed by Traefik [26].

Communication with the MongoDB container and the DevContainer runtime (via Docker-in-Docker) is maintained, as is the dependency on the `/var/submissions` directory, which is mounted as a shared volume.

```
backend:
  image: antonibote/testbench-backend:0.0.35
  container_name: testbench-backend
  env_file:
    - backend/.env
  environment:
    - ENV=production
  expose:
    - "8000"
  depends_on:
    - mongodb
  volumes:
    - /var/submissions:/var/submissions
    - /var/run/docker.sock:/var/run/docker.sock
  networks:
    - traefik
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.backend.rule=Host(`testbench.upc.edu`)
       && PathPrefix(`/api`)"
    - "traefik.http.routers.backend.entrypoints=websecure"
    - "traefik.http.routers.backend.tls=true"
    - "traefik.http.routers.backend.priority=2"
    - "traefik.http.services.backend.loadbalancer.server.port=8000"
```

Figure 4.17: Docker Compose service definition for backend

4.5.2.2 Frontend

The frontend is built using Flutter and compiled into static web assets using the build web command. In production, these files are served through a lightweight Nginx container [27], packaged into a custom Docker image. The Dockerfile for this image simply copies the build output into Nginx's default serving directory, making it efficient and easy to deploy.

Unlike in development, where the app might be served locally through Flutter's development server, the production frontend is started automatically via Docker Compose. The container runs on port 80 internally and doesn't expose any ports to the host directly; instead, it connects to the Traefik network, which handles all external traffic.

Traefik is configured to route requests sent to the root domain (testbench.upc.edu) to the frontend container over HTTPS. This is done through a set of container labels that specify the routing rule, TLS usage, and the internal port to forward to. With this setup, users accessing the domain receive the compiled Flutter web app securely, served by Nginx behind Traefik.

```
frontend:
  image: antonibote/testbench-frontend:0.0.9
  container_name: testbench-frontend
  expose:
    - "80"
  networks:
    - traefik
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.frontend.rule=Host(`testbench.upc.edu`)
      "
    - "traefik.http.routers.frontend.entrypoints=websecure"
    - "traefik.http.routers.frontend.tls=true"
    - "traefik.http.routers.frontend.priority=1"
    - "traefik.http.services.frontend.loadbalancer.server.port=80"
```

Figure 4.18: Docker Compose service definition for frontend

4.5.2.3 MongoDB Database

The MongoDB service remains mostly unchanged compared to development, but it now uses hardcoded credentials within the Docker Compose file for simplicity (secured by limited external access and internal networking). The database container is linked to a persistent volume named `mongo_data`, ensuring data survives restarts and upgrades.

A health check is defined to ensure the database service is responsive before the backend attempts to connect.

```
mongodb:
  image: mongo:5.0
  container_name: testbench-db
  environment:
    MONGO_INITDB_ROOT_USERNAME: antonibote
    MONGO_INITDB_ROOT_PASSWORD: antonibote
  volumes:
    - mongo_data:/data/db
  networks:
    - traefik
  healthcheck:
    test: ["CMD", "mongo", "--eval", "db.adminCommand('ping')"]
    interval: 10s
    timeout: 5s
    retries: 5
  labels:
    - "traefik.enable=false"
```

Figure 4.19: Docker Compose service definition for MongoDB

4.5.2.4 Traefik Reverse Proxy

Unlike in development, the production deployment introduces Traefik as a dynamic reverse proxy. Traefik handles all incoming HTTP and HTTPS traffic and routes it to the appropriate container based on domain and path rules.

It also manages TLS encryption using a self-signed certificate resolver and exposes a secure dashboard interface for monitoring purposes. Traefik is configured to read Docker metadata and automatically discover services through their container labels, significantly reducing manual routing configuration.

```
traefik:
  image: traefik:v3.0
  container_name: traefik
  command:
    - "--api.dashboard=true"
    - "--providers.docker=true"
    - "--entrypoints.web.address=:80"
    - "--entrypoints.websecure.address=:443"
    - "--certificatesresolvers.selfsigned.acme.tlschallenge=true"
    - "--certificatesresolvers.selfsigned.acme.email=
        admin@testbench.upc.edu"
    - "--certificatesresolvers.selfsigned.acme.storage=/letsencrypt
        /acme.json"
    - "--entrypoints.websecure.http.tls.certresolver=selfsigned"
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - "/var/run/docker.sock:/var/run/docker.sock:ro"
    - "./letsencrypt:/letsencrypt"
  networks:
    - traefik
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.api.rule=Host(`traefik.testbench.upc.
        edu`)"
    - "traefik.http.routers.api.service=api@internal"
    - "traefik.http.routers.api.entrypoints=websecure"
    - "traefik.http.routers.api.tls=true"
```

Figure 4.20: Docker Compose service definition for Traefik

4.5.2.5 Setup Script

Much like in development, the container is initialized using a helper script named `setup-prod.sh`, which automates the installation of necessary runtime tools and environment preparation. This includes installing Python dependencies, Node.js, and the DevContainer CLI, as well as creating the shared submissions directory.

```
#!/bin/bash

# Install Python dependencies
pip install -r requirements.txt

# Install Node.js (v20) and npm
apt-get update
apt-get install -y curl gnupg
curl -fsSL https://deb.nodesource.com/setup_20.x | bash -
apt-get install -y nodejs

# Install Devcontainer CLI
npm install -g @devcontainers/cli

# Create submissions directory
mkdir -p /var/submissions
chmod 777 /var/submissions
```

Figure 4.21: Production environment setup steps

4.5.2.6 Architecture Overview

Figure 4.22 presents a high-level overview of the containerized deployment architecture. Traefik acts as the entry point to the system, forwarding requests to either the frontend or backend based on routing rules. The backend communicates with the database.

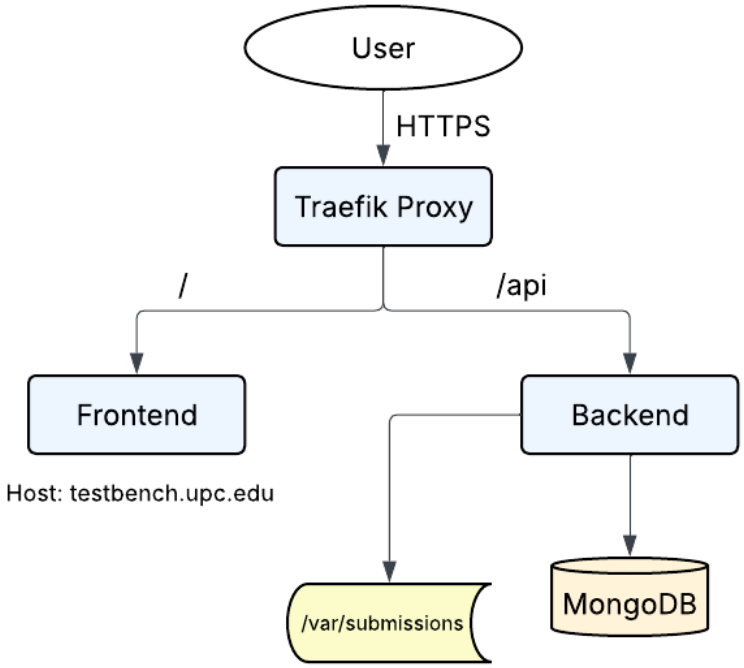

Figure 4.22: Production deployment with Traefik routing

As previously stated, the system is fully containerized and managed through Docker Compose to ensure reliability, scalability, and security. Traefik acts as a reverse proxy, terminating HTTPS connections and routing incoming requests to the appropriate service based on domain and path rules. The Flutter Web frontend is served over HTTPS, while API requests are directed to the FastAPI backend, which connects to a persistent MongoDB instance. The backend also mounts a host volume to store user submissions. This setup ensures consistent behavior, secure access, and clear separation of responsibilities across services in a live environment.

5 Results

The aim of this chapter is to offer a walkthrough of the application's interface to demonstrate its key features and operational flow.

5.1 Splash Screen & Login Screen

When navigating to `testbench.upc.edu` in a browser, users are first presented with a splash screen, which then transitions to the login screen. Figure 5.1 shows the splash screen.

Figure 5.1: Splash screen of the application.

The login screen allows users — both teachers and students — to enter their credentials (email and password) to access the platform. Because student accounts are generated by teachers, a registration screen is not necessary. While a registration interface does exist, it is intentionally hidden from the user interface.

The design of these screens — like the rest of the application's interface — follows a minimalist and clean aesthetic. The goal is to ensure that the platform remains accessible, intuitive, and free of distractions. By reducing visual clutter, the focus remains on the core functionality and purpose of the app, rather than on decorative elements. Figure 5.2 shows the login screen.

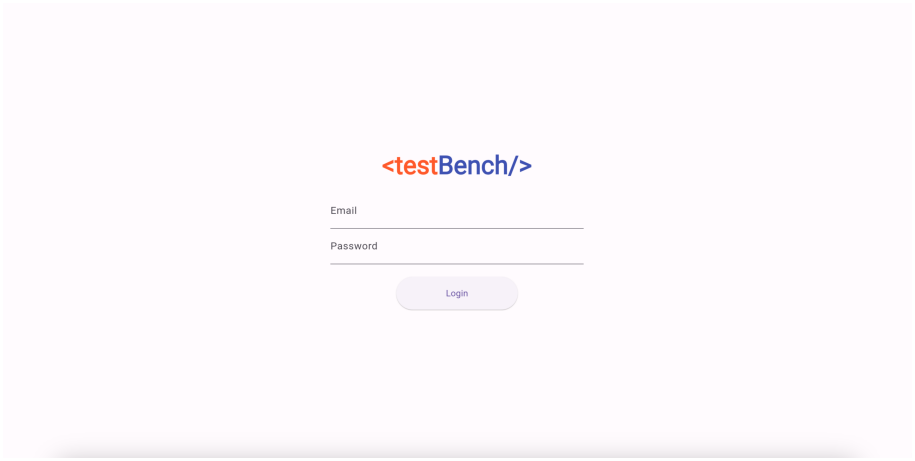

Figure 5.2: Login screen with fields for email and password.

5.2 Home Screen

After a successful login, users are directed to the home screen, which provides a personalized overview of their activity on the platform.

At the top of the screen, a welcome message greets the logged-in user by name, alongside a link to access profile settings. Below, users can navigate through three main interface areas: My Profile, My Results, and My Submissions. A fixed navigation bar at the bottom of the screen provides quick access to primary application sections, using intuitive icons for navigation.

Interactive elements, such as submission cards and buttons, respond to hover events by slightly changing color — providing a visual highlight to guide user interaction.

Figure 5.3: Home Screen

To better understand the structure and functionality of the interface, we now examine each key component that constitutes the home screen in detail.

5.2.1 My Profile Screen

Accessible from the top-right corner of the home screen, the “My Profile” link leads to a personal profile editor (Figure 5.4). Here, users can view and update basic information such as:

- Name and surname,
- A brief “About Me” section,
- Their password.

Each field is presented using clean, bordered input boxes for easy readability. A "Save Changes" button allows the user to confirm and persist any updates. A back arrow in the top-left corner enables returning to the home screen.

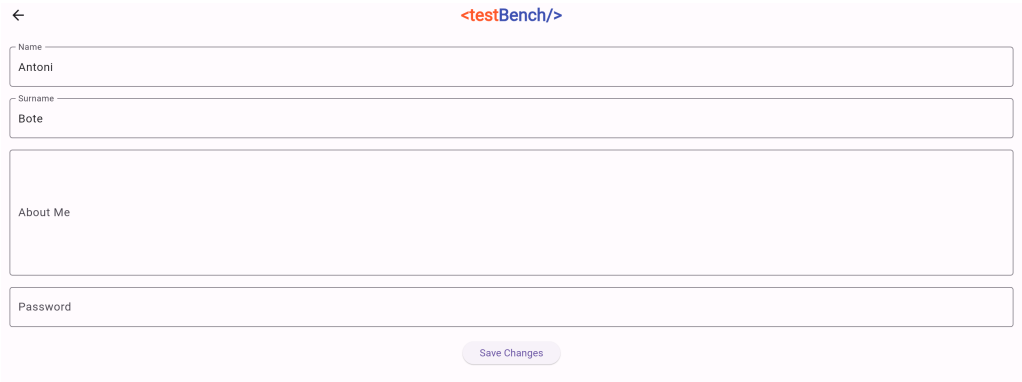

Figure 5.4: My Profile

5.2.2 My Results Screen

By clicking the “My Results” button from the home screen, users are redirected to a summary view of their performance (Figure 5.5). A dropdown menu at the top allows filtering results by course. Below that, a table lists all associated exercises along with their corresponding grades or status.

A “Download CSV” button on the right provides an option to export the data, enabling offline access or external analysis of performance.

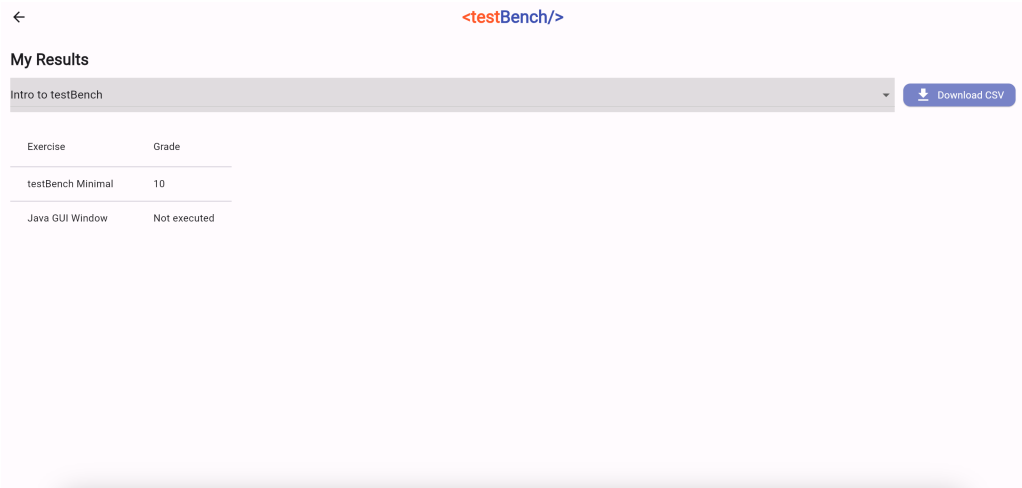

Figure 5.5: My Results

5.2.3 My Submissions

The “My Submissions” section on the home screen lists all exercises the user has submitted, each displayed as a card element. These cards summarize essential information:

- The title of the exercise
- The submission date
- The assigned grade or evaluation status

Each card includes an icon on the right-hand side which, when clicked, opens a modal dialog displaying a breakdown of the test results associated with that particular submission (Figure 5.6). This popup presents detailed feedback on each individual test case, including:

- The test name
- Points earned
- A short description of the test outcome

- The final status (e.g., *Passed*), visually emphasized using color coding — green for success and red for failure

This detailed view provides transparent feedback, helping students understand their score and performance at the level of individual test cases. Users can scroll through the modal if multiple test cases are listed and may close it with a dedicated action at the bottom.

Additionally, hovering over any submission card changes its background color slightly, offering visual feedback and highlighting interactivity.

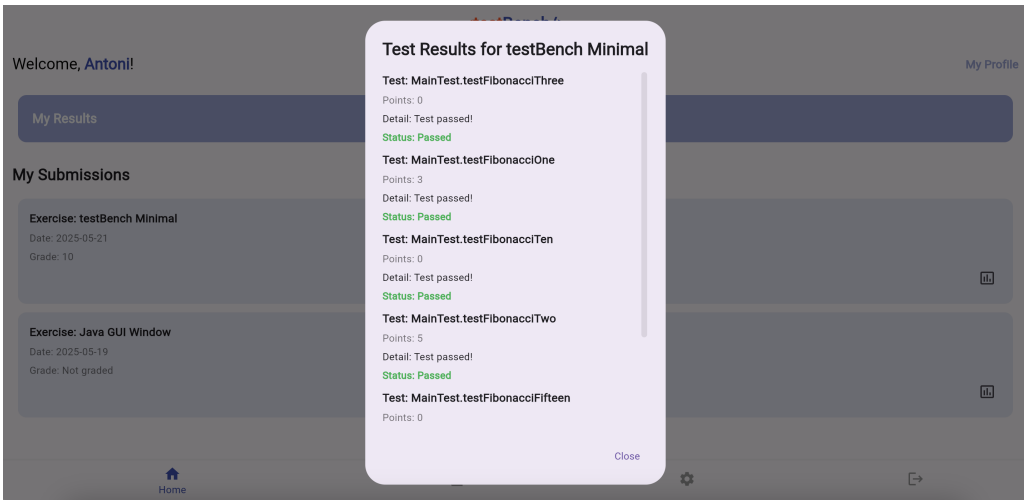

Figure 5.6: Popup modal showing detailed results for an individual submission.

5.3 Exercises Screen

The “Exercises” section of the application serves as the central hub where users can browse and access both exercises and courses. At the top of this page is a segmented control that allows users to toggle between two views: the Exercises List and the Courses List, as shown in Figure 5.8.

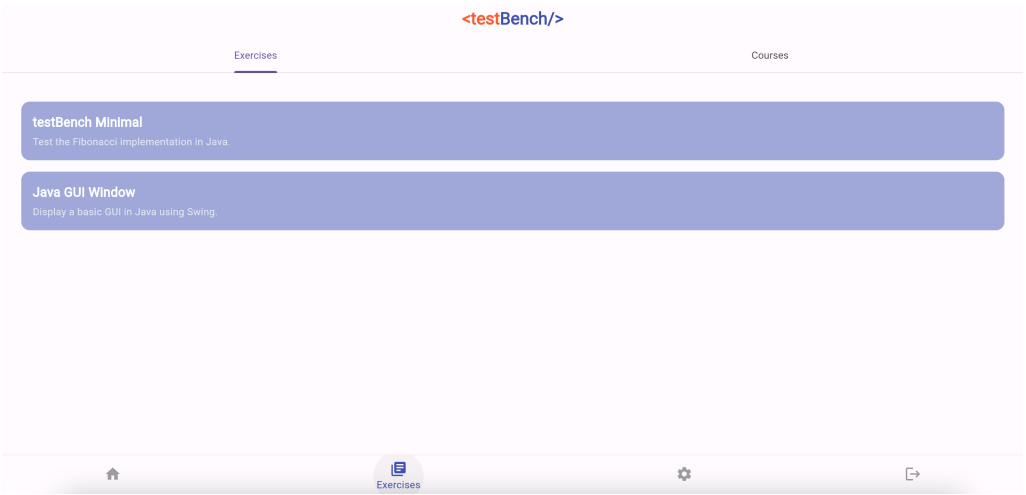

Figure 5.7: Exercises Screen

5.3.1 Exercises Tab

In the exercises tab, individual exercises are displayed as interactive cards. Each card represents an exercise. These cards feature:

- The title of the exercise
- A brief description

Hovering over any card causes a subtle change in background color, consistent with the interaction model established in earlier parts of the interface. When clicking on an individual exercise, the user is taken to its detail, which will be inspected later.

5.3.2 Courses Tab

Similarly to the exercises tab, individual courses are displayed as interactive cards. Each card represents a course. These cards feature:

- The title of the course
- A brief description

Figure 5.8 shows this.

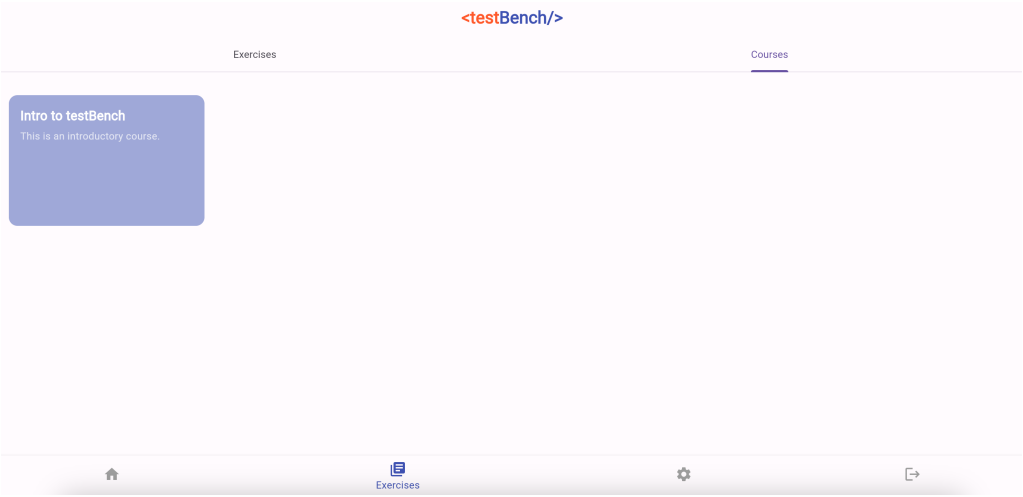

Figure 5.8: Courses Screen

5.3.2.1 Course Detail Screen

When a user selects a course from the Courses tab, the application transitions to a detailed view of that specific course. This view presents a list of all exercises contained within the course, each represented as a clickable card. An example is shown in Figure 5.9.

Figure 5.9: Course Detail

The course detail screen also includes a persistent back button in the top left corner, allowing users to return to the Courses list. This helps maintain navigational clarity.

As with other parts of the platform, the interface remains clean and functional, prioritizing accessibility and content clarity.

5.4 Teacher Dashboard Screen

The Teacher Dashboard is a role-specific screen that is accessible only to users with the "teacher" role. It provides a streamlined interface with various cards, each leading to a screen that allows teachers to manage specific aspects of the learning environment.

Figure 5.10: Teacher Dashboard

Each subsection below will provide a detailed explanation of the functionalities and layout of each screen within the Teacher Dashboard.

5.4.1 Students

The "Students" card provides an intuitive interface for instructors to manage all students enrolled in their courses. Teachers can view a list of students, delete individual student entries, and import new students via email addresses. The student information appears in a clear and organized list,

where each entry displays the student's email address. The interface also includes a button for importing students, represented by an upload icon. The delete button allows teachers to remove a student from the app, along with all their related submissions and data, ensuring complete removal from the system.

As explained in previous chapters, student data can be imported by uploading a CSV file containing student email addresses. After the import, credentials are automatically sent to each student's email, granting them access to the platform.

Figure 5.11: Students

5.4.2 Courses

The courses dashboard allows teachers to manage all courses they are responsible for. Here, they have the ability to add and delete courses, as well as to access each course by clicking on its card to view or modify course content.

Figure 5.12: Course Dashboard

5.4.2.1 Course Dashboard Detail

The course dashboard provides an overview of the course, including available exercises and a list of assigned students. Teachers can access the detailed view by selecting the course card. The main features in this section include:

- **Viewing Exercises:** The course dashboard displays a list of exercises associated with

the course, including titles and descriptions.

- **Assigning Exercises:** Teachers can add existing exercises to the course by selecting from a list of available exercises.
- **Unassigning Exercises:** Teachers can unassign exercises from the course by clicking the trash can icon next to the exercise. This allows the removal of an exercise from the course without deleting the exercise from the system.
- **Assigning and Unassigning Students:** Teachers can assign students to specific exercises. The "Assign Students" modal displays a list of students with checkboxes, allowing teachers to easily assign or unassign students to the course.

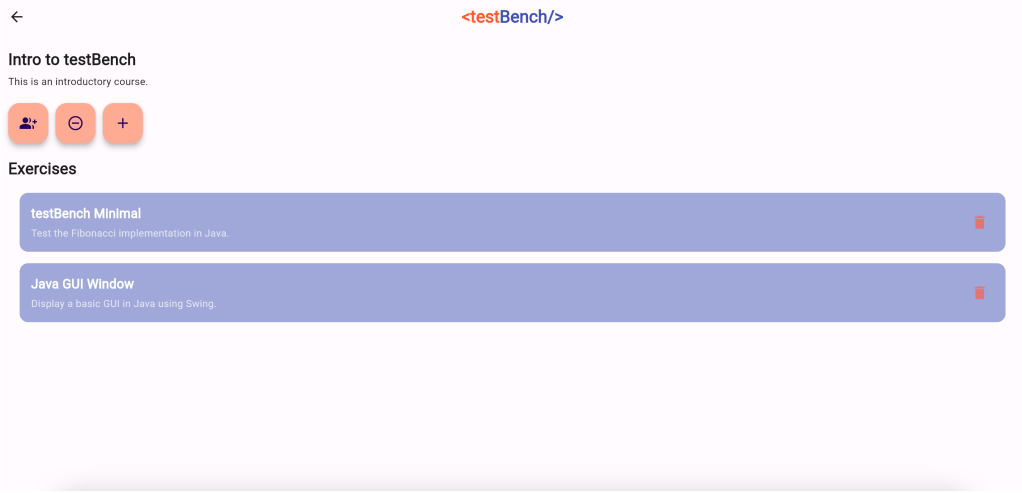

Figure 5.13: Course Detail Dashboard

Assign Exercises

In this section, teachers can assign exercises to a course from a preexisting list of exercises. By clicking the + icon, a list of exercises appears, from which the teacher can select the exercises they want to add.

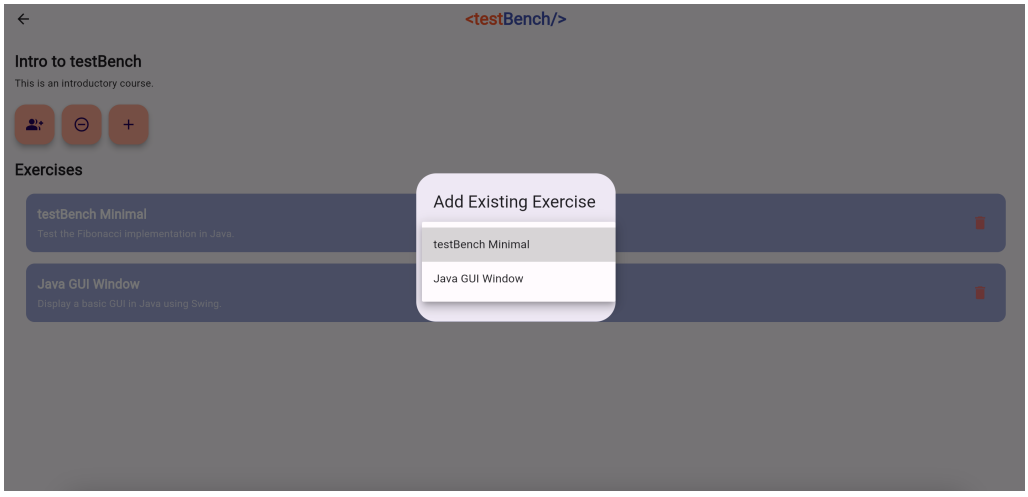

Figure 5.14: Assigning Exercises to a Course

Assign Students

Teachers can assign students to the course through the “Assign Students” modal. This modal presents a list of students with checkboxes, allowing teachers to select students they want to assign to specific exercises. Once selected, students are assigned to the chosen exercises.

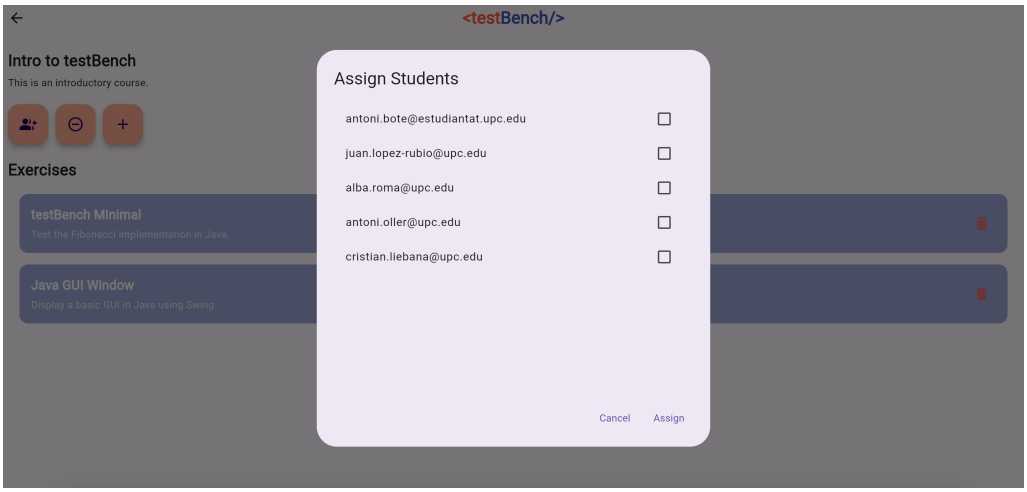

Figure 5.15: Assigning Students to a Course

Unassign Students

Teachers can also unassign students from the course. This modal presents a list of students with checkboxes, but in this case, teachers deselect the students they no longer wish to assign to a particular exercise. Once deselected, the student is removed from the exercise without being deleted from the system.

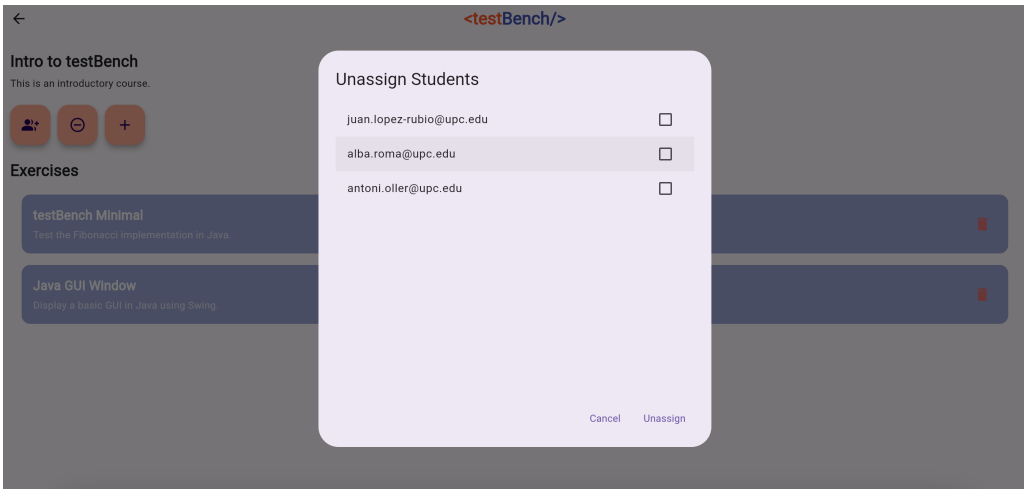

Figure 5.16: Unassigning Students from a Course

5.4.3 Exercises

The Exercises menu in the dashboard provides teachers with complete control over exercise management.

All available exercises are displayed in a scrollable list. Each exercise card includes:

- A title and description.
- A visibility icon to toggle the exercise's visibility in the other menus of the application.
- A trash icon to delete the exercise and its related submissions.

Clicking anywhere on a card navigates to the detailed view of that specific exercise.

At the top-left corner, two prominent buttons allow users to import new exercises:

- The first button (with a folder icon) enables uploading exercises from a local .zip file.

- The second button (with a GitHub icon) allows importing directly from a GitHub repository via its URL.

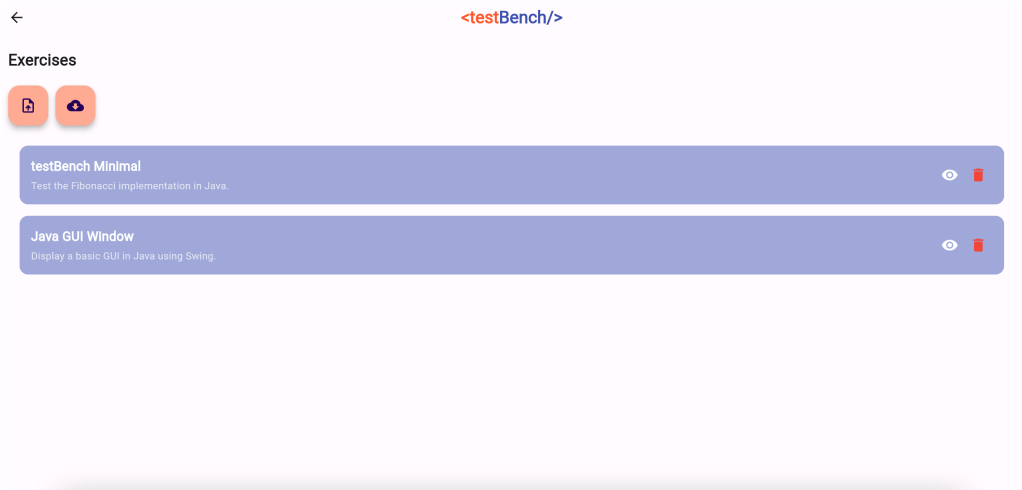

Figure 5.17: Exercises Dashboard Screen

5.4.4 Submissions

The Submissions screen enables teachers to view and manage student submissions for each exercise.

At the top-left, a dropdown selector allows the teacher to choose a specific exercise. Once selected, all corresponding submissions are displayed in a list format.

Each submission card includes:

- The name of the exercise.
- The email address of the student who submitted it.
- A trash icon that allows the teacher to delete the submission.

Figure 5.18: Submissions Dashboard Screen

5.4.5 Results

The Results menu allows teachers to easily view and analyze student performance for a specific course.

A dropdown menu at the top lets teachers select the course or group. Once selected, a table displays:

- The list of enrolled students.
- Their scores for each exercise.
- A calculated average score per exercise across all students.

Missing or unsubmitted exercises are represented with a dash (-).

Additionally, the complete results grid can be exported as a CSV file using the button on the top right.

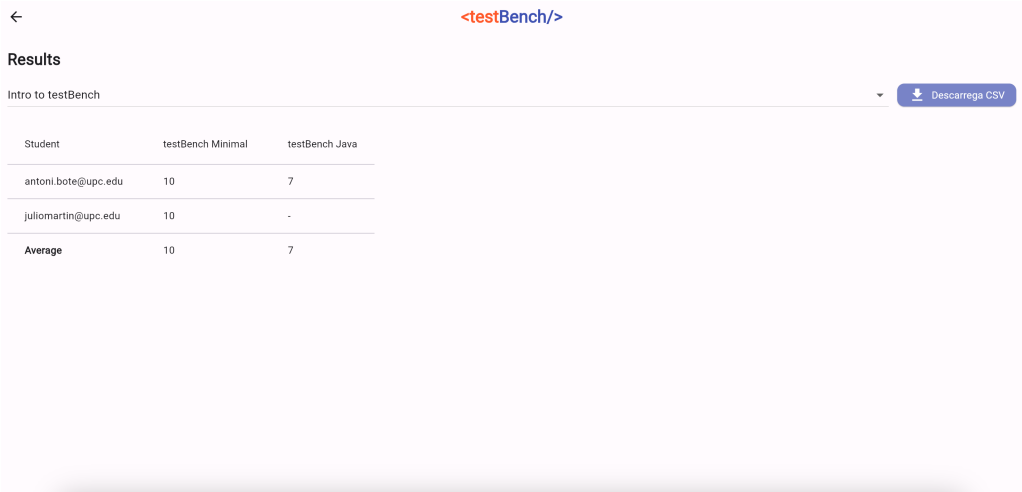

Figure 5.19: Results Dashboard Screen

5.5 Exercise

Each exercise in the application can be accessed through its corresponding card. An exercise consists of two main interfaces: the Exercise Detail and the Exercise Editor, both of which serve complementary purposes in the student workflow.

5.5.1 Exercise Detail

The Exercise Detail screen provides an overview of the exercise. It includes key metadata such as:

- The name of the exercise.
- A brief description.
- The programming language used.
- The associated course.

It also displays a scrollable exercise statement, which may include a project structure and additional instructional content written in Markdown. Students can review this information before starting the exercise using the Start Exercise button.

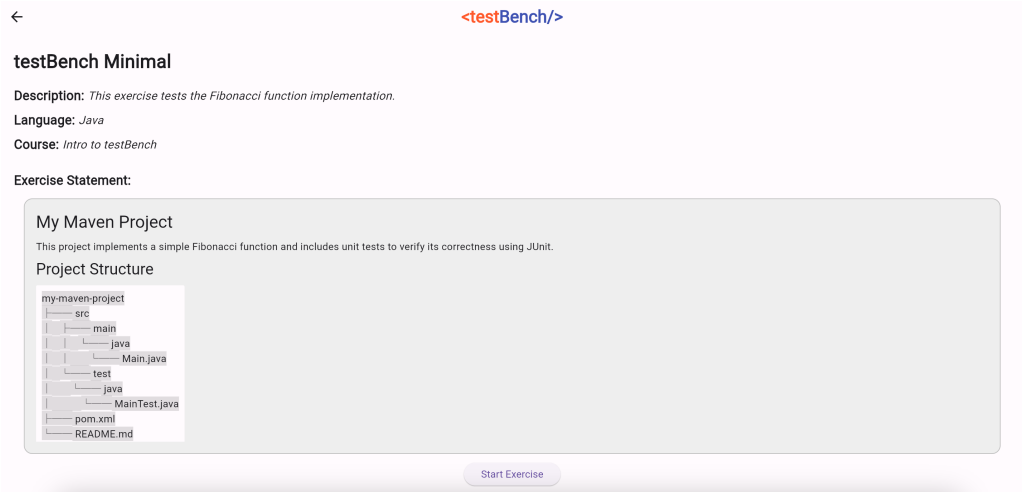

Figure 5.20: Exercise Detail Screen

5.5.2 Exercise Editor

Once the exercise is started, students are taken to the Exercise Editor. This environment includes:

- A file explorer on the left, listing all files and folders relevant to the project.
- A code editor in the center, where students can read and modify source code.
- A set of action buttons, including options to save changes, run the code, launch graphical interfaces (GUI), and stop execution.
- A built-in collapsible terminal at the bottom, which provides real-time feedback from build tools, test results, and standard output.

As noted in earlier sections, not all files displayed in the file explorer are editable. This is visually indicated by icons: a green pen denotes an editable file, while a red lock signifies a read-only file.

Figure 5.21: Exercise Editor Screen

5.6 Other components

Although not standalone screens, there are several important components that contribute to the overall user experience of the application.

5.6.1 Navigation Bar

The navigation bar is visible across all main screens and allows users to switch between different sections of the application. Its content is dynamic and adapts to the user's role, whether student or teacher. The available navigation items include:

1. Home screen
2. Exercises screen
3. Teacher dashboard (visible only to teachers)
4. Log out

5.6.2 Alerts

The application features alerts and snackbars that provide users with feedback on their actions. A green snackbar is displayed for successful actions, while a red one indicates an error or failed operation. Each snackbar includes a short message explaining the outcome. Below there is an example of each.

Figure 5.22: Succesful Alert Sample

Figure 5.23: Unsuccesful Alert Sample

This concludes the chapter, which provided a walkthrough of the application's main interface and features. By exploring each screen and its components, we've seen how the platform is designed to support both teachers and students in their day-to-day tasks. The goal has been to keep the experience clear, focused, and easy to use.

6 Evaluation

This chapter explores whether *testBench* is a suitable and reliable tool for real-world use in educational environments. It looks at the platform's technical performance, its potential effectiveness in supporting programming education, and discusses current limitations as well as areas for future improvement.

Since the platform has not yet been publicly released and no formal user testing has been conducted, this evaluation is based on the developer's own testing and experience, along with a forward-looking perspective on how the system is expected to perform in practical use.

6.1 Performance assessment

Evaluating the platform's performance requires looking at both the infrastructure it runs on and the internal mechanisms it uses to manage resource-intensive tasks. Because *testBench* relies heavily on containerized environments to execute code, system resources and the efficiency of container orchestration are key to its overall reliability.

6.1.1 Infrastructure Considerations

The application is designed to run on a Linux-based host with Docker and DevContainers support. As each code execution takes place inside a dedicated container, the underlying hardware plays a crucial role in determining how many submissions can be handled in parallel.

In its current setup, the backend launches containers using the DevContainers CLI. While this CLI does not expose resource limit flags such as `-cpus` and `-memory` directly, these constraints can be defined in a `docker-compose.yml` file within the `.devcontainer` folder. This file can specify per-service resource limits, and be referenced by the `devcontainer.json`, providing a configuration-based approach to resource control.

Additionally, although not currently implemented, the backend could be extended to enforce these limits directly in code. Since container execution is managed via subprocess calls to the CLI, switching to a more manual container launch approach would allow precise programmatic control over CPU, memory, and other runtime parameters. This provides flexibility to adapt the system as performance demands increase.

This becomes particularly important in educational scenarios where many students may run their code simultaneously. While typical development machines might handle up to 5–10 concurrent containers, production environments would benefit from dedicated servers or cloud-based instances with higher core counts and ample memory.

6.1.2 Scalability and Deployment Flexibility

One of the key strengths of *testBench* is its ease of deployment. The application can be containerized and migrated to more powerful infrastructure with minimal effort. This makes it highly adaptable to different contexts — from small classroom pilots to institution-wide deployments.

As usage grows, institutions could scale horizontally (adding more container hosts) or vertically (upgrading the hardware of a single host). Additionally, deploying to cloud platforms (e.g., AWS, GCP, Azure) would enable autoscaling based on demand.

6.1.3 Container Management and Execution Workflow

The execution pipeline for running student code is carefully structured:

- When a student submits their code, the backend writes the relevant files into a dedicated

execution directory.
- A `devcontainer.json` file defines the runtime environment, which is bootstrapped using the `devcontainer up` command.
- Code execution and testing are triggered via `make test`, and results are parsed from JUnit XML reports.
- WebSocket connections stream real-time logs and feedback back to the frontend.
- After execution, containers are stopped and removed, and their images are optionally cleaned up to save disk space.

This lifecycle ensures that containers are only kept alive as long as needed. The system also includes a background thread that monitors idle containers and automatically stops them after a predefined timeout. This prevents resource leakage and allows the platform to remain responsive over time.

6.1.4 Resource Efficiency and Optimization

The platform already employs several best practices to remain resource-efficient:

- Temporary files are scoped to isolated directories under `/var/submissions` and cleared as needed.
- Makefile-based builds ensure predictable and consistent execution workflows.
- Test parsing avoids unnecessary recomputation by only extracting relevant test case outcomes.
- Container cleanup with ‘docker rmi‘ helps reclaim disk space by removing unused images.

Together, these strategies allow the system to maintain stability and performance, even under moderate concurrent usage.

6.1.5 Summary

While *testBench* has not yet been tested under large-scale usage, its architecture is well-positioned to support growth. With its containerized execution model, and support for easy redeployment, the platform offers a solid technical foundation. Proper infrastructure provisioning and resource limiting will be key to scaling it for use in real environments.

6.2 Potential effectiveness in an educational context

Although the platform has not yet been tested in a live classroom setting, its design suggests strong potential for supporting programming education. Several aspects contribute to its educational value for both students and teachers.

6.2.1 Support for Student Learning

The application provides an interactive and immediate environment for writing, running, and testing code. Each exercise includes:

- A clear problem statement and project structure.
- A code editor with file navigation and terminal access.
- Instant feedback through test results and grading.

This setup encourages iterative learning, where students can experiment with code and quickly see the outcome. Detailed test feedback—delivered through color-coded result messages and point breakdowns—also helps students understand their mistakes and learn from them.

6.2.2 Assistance for Teachers

From the teacher’s perspective, the platform simplifies several key tasks:

- Creating and distributing coding exercises.

- Tracking student submissions and grades.
- Reviewing individual test outcomes when needed.

The teacher dashboard centralizes all of these actions and presents them in a clear, card-based layout. Automated grading further reduces the burden of manual review, allowing instructors to focus on more qualitative aspects of student progress.

6.2.3 Scalability and Accessibility

Because the platform is browser-based and requires no local setup from students, it lowers the barrier to entry, especially in courses with large or diverse student groups. Teachers can onboard students by importing a list of email addresses, and students can log in and begin coding almost immediately.

Moreover, its modular structure allows courses and exercises to be reused or adapted across different terms or subject areas, adding long-term value for institutions.

6.2.4 Conclusion

Taken together, these features position the platform as a promising tool for teaching and assessing programming skills. While future testing with real users will be necessary to validate its full effectiveness, the current design already reflects core pedagogical principles: clarity, feedback, and accessibility.

6.3 Limitations & Areas for Improvement

While the platform already offers a broad set of features for managing, submitting, and evaluating programming exercises, several limitations and areas for improvement have been identified during development.

Limitations

1. **Rigid exercise specification.** Each exercise must contain a precise set of files — such as `devcontainer.json`, `exercise.json`, and a `Makefile` — in order to function correctly. While this ensures consistency, it may also limit flexibility and increase the setup effort for teachers, especially those unfamiliar with containerized workflows.
2. **Image dependency in `devcontainer.json`.** The application relies on valid and appropriate base images defined in each exercise's `devcontainer.json`. If the specified image is outdated, missing, or incompatible with the student code or test setup, the execution will fail. This makes exercise portability somewhat fragile and can result in runtime errors that are hard for non-technical users to diagnose.
3. **Lack of formal user testing.** As of now, no live deployments or formal classroom trials have taken place. As a result, some design choices and workflows remain unvalidated by real-world users and may require revision based on future feedback.
4. **No offline or low-connectivity mode.** The platform currently assumes constant internet access and a functioning backend server. In contexts with unreliable connectivity, this could hinder its use, especially in under-resourced environments.
5. **Limited support for non-standard execution flows.** The current execution logic is tightly coupled to Makefile-based testing and assumes a linear run–test–grade pipeline. More complex or interactive workflows (e.g., debugging, graphical output, or language-specific REPLs) are not fully supported.

Areas for Improvement

1. **Modular learning with course units.** One potential enhancement is the introduction of "units" within courses. Units could act as learning checkpoints, with each unit unlocking upon successful completion of the previous one. This would support progressive, structured learning flows and increase engagement through gamified progression.
2. **Exercise duplication and editing.** While exercises can currently be imported, there is no support for duplicating an existing one or editing it directly through the interface. Adding this feature would simplify the process of iterating on assignments or reusing content across different courses or semesters. This functionality should be restricted to teachers and could include version history for safety.
3. **Incomplete GUI exercise support.** The platform includes an early-stage feature for GUI-based exercises, but its functionality remains limited. Some graphical interfaces fail to launch reliably, and there is no feedback mechanism for GUI status within the frontend. Improving this would open the door for more diverse types of programming assignments.
4. **More robust error reporting and debugging tools.** When a container fails to start or an execution encounters an error, the user is only shown minimal feedback. Adding clearer error messages, stack traces, or logs would help both teachers and students troubleshoot issues more effectively.

Even with its current limitations, the platform is in a solid place. It already provides a reliable and practical way to manage and assess coding exercises in an educational setting. With a few targeted improvements to address the gaps mentioned above, it has the potential to become an even more complete and dependable tool for both teachers and students.

7 Conclusions

This last chapter aims to look back and reflect on the journey so far. It revisits the original goals of the thesis to see how many were met, what challenges came up along the way, and how much progress was made from the initial idea to the final result.

It also looks ahead, suggesting possible next steps that could help improve or expand the app in the future. There is always room to grow, and a list of clear directions can give this project a second life beyond the scope of the thesis.

Finally, there is space for some personal reflections: what this experience has meant on a practical and personal level, and the lessons learned throughout the process.

7.1 Technical Conclusions

This thesis was developed to address a challenge identified by my thesis advisor during my internship in the Department of Network Engineering of the university: the need for a reliable and automated platform to manage programming projects. The aim was to design and implement *testBench*, a tool that streamlines project import, standardized environment setup, automated testing, and grading.

It is also important to mention that the goal was not just to achieve technical efficiency but also to make programming education more accessible, scalable, and engaging for both students and instructors.

With that in mind, a set of goals were outlined at the outset, most of which were successfully achieved:

- A backend system was implemented to manage multiple student submissions efficiently.
- A user-friendly frontend interface was developed to simplify tasks for instructors.
- DevContainers were successfully integrated, ensuring consistent, reproducible environments across all use cases.
- Automated testing, using frameworks like JUnit, was incorporated into the grading pipeline.
- The grading system compiles test results into scores in a scalable and objective manner.
- Project import features were built to integrate with existing course workflows.
- The system was designed with extensibility and maintainability in mind, following modular architecture principles.

Technically, the development of *testBench* involved the practical application of key technical skills across multiple areas, including container orchestration, backend development, and frontend integration. The system required designing reliable mechanisms for executing student code in isolated environments, coordinating asynchronous workflows, and ensuring smooth communication between components. These implementation challenges provided valuable experience in building real-world software systems with clear, outcome-driven objectives, particularly in the context of educational technology.

Beyond technical implementation, *testBench* reflects a broader educational motivation: to encourage more autonomous, hands-on learning among students, while minimizing friction caused by local setup issues or lack of timely feedback. The platform empowers students to experiment and learn through doing, and helps instructors focus on teaching rather than infrastructure management.

7.2 Future Work

Although *testBench* achieves the core objectives it set out to accomplish, there is still plenty of room for growth. Like any practical tool, its development doesn't end with the first working version. Instead, this marks a solid foundation on which further improvements, optimizations, and new features can be built.

At its current stage, there are still key areas where testBench could evolve further, making meaningful progress toward becoming a widely adopted, standardized solution.

7.2.1 Technical Improvements

From a technical standpoint, several enhancements could strengthen testBench's performance and scalability:

- Support for multiple runtimes: While current support is tailored to Java (via JUnit), extending support to other languages and testing frameworks (e.g., Python with Pytest, JavaScript with Jest) would make the platform more broadly usable.
- Caching and layer reuse: Implement smarter Docker image caching to avoid redundant rebuilds for unchanged containers, reducing execution time and disk I/O.
- Error monitoring and logging: Add centralized logging and alerting to track backend failures and improve observability.

7.2.2 Feature Enhancements

As for features, several additions could improve both the flexibility and the pedagogical reach of the platform:

- Course units and progression tracking: Allow instructors to define course units or modules, with exercises that unlock progressively as students complete prerequisites.
- Exercise duplication and editing tools: Enable instructors to clone, edit, and version existing exercises from the UI, making it easier to reuse or refine content across terms.
- Expanded GUI support: Improve the reliability of GUI-based exercises and provide frontend indicators for GUI availability and status.
- Offline mode or desktop client: Explore the possibility of a lightweight local client or progressive web app that allows students to work offline and sync when back online.
- Submission history and comparisons: Allow students and teachers to view previous submissions, compare changes, and roll back to earlier versions if needed.

7.2.3 Usability and Interface

In terms of usability, refining the interface could significantly enhance the experience for both students and teachers:

- Improved error feedback: Provide more detailed and user-friendly error messages when containers fail to launch or when tests don't run correctly.
- Test result breakdown: Enhance the frontend grading display with collapsible sections, clearer point attributions, and links to related test cases or code.
- Accessibility improvements: Ensure the platform meets accessibility standards to support users with disabilities.
- Mobile responsiveness: While coding is best done on desktop, adapting key pages (e.g., dashboards, grade views) for mobile use would improve accessibility.

7.2.4 Adoption

To move testBench toward broader adoption and real-world classroom use, several steps could be taken:

- Pilot programs with educators: Partner with instructors to run small-scale pilots and

gather real user feedback on workflows, pain points, and feature needs.
- Documentation and onboarding: Create thorough guides and tutorials for both teachers and students, including walkthroughs of setup, exercise creation, and troubleshooting.
- User roles and permissions: Add more granular control over user roles (e.g., assistant teachers, reviewers, students) to support a wider variety of teaching models.
- Integration with LMS platforms: Develop optional integrations with learning management systems like Moodle or Canvas to streamline user management and grade syncing.

The hope is for the development of *testBench* to be a step toward more scalable, fair, and engaging computer science education. It sets the groundwork for future innovations in how students learn, how teachers teach, and how we all think about the role of automation in the classroom.

7.3 Personal Conclusions

Previously to working in *testBench*, I had already gained experience in software development during my internship as a developer. However, I had never had the opportunity to take full ownership of a real problem and to be solely responsible for every decision required to arrive at the most effective solution.

testBench offered me an opportunity to further explore a sector I had not initially considered pursuing, one that has turned out to be both intellectually stimulating and deeply motivating. Engaging with the challenges it presented sparked a genuine interest in creating practical solutions that can improve people's daily lives. Though just a step along the way, this project has reaffirmed my desire to keep building, learning, and contributing meaningfully through technology.

All in all, I am very satisfied with the journey that *testBench* has represented. Beyond the results themselves, it has been an incredibly rewarding experience to explore and enhance its potential alongside my thesis advisor, Juan López Rubio, who has been an unwavering source of ideas and support.

References

[1] Codio. “Grading and student feedback challenges faced by cs instructors. ”[Online]. Available: https://www.codio.com/blog/grading-student-feedback-challenges-cs-instructors.

[2] H. Liao, P. Denny, A. Luxton-Reilly, and E. Tempero, *The role of automation in undergraduate computer science*, https://scispace.com/pdf/the-role-of-automation-in-undergraduate-computer-science-2885l4apeu.pdf, 2019.

[3] LeetCode. “Leetcode quickstart guide. ”[Online]. Available: https://support.leetcode.com/hc/en-us/articles/360012067053-LeetCode-QuickStart-Guide.

[4] GitHub, Inc. “About github classroom. ”[Online]. Available: https://docs.github.com/en/education/manage-coursework-with-github-classroom/get-started-with-github-classroom/about-github-classroom.

[5] Gradescope, Inc. “Gradescope autograders documentation. ”[Online]. Available: https://gradescope-autograders.readthedocs.io/en/latest/.

[6] Replit, Inc. “Getting started with replit. ”[Online]. Available: https://docs.replit.com/getting-started/intro-replit.

[7] IBM. “What is a three-tier architecture? ”[Online]. Available: https://www.ibm.com/think/topics/three-tier-architecture.

[8] N. Sundaramoorthy, *UML Diagramming: A Case Study Approach*. CRC Press, 2023, ISBN: 9781032120782.

[9] M. B. Braun, *RESTful API Design: Best Practices in API Design with REST (API-University Series Book 3)*. API-University Press, 2015, ISBN: 9781514735169.

[10] M. Jones, J. Bradley, and N. Sakimura, *Json web token (jwt)*, https://datatracker.ietf.org/doc/html/rfc7519, RFC 7519, 2015.

[11] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, ISBN: 9780321127426.

[12] S. Ramírez. “Fastapi documentation. ”[Online]. Available: https://fastapi.tiangolo.com/.

[13] Python Software Foundation. “Python 3 documentation. ”[Online]. Available: https://docs.python.org/3/.

[14] T. Christie. “Starlette documentation. ”[Online]. Available: https://www.starlette.io/.

[15] S. Colvin. “Pydantic documentation. ”[Online]. Available: https://docs.pydantic.dev/latest/.

[16] ASGI Project. “Asgi: Asynchronous server gateway interface. ”[Online]. Available: https://asgi.readthedocs.io/en/latest/.

[17] MongoDB, Inc. “Mongodb documentation. ”[Online]. Available: https://www.mongodb.com/docs/.

[18] Docker, Inc. “Docker documentation. ”[Online]. Available: https://docs.docker.com/.

[19] Open Dev Containers Specification. “Development containers specification. ”[Online]. Available: https://containers.dev/.

[20] Google. “Flutter documentation. ”[Online]. Available: https://docs.flutter.dev/.

[21] Google. “Dart documentation. ”[Online]. Available: https://dart.dev/docs.

[22] The Markdown Guide. “Getting started with markdown. ”[Online]. Available: https://www.markdownguide.org/getting-started/.

[23] JUnit Team. “Junit 5 user guide. ”[Online]. Available: https://junit.org/junit5/docs/current/user-guide/.

[24] IETF, *Rfc 6455 - the websocket protocol*, https://datatracker.ietf.org/doc/html/rfc6455, Internet Standard, 2011.

[25] Docker, Inc. “Docker compose documentation. ”[Online]. Available: https://docs.docker.com/compose/.

[26] Traefik Labs. “Traefik documentation. ”[Online]. Available: https://doc.traefik.io/traefik/.

[27] R. Soni, *NGINX: From Beginner to Pro*. Apress, 2016, ISBN: 9781484216571.