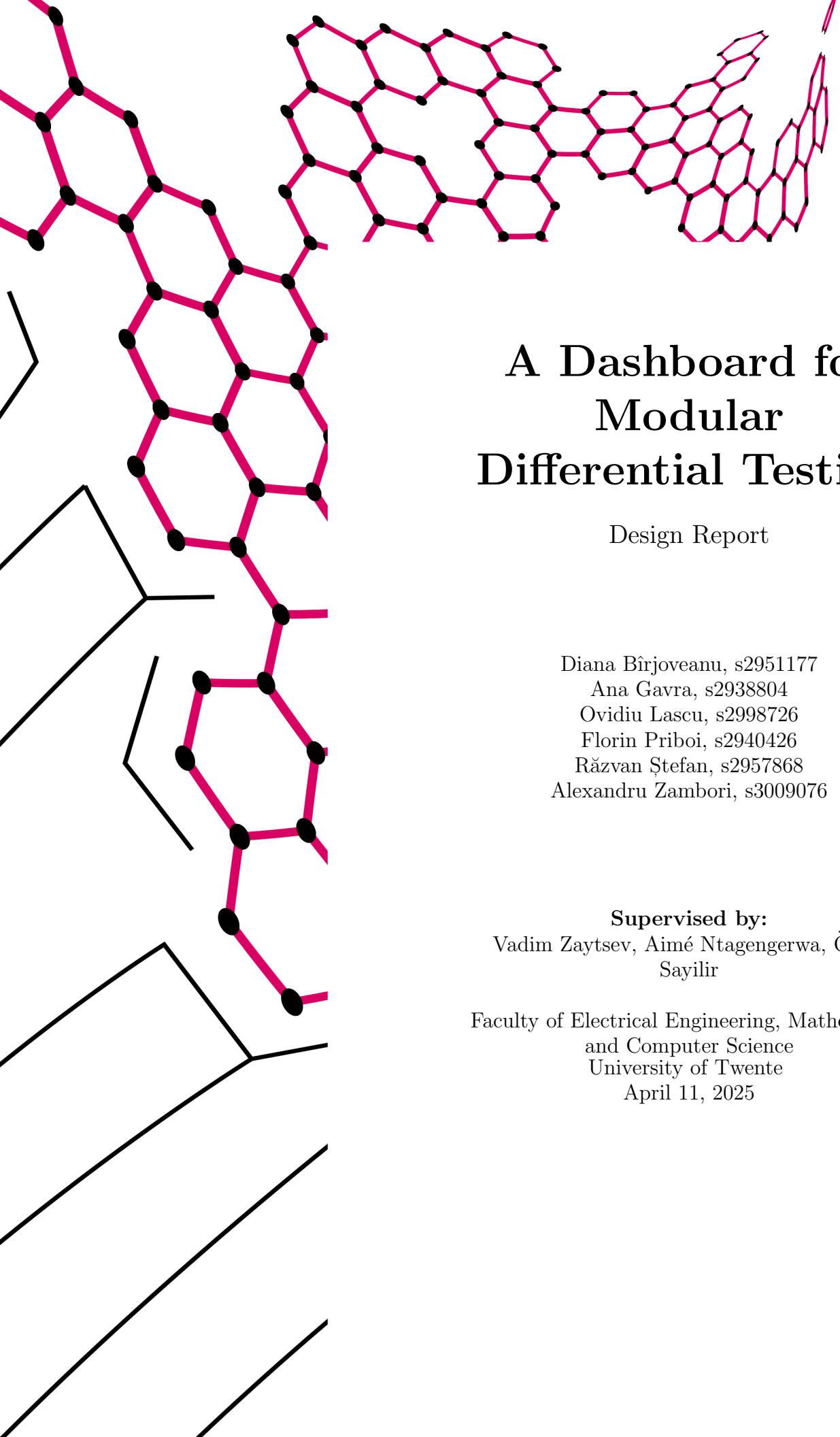

A Dashboard for Modular Differential Testing

Design Report

Diana Bîrjoveanu, s2951177
Ana Gavra, s2938804
Ovidiu Lascu, s2998726
Florin Priboi, s2940426
Răzvan Ștefan, s2957868
Alexandru Zambori, s3009076

Supervised by:
Vadim Zaytsev, Aimé Ntagengerwa, Ömer Sayilir

Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
April 11, 2025

Contents

1 Introduction

For this Design Project, the team has chosen to create a dashboard for modular differential testing. This application is used to compare multiple pieces of code, possibly in different languages and versions, and for software testing automation. With the help of this product, users will be able to simplify the process of software testing by being able to automatically test different implementations of the same program in various languages to find the optimal solution for their projects. With access to detailed metrics and differences, users can choose the most efficient algorithm and/or language for their task, resulting in better productivity and product performance.

This report will describe the product's design process, including research into available technologies, requirements engineering, and our preliminary and final design choices, from a global and in-depth perspective. Choosing the appropriate tech stack involved conducting thorough research on available options while aligning them with our requirements. We will go over the different types of requirements and their acceptance criteria, as well as how these requirements influenced our choice of tools. The final mechanism behind the application is explained in great detail, providing a comprehensive description of how each component operates and interacts with the overall system. Thorough testing has been done to ensure proper functionality of the final product, which is explained in a later chapter. Finally, we suggest some future improvements currently outside our project's scope.

2 Domain Analysis

Before starting implementation, the team needed to understand the scope and stakeholders of the project in order to get a good grasp of the problem at hand. This chapter outlines the domain within which the problem exists, reviews already existing work and also provides an analysis of stakeholders.

2.1 Scope of Project

Software testing is one of the pillars of software engineering and a necessary step in providing correct software. There are several stages involved in testing a program at various levels of implementation. Currently, in order to verify different implementations of the same program, frameworks and libraries are available, or the user can also manually write the tests they need. Automated testing methods have been developed to effectively test individual applications, ensuring that they function correctly and meet quality standards. However, there is still no way of conducting automatic testing between different implementations of the same algorithm, written in different programming languages. Thus, the system aims to fill this gap and facilitate differential testing of algorithms across various languages.

2.2 Stakeholder Analysis

Following our initial meeting with the clients, during which we discussed the scope of the project and further details, we identified the primary users of the system as individuals with a technical background. These users are already proficient in the programming language they intend to test and possess a solid understanding of the performance and output metrics relevant to their code. Consequently, the system is designed with the strong assumption that users can interpret execution results without the need for extensive guidance or explanation.

2.3 Already Existing Work

To understand modular differential testing (MDT), we decided to research existing systems and applications that implement it.

The most relevant paper found is CrossASR++, which uses MDT for automatic speech recognition (ASR). This tool compares multiple ASR outputs to identify inconsistencies. CrossASR++ is also able to generate tests automatically, by starting with a text dataset and using a Text-to-Speech system to convert the text into audio. In this context, the generation of automatic texts is important because manually writing tests for ASR is tedious and time-consuming [2]. We took inspiration from generating tests automatically and decided to implement such a system in our application, as this could facilitate finding edge cases and unusual behaviour.

Another application that makes use of MDT is using it in finding bugs in C/C++ compilers. In their paper, Shaohua Li and Zhendong Su introduce CompDiff, a methodology that leverages differential testing to detect unstable code in C/C++ programs. By examining discrepancies in outputs produced by different compiler implementations, CompDiff effectively identifies code segments susceptible to undefined behavior [4].

3 Requirements

Based on the meeting we had with the clients, the project description offered, and the research into existing work, the team has come up with a list of requirements for the product. These have been broken down into user requirements and system requirements. Before specifying our requirements, the team needed to decide how they would be handled during the project following the Agile methodology. Further, a formulation and prioritization technique has been decided upon to ensure proper handling of requirements.

3.1 Comments on the Agile Development Method

At the beginning of the project, we decided to adopt the Agile framework, specifically working under the SCRUM development method [5]. This allowed us to get constant feedback on progress and revisit requirements as we move further with the implementation, or the client proposes new requirements. A sprint lasts two weeks, and we had biweekly sprint reviews with the client to present progress and future planning for the next sprint to come. In between sprint reviews, informal meetings were conducted where we discussed more technical issues with our supervisor.

The overall project was divided into three main phases. The first phase was Planning, which included deciding the project and meeting with the clients to discuss details. Following this, the Design phase will start, which includes requirement engineering, diagrams of the system overview and wireframes for the user interface. As the Design phase finishes, the Implementation one begins. Alongside this, we will also start testing the implemented code until the ninth week. The last week will be reserved for reviewing the product and possibly deploying it.

A more detailed presentation of the team's planning is included in the Project Proposal. The Project Proposal is a separate document, that outlines the project timeline, planned deliverables and initial risk assessment. This document serves as a contract between the development team and client. As a standalone document, the Project Proposal is not included in this report.

3.2 Requirement Specification, Formulation and Prioritization

The first phase of Requirements Engineering [3] will be specifying the user requirements based on the stakeholder analysis and all details discussed with the client about the product.

Each user requirement is expressed in the form of a user story. This way, it is made clear who the user is, what they need and why they need it. In our case, all user stories are written from the point of view of the "user", since all our users are technical and don't need to be split into specific groups. In order to fully establish the details of how each user requirement will be accomplished throughout designing and developing the product, each requirement's flow is described through use cases. Further, each use case has been given a description in order to ensure full clarity on what is expected. Therefore, when accepting our requirements, the client can use the use cases as acceptance criteria.

Moving forward in the design process, identified user requirements are mapped to system requirements. The resulting system requirements are prioritized using the MoSCoW [1] analysis technique. This way, the client can clearly understand what the team considers to be within or outside the scope of this project, but also what the limitations of implementing the product are. Some of the system requirements are provided with a list of sub-requirements, in order to make it as clear as possible to the client what the requirement entails.

3.3 User Requirements

1. As a user, I want to have an interface that allows me to compare at least two different programs, so that I can evaluate their differences effectively.
2. As a user, I want to be able to choose the configurations of each code cell, so that I have flexibility in testing.
3. As a user, I want to be able to choose what to display between performance metrics and test results, so that I only see the information I'm interested in.
4. As a user, I want to be able to upload a code file, so that I can easily test already existing programs.
5. As a user, I want to be able to write code in the code cells, so that I can easily change my code during testing.
6. As a user, I want to be able to manually write my own test cases, so that I can test specific cases I'm interested in.
7. As a user, I want to be able to understand and easily navigate the interface, so that I have a smooth experience.
8. As a user, I want to have access to a user manual with clear instructions, so that I can understand the system and troubleshoot potential issues.
9. As a user, I want to be able to change interface settings such as font size and theme, so that I can work in my preferred style.
10. As a user, I want to be notified by the system if a test case has a different output, so that I can catch unusual behaviour quickly.
11. As a user, I want to have the option of generating test cases based on the function signature, so that I can find potential unusual behaviour and facilitate my testing experience.
12. As a user, I want to be able to save my projects, so that I can work with them another time.
13. As a user, I want to be able to add my own plugins to the application, so that I can add additional features according to my needs.
14. As a user, I want to have syntax highlighting on the provided code, so that I can easily read my code.

As mentioned in the previous section, each user requirement's flow has been broken down into use cases, with each one of them having their own description for full clarity. A table of all use cases for each requirement can be seen below in Table 1, and their thorough descriptions can be found in Appendix A, Table A.1.

No.	Requirement	Use Case(s)
1	As a user, I want to have an interface that allows me to compare at least two different programs.	Open the app, Open an existing project, Create a new project, Configure project, Upload configuration, Upload file, Write code in cell, Run the code cells, Choose output type, Analyze results
2	As a user, I want to be able to choose the configurations of each code cell.	Configure project, Upload configuration
3	As a user, I want to be able to choose what to display between performance metrics and test results.	Run code cells, Choose output type
4	As a user, I want to be able to upload a code file.	Open app, Upload file
5	As a user, I want to be able to write code in the code cells.	Open app, Open existing project, Create new project, Configure project, Upload configuration, Write code in cell
6	As a user, I want to be able to manually write my own test cases.	Open app, Select file, Upload file
7	As a user, I want to be able to understand and easily navigate the interface.	Open app, Upload file, Write code in cell, Configure project, Upload configuration, Run code cells, Choose output type, Analyze results, Save project
8	As a user, I want to access a user manual with clear instructions.	Open app, Press HELP button
9	As a user, I want to be able to change interface settings such as the font size and theme.	Open app, Press settings, Select "Interface Settings", Change interface aspect, Save changes
10	As a user, I want to have the option of generating test cases based on the function signature.	Write code in cell, Upload file, Configure project, Upload configuration, Generate test suite
11	As a user, I want to be notified by the system if a test case has a different output.	Configure project, Upload configuration, Write code in cell, Run code cells, Choose output type, Analyze results
12	As a user, I want to be able to save my projects for later use.	Press "Settings", Select location in computer, Save project
13	As a user, I want to be able to have clear intrusctions about how to add new features to the application, such as a new language.	Open app, Extend Feature
14	As a user, I want to have syntax highlighting on the provided code.	Configure project, Upload configuration, Upload file, Write code in cell

Table 1: List of Use Cases

3.4 System Requirements

Must

1. The system interface must have 2 code cells.
    1.1 The two code cells can be run at the same time.
    1.2 Each code cell should have an individual configuration chosen by the user (e.g., language, version, external libraries).
    1.3 Each code cell should run on the specified test cases.
2. There must be a configurable dashboard based on multiple tiles.
    2.1 The user should be able to choose the metrics and outputs showcased in the code comparison.
    2.2 The system should showcase at least the following: performance metrics (e.g., execution time, memory usage) and test suite results.
3. The user must be able to upload code to the dashboard's code cells.
    3.1 The user should be able to change the input program within the code cell.
    3.2 The user should be able to upload a code file.
    3.3 The user should be able to insert code in the code cell.
4. The user must be able to manually upload an input and output file for specific test cases.
5. The system must be able to choose a compiler according to the programming language chosen by the user.
6. The system must be user-friendly and intuitive.
    6.1 The user should be able to learn how to use the system within 5 minutes.
    6.2 The buttons should be easily recognizable by their icons.
    6.3 The user should easily distinguish from which code cell the results are coming from.
7. The system must be a standalone app.
8. The user must have a clear and concise set of instructions for using the app.

Should

9. The system should be able to compare the outputs of the program with the expected outputs from the test cases.
10. The system should be easily scalable and extensible.
    10.1 The code base of the system is well-encapsulated and uses interfaces.
    10.2 The system architecture allows for the addition of new features.
11. The system interface must be visually accessible.
    11.1 The user has the option of changing the font size.
    11.2 The user has the option of choosing between dark mode and light mode.

Could

12. The system could be able to generate test cases based on the given code and function signature.

13. The user could be able to save their projects locally.
14. The user could be able to close the app and not lose their local project.
    14.1 There should be a warning when closing the app without saving.
15. The user could be able to write and add plugins to the application.
16. The text within the code cells could have syntax highlighting.

Won't

17. The application won't be able to be integrated as an extension to an already existing IDE.

3.5 Requirement Acceptance

Progress is tracked during the development of our product through the acceptance of use cases. The client reviews and accepts use cases based on their description, which can be found in Appendix A, Table A.1. Since system requirements are derived from user requirements, accepting use cases allows us to verify the completion of system requirements. For non-functional system requirements, such as **Must 6**, usability testing will be conducted. More details about Usability testing are given in Chapter 6.

4 Global Design

To identify the best tools and design a system that meets the most requirements, the team conducted extensive research into the available options. After deciding which tools and programs would allow us to design the most viable application, each option has been assessed, alongside with the motivations to use them. In the end, an overarching design is proposed.

4.1 Research into Compiler Approaches

The main technical challenge of this project is executing code provided by users. The system must be robust enough such that it handles various inputs and can obtain several execution metrics, such as execution time, memory usage, and CPU usage. Additionally, selecting the right approach for this problem is crucial in order to ensure the application is easily scalable. A key requirement is that developers should be able to add new languages. The method we choose to execute user code plays a crucial role in the process of incorporating a new programming language.

After researching possible ways of executing unknown code, various techniques have been discovered for compiling and running code from different programming languages. Each approach will be identified and further elaborated on.

4.1.1 Docker Containers

The first approach considered was running the code in Docker containers. Docker is a software that virtualizes the computer's file system and is used to run software packages called containers. This solution has several advantages.

Because Docker is a critical tool in software development, there are already many well-maintained images of popular programming languages. An image contains instructions to build a container, and in our case, it includes all the tools required to create a virtual environment in which we execute code from a specific programming language, facilitating requirements **Must 1.1, 1.2, 5, Should 10.2**. Therefore, scaling the system by adding support for more languages is straightforward.

Using containers is also beneficial because it facilitates parallelism and obtaining execution metrics. It is simple to manage multiple containers simultaneously on separate threads, thus improving performance significantly, and fulfilling requirement **Must 1.1**. Also, all the official images are made on Linux distributions, which have packages such as "time" that make it easy to run a process and obtain its metrics, making possible the realization of requirement **Must 3**.

Another relevant factor is the extra layer of security provided by containerization. Although it is not an important requirement for our clients, it is still relevant to consider. By executing code in virtual containers, we separate the piece of code from the critical systems on the user's computer. Thus, if a user tries to execute malicious code, it will not have access to the server, only to its assigned container.

Although this is an interesting option, it comes with some downsides. The main downside is that the user would be required to start the Docker Engine before actually using the dashboard. This would make requirement **Must 7** impossible to achieve. When discussing this possibility with the clients, it was made clear that they would be fine with having Docker, as long as the user is not aware of it. Therefore, the idea of using basic Docker was scrapped.

4.1.2 C# Compiler

Another option would be to use the C# open-source compiler. From our discovery, this option would restrict the amount of languages that the product could support. Languages have been developed to be compiled by Roslyn, such as IronPython, but they don't support commonly used libraries such as numpy or pandas. Additionally, another factor that deterred us from using the C# compiler was our lack of experience in working with C#.

4.1.3 GraalVM

Another version with similar issues is GraalVM, since it supports only a few languages. Some important programming languages that it lacks are C and C++, which we consider very important to handle.

4.1.4 Podman

When looking for alternatives to Docker that fix the issue of requiring the user to manually start the Docker daemon, we found Podman. Similar to Docker, Podman is software that allows containerization, but it doesn't require a daemon, so the user doesn't need to take any steps to activate it. In every other way, Podman works exactly as Docker, thus keeping all the previously mentioned advantages while remaining easy to use. This is why we decided to use Podman in our solution to execute code. Podman is also available for Windows, Linux and macOS.

Even with the decision to use Podman to containerize user's code, we still need more in order to actually execute the code. When using the system, a user would write a piece of code in a specific programming language, but that piece of code is never called. Therefore, if we just compile and execute the raw user code, nothing will actually happen. A solution needs to be found to transform the raw code into an injected code that makes the calls to the function that is being tested. Several techniques of code injection have been considered.

4.1.5 Compiling to LLVM and injecting LLVM IR code

Besides using Podman, we planned to use LLVM. LLVM is software that compiles code from many popular programming languages into a language-agnostic intermediate representation. This representation will easily allow us to run the original code from within the Podman container.

However, after experimenting with LLVM IR code injection, we realised that for higher-level programming languages, the compiler's Frontend added a lot of overhead and helper methods when compiling code into LLVM IR. This made injecting the user code into a main function very difficult, especially when the input is something more than a primitive type. The complexity was too high, and we decided that it was outside our boundaries to try to implement it within the allocated time, since we lack the required experience to achieve the proper result. Besides the technical issues, LLVM is a great tool to use for the programming languages that are natively compiled in LLVM. However, the system we are developing should support multiple programming languages, such as Python and Java, for which there was little to no support in LLVM.

4.1.6 Injecting at the Language Level

Ultimately, we decided that code injection would happen at the language level, with different injectors for each language that the application supports. This makes it easier for the user to add support for other languages. The only step would be to extend the Injector class with a new class with the specific syntax of that language, facilitating requirement **Should 10.2**.

4.1.7 Final Compiler Approach

Based on the list of possible approaches, there is an obvious tradeoff to be made in either direction. In terms of containerization, Docker has more support and reliability, but needs to be run as a separate application in order to provide the functionality. In contrast, Podman can be run completely as a CLI, which makes the user experience nicer, better satisfying requirement **Must 7**, but offers less support and available information in case of errors.

Approaches like C# compilation, GraalVM and LLVM provide additional functionality that may allow the user more flexibility on the information that can be retrieved about the program. This flexibility sits in contrast to extensibility towards other programming languages in the case of the C# and GraalVM and complexity in the case of LLVM. Code injection is a simpler approach that is also very extensible, thus requirement **Must 10** is satisfied, however there is an inherent limit to what information can be retrieved about the program.

Due to considerations for time, extensibility and user experience, the final approach is that of language-level injection for compilation and Podman for containerization.

The final execution pipeline is as follows:

1. The user code, with all the language specifications and requirements, is sent to the server.
2. The server then sends this information to the **Container Manager**.
3. The **Container Manager** keeps track of running containers and their images and checks if there are already containers with the same specifications. After finding or creating a suitable image, it then sends the code to the container interface.
    3.1. If there is no suitable container, the program creates a directory specific for mounting, then adds in that directory the Dockerfile and the injected user code. Then the image is built and the container automatically runs.
    3.2. If the **Container Manager** finds a fitting container, it then sends it the new data, at which point the existing code is deleted, the new one is added, and it compiles and runs it.
4. The output is then processed by the **Result Parser** and sent back to the Frontend.

4.2 Architectural Design Choices

4.2.1 Python

For the realization of the Backend part of the system, we have decided to use Python, as it offers a myriad of libraries that would ease the implementation of the server and testing. We also felt the most comfortable with coding in Python, so the decision was clear. For the version, we ended up using 3.13.2. From version 3.13 onwards, the implementation of the locks for multi-threading applications was redone, and this new version facilitated a correct implementation. Since Python is widely known and easily accessible, having the Backend codebase written in it facilitates requirements **Should 10** and **Could 15**.

4.2.2 Podman

As discussed above, Podman is used for containerization and for getting the correct compilers, interpreters and any other necessary software for code execution. This works towards achieving requirements **Must 1.1, 1.2, 5 and 7**. We have decided to go with the latest stable release. Although it is not a hard requirement for our product to be compatible with all operating systems, documentation shows that Podman is compatible with all OS's. This way, through the way we use Podman, our product should be compatible. Moving forward in the project, although Podman works with Windows operating system without issues, we have experienced major issues when trying to install it on Mac. From our research, it seems to be an unfixable error from our side. Further details about these issues are given in Chapter 7 of this report.

4.2.3 Flask

To satisfy the requirement of the system to be scalable and extensible (**Should 10**), we have decided to encapsulate the functionality of the Backend within a server. This in turn would allow for the application to be hosted and be made available online as a service. Flask was chosen as a framework for the server, as it offers all the necessary HTTP methods handling. The version chosen is also the latest stable release.

4.2.4 Pytest

In order to write the integration tests, we used Pytest, with the latest release. This library automatically executes all the written tests and also has the option of having a set-up and cleaning after each test, ensuring that there are no side effects from executing the tests. Another used feature was patching environment variables, allowing for mocking some Podman errors that would only appear if the Podman system is faulty or non-existent.

4.2.5 Colorlog

Colorlog is a Python library that allows for coloured outputs in the terminal. This was needed for debugging purposes and to improve the readability of the outputs from the server.

4.2.6 React

For the Frontend part of the project, we decided to use React JS, with its latest release. The main reason why we chose React was that it offers a lot of flexibility when it comes to functionality but also when it comes to the look of the final application, which all contribute to requirements **Must 1, 2, 6, 8, Should 11 and Could 16**. React offers access to multiple component libraries that will facilitate coding, but also to frameworks that will make porting the application from a web application to a desktop one easier. Further details about conversion to a desktop app and packaging will be discussed further in the next section.

4.2.7 Frontend-Backend Integration

The initial implementation of the system intentionally follows a traditional Server-Client format. This allows for greater extensibility in terms of porting the system to a Web-based version (**Should 10**). However, this means that additional tooling is necessary for packaging everything into a standalone application.

4.2.8 Electron

Electron is a Node.js based framework for porting Web applications to desktop. We have used it as a means of packaging our system, such that requirement **Must 7** can be satisfied. The Frontend is built into static files, so that it can be locally served, while the Backend is spawned as a subprocess. Additionally, the Electron application will try installing an appropriate Python environment with all its necessary dependencies for the Backend to run. The user is expected to install Python 3.13.2 and Podman by themselves.

4.2.9 Electron Builder

Electron Builder is a framework that can generate installers for an Electron application. The resulting installer comes with its own Node.js runtime and Chromium image, and as such, the user does not need to have Node.js, Electron or React installed on their system for the installer and application to work.

4.3 System Description

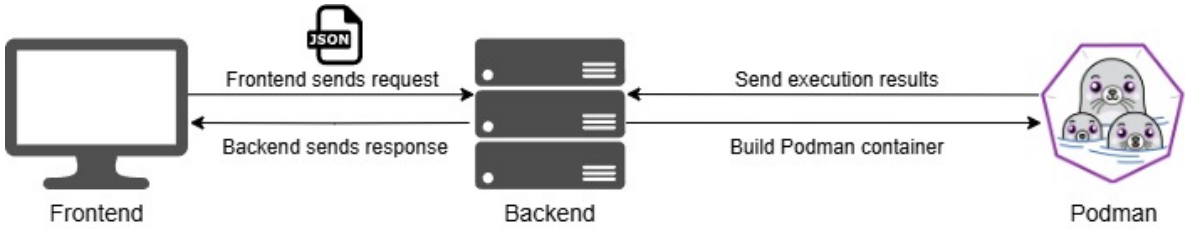

Figure 1: High-level system overview

Based on everything that has been discussed in this chapter of the report, a high-level overview of the system is given with respect to the system requirements, and can be visualised in Figure 1. The system will work as an app hosted locally on the user's machine (**Must 7**). When starting the app, the user will see a dashboard with two main cells in which code can be added (**Must 1**). Then, the user will add code, either by uploading files or writing code in the cells (**Must 3**). Additionally, the user will also fill in the necessary configuration, such as programming language, desired version, required libraries and others (**Must 1.2, 5**). After these steps, the user can press the "Run" button in order to execute the provided code (**Must 1.1, 1.3**).

The Frontend will send a request to the Backend with all the necessary information to execute the code in the desired environment. When the request is received, the Backend will start by putting the code which needs to be executed in a file and generate the required Dockerfile from the given specifications. The system will then use Podman to start a container with the executable code. The code file will be run in the Podman container and output the results. The Backend will process the outputs and send a response with the results of the executed code back to the Frontend.

Based on the response, the Frontend will display the results to the user. In case the code gives an error, the Frontend will show the logged exceptions and error codes. Otherwise, if the code runs, the dashboard will show performance metrics such as execution time, memory usage and results of the test cases (**Must 2.2**), highlighting any mismatches between the expected output and the actual output of the executed code (**Should 9**). Figure 2 provides a visual summary of the process.

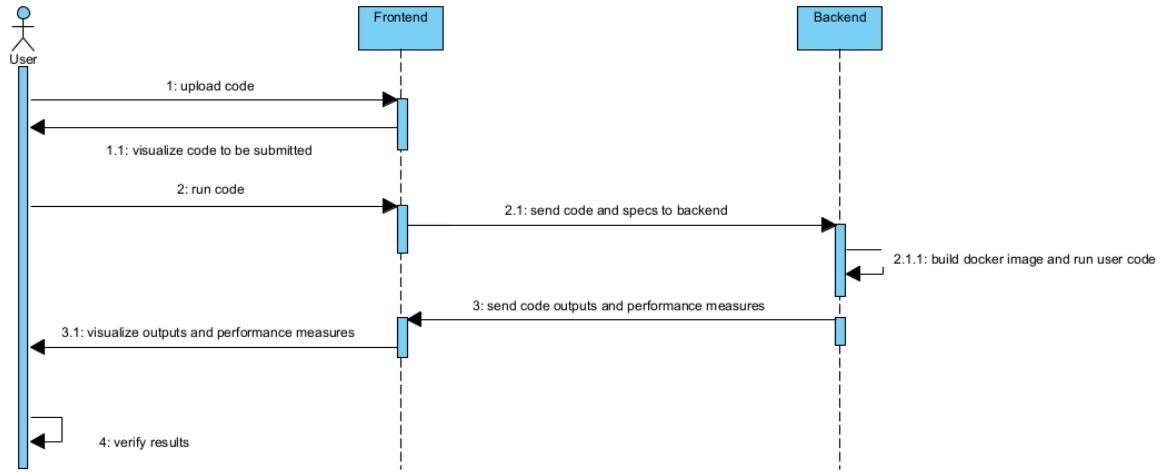

Figure 2: Sequence diagram for Frontend - Backend communication

4.4 UI/UX

4.4.1 Understanding the Stakeholder

The first step in creating a smooth user experience accompanied by an easy-to-understand interface is understanding the stakeholders and their needs. From the meetings conducted with our clients and supervisor, the team has received a lot of liberty for the UI design, with the only "hard" requirement being that our product should have the same feel as the software they usually use, such as code editors or IDEs.

Hence, together with previous stakeholder analysis, the general idea for the product implied simple but information-heavy interfaces, with design choices similar to those made by products with predominantly technical users. This approach ensures that no matter how much information is given to the user, it will be presented in a familiar way, thus achieving a sleek experience.

Taking into account the priorly discussed stakeholders' needs and the identified use cases (Table A.1), the composition of the final product was decided to include four main components that achieve all the system's main functionalities. Each component is identified and elaborated on.

4.4.2 Home Page

Similar to development-oriented desktop applications, the user must be able to choose between creating a new project (**Use Case 3**) or opening an already existing one from their machine (**Use Case 2, 18**). Therefore, after clicking on the attributed desktop icon (**Use Case 1**), the user is welcomed by a Home Page that presents the two aforementioned possibilities. However, the option to open an already existing project will continue to be accessible through other pages of the application so as not to impose a return to the Home Page on the user every time they want to do so.

4.4.3 Code Page

After passing the Home Page, the user is redirected to the Code Page, which serves all use cases for visualizing results and interaction with code. Here, the user can insert code into each available code cell based on their needs (**Use Case 8, 7a**), but also set the configuration of each individual code cell (e.g. language, version, and compiler) **(Use Case 5, 6, 7c)**. After running the code **(Use Case 10)**, the user is presented with two options: either to view the performance metrics for each cell individually **(Use Case 12b)** or to see the differential results comparing all cells **(Use Case 12a)**. Through the differential results modal, the user can also compare their outputs to the expected ones, or to see which code cells have unusual behaviour when auto-generated inputs that do not have a specified expected output are used **(Use Case 9)**. Furthermore, the page allows uploading a file containing the code **(Use Case 7a)**. This allows the user to upload code files directly, avoiding the need to rewrite or repeatedly copy-paste code into the interface.

4.4.4 Settings Page

The Settings Page provides the user access to cell functionality and interface configuration, program run process changes, and general application settings such as adjustments to the color theme (**Use Case 16**), assignment of timeouts, generation of test cases (**Use Case 9**), and definition of code cell signatures (**Use Case 5, 7c**). Further, the user can also select how the final results of performance metrics and terminal outputs will be displayed (**Use Case 11**). All presented use cases have been moved to the Settings Page, as the user interacts with these options only occasionally, making frequent display unnecessary.

4.4.5 Sidebar Component

To provide easy access to all the aforementioned pages (**Use Case 14**), plus the functionalities that are used more frequently, such as accessing the user manual (**Use Case 13**), importing local projects (**Use Case 2**), saving current ones (**Use Case 4**), opening the differential output modal (**Use Case 12a**), and adding a new cell, a sidebar acting as an always-available menu for all these features has been integrated.

4.4.6 Preliminary Design Choices for UI/UX

Based on the user's perspective and decisions regarding how the system's use cases should be implemented, the team developed a set of wireframes, which were later presented to the client for feedback. The wireframes serve as Lo-Fi prototypes that assess the potential user experience of the product since there are many ways in which the pages and their components can be visualised.

The first wireframe (Appendix E.1) that was presented received meaningful remarks regarding how we use the available screen space and the induced confusion of our layout. Therefore, a second wireframe (Appendix E.2) was created in response to the received feedback. In the second iteration, the user can see all available options in a sidebar menu without unnecessary empty space. Alternatively, another idea that was discarded was a configuration tab present before access to code insertion. Instead, it was modified into a drop-down configuration menu, specific to each code cell. By doing this, the client can change a cell's configuration at any time while using the application. The aforementioned wireframes can be found in Appendix E.

Based on the feedback loop generated by our wireframes, a final interactive Figma prototype was presented (Appendix E.3). This development step resulted in favorable observations of our UI and approval of our design decisions from the client. Ultimately, we moved on to implementing this interface. The final look of the interface can be found in Appendix F.

5 In Depth Design Choices

This chapter contains detailed explanations of the Backend execution pipeline, alongside with a detailed description of each component or mechanism of the system. For the Frontend, all the functionalities have been expanded upon and the order of execution for tasks has been presented. Each functionality and component has been linked to the requirement it helps satisfy.

5.1 Backend Overview

5.1.1 Execution Pipeline

When the Backend receives a request, it is forwarded to the Container Manager. Afterwards, there are two possible execution paths in the pipeline. The Container Manager first checks for suitable containers by comparing the language specifications of the request to those of the available containers. An available container is one that is not running code. If there is a suitable one, it adds the code from the request into the container and executes it. If none is found, a new one will be created. The output is then differentiated (**Should 9**) and forwarded to the Frontend for display. This whole process is modeled in Figure 3.

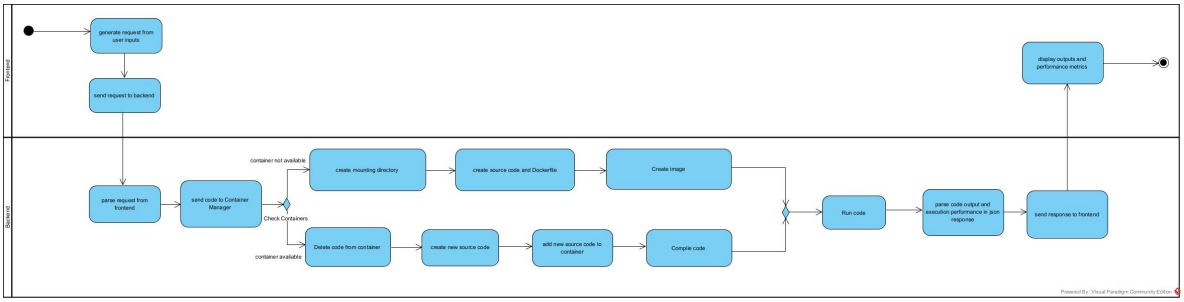

Figure 3: Activity Diagram of Execution Pipeline

5.1.2 Components Overview

The system comprises multiple components, each offering a higher level of abstraction from the Podman system, in order to comply with **Should 10**. In the following section, we will explain each component and some technical mechanisms implemented to make the system work. The simplified class diagram in Figure 4 shows how these components are interconnected. This diagram shows the core functionalities of the server components. The full class diagram can be seen in Appendix C.

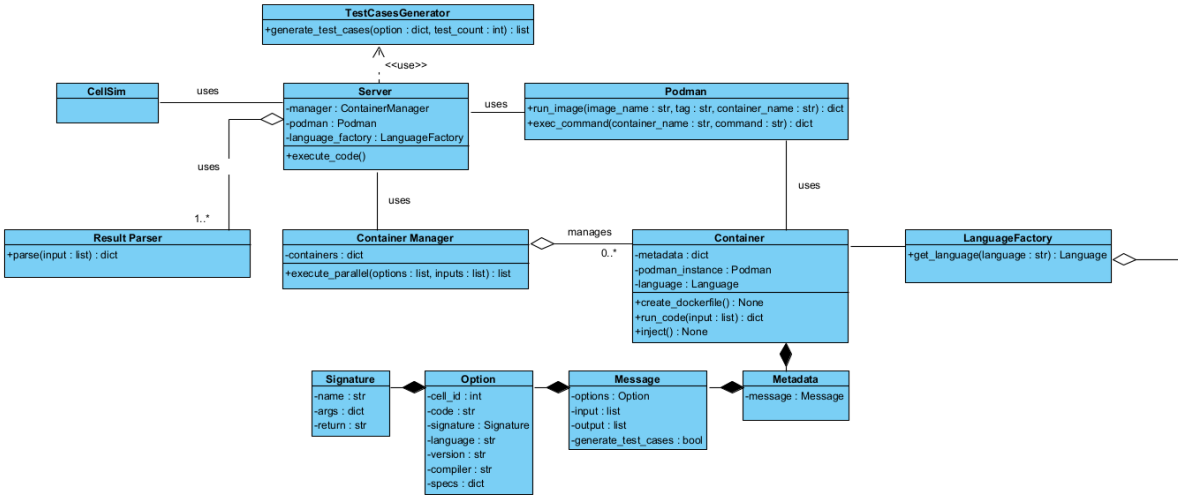

Figure 4: Simplified Class Diagram of Server Component

5.1.3 Containerization

To handle a multitude of programming languages and their respective environments in a centralized and extendable way, containerization has been chosen as a mechanism central to the project's Backend. As expressed earlier, containerization brings benefits to the project structure: horizontal scalability of the code structure and system security during execution. The security aspect is only secondary in our specific use case, so we do not discuss it further. The more interesting facet of using containerization is the extensibility it provides (**Should 10**). A container is mainly represented by a relatively small footprint, UNIX-based virtual machine. By using a virtual machine that is UNIX-based, one can easily ensure that the environment necessary to run the user code is present and configured correctly (**Must 1.2**), irrespective of the current state of the physical machine running it. Using a Docker-like container management system, we can also use the Docker repo for base images, which provide most, if not all, functionality required to run particular programming languages out-of-the-box.

5.1.4 Podman Interface

Podman has been chosen as the container management tool for development. It is a Docker-like system that is also fully interoperable with systems that already use Docker in their tech stack. The primary advantage of Podman over Docker is that it is fully operable from its CLI, so unlike Docker, the user does not need to start any separate application in order to run the project (**Must 7**).

Originally, we wanted to use the podman-py package, which would have allowed us to utilize Podman fully programmatically. Unfortunately, from our initial testing, we found that this package is only available on UNIX-based systems, which would have made the application development process significantly more difficult, based on the resources available to the team. Thus, as a solution, we have developed a minimal Podman API through the usage of the Podman CLI and the subprocess Python module.

The Podman class provides standard functionality for creating an image, building an image into a container, executing programs and commands on a container, and removing images and containers. Under normal circumstances, a container would be run with a given program, and once that program finishes, the container itself stops. For our use case, however, we wanted to change what program is running on a container, keeping the container open for further executions. This approach allows for faster subsequent executions of programs, given that they require the same environment. The speed-up

comes from image creation and container building being costly operations, so minimizing the number of times those need to be done is advantageous.

While this approach allows for great extensibility in implementing additional languages, it does come with drawbacks, primarily error handling. Because the interface is made through the use of subprocesses, the output is always a set of strings. This is not inherently a problem for program output, but it makes it more difficult to assess results in case of errors. The only way to mitigate this is to parse error strings, instead of catching the errors programmatically.

5.1.5 Container Class

The container class provides an abstraction for a container entity, alongside all associated Podman API calls. While it would be possible for the system to manage its containers without using a specific class (so by using the API directly), having this class allows for better decoupling. Since there is a concept of a container entity, the higher-level parts of the system need not concern themselves with subprocess management, API calls, and other such issues. The rest of the system simply knows that a container exists and what its interface is. Also, by employing Object-Oriented Programming principles in the system, its testability increases. If a higher-level component had managed its own API calls, testing it would become more cumbersome, and even more importantly, reliant on the fact that the API is working properly. By having the Container class, it can be swapped out for a Mock version, which returns the desired results within a testing environment.

5.1.6 Container Management

In order to comply with the user requirement of having multiple code cells (**Must 1**), some form of container management was necessary. Moreover, the potentially high resource utilization of the Podman system and the extended time needed for container creation were also significant factors. Two main mechanisms have been designed to solve these issues: threading and recycling. The ContainerManager class keeps track of available containers and their respective metadata, spawns the worker threads that are responsible for container operations, and also recycles old containers for later use.

5.1.7 Threading

Since containers exist independently of each other as processes, all the internal operations related to one container, such as creation, execution, and termination, happen separately on different threads (**Must 1.1**). All threads share two resources: the job queue and the list of containers. As the queue system is a built-in thread-safe feature of Python, the only necessary implementation was container verification.

The application has an upper limit on the number of containers, which is set by the user. This is done to avoid instability caused by high resource utilization. When it receives the request to execute multiple pieces of code, the container manager adds the input from each code cell into the job queue, spawning the maximum available threads. Each thread in turn checks if there are suitable containers, either creates a new container or selects an available one, and then executes the code. Besides that check on the shared resource, all subsequent operations happen independently. This has dramatically improved the performance of our application, as container creation was the most extensive operation, due to the need to download the image.

5.1.8 Recycling

During the research phase, we discovered that containers can be kept alive and file communication is possible. These properties have led to the recycling and reutilization feature of the system. When a

new container is created, it is stored in a dictionary, alongside its specific metadata that represents all language specifications. When each thread wants to execute a new job from the queue, it first checks for available and compatible containers. If a suitable container is found, it deletes all the code inside, uploads the new user code, and executes it with the specified input. When no suitable environment is found, the thread goes on to remove the container that has been unused for the longest period of time and then creates a new one. Such a measure ensures that the number of running containers stays below the maximum threshold. If no containers are available to remove, the job is put back in the queue. This recycling of the containers has been designed with the thought that the user would want to run code with the same language specifications in one session, and to avoid the lengthy image creation.

5.1.9 Language Interface

Since the system is supposed to offer support for multiple programming languages, alongside the possibility of integrating new ones, there is a need for a centralized system of implementing additional languages into the project. Multiple parts of the system require information, such as command format, that are specific to the programming language used. If each part were to be implemented where it is used, it would create a system that is difficult to extend in terms of language support. As such, we have created a Language Interface class that implements the concept of a programming language. It contains two major components: a DockerMaker instance and an Injector instance, alongside a few other functions, such as compile and run command formats, as well as helper functions to be used by the container to run code in that specific language (**Should 10**). Both the DockerMaker and Injector classes are similarly defined as interfaces and are subclasses of the Language class. The simplified Language subsystem can be seen in the class diagram in Figure 5, and the whole class diagram can be found in Appendix C.

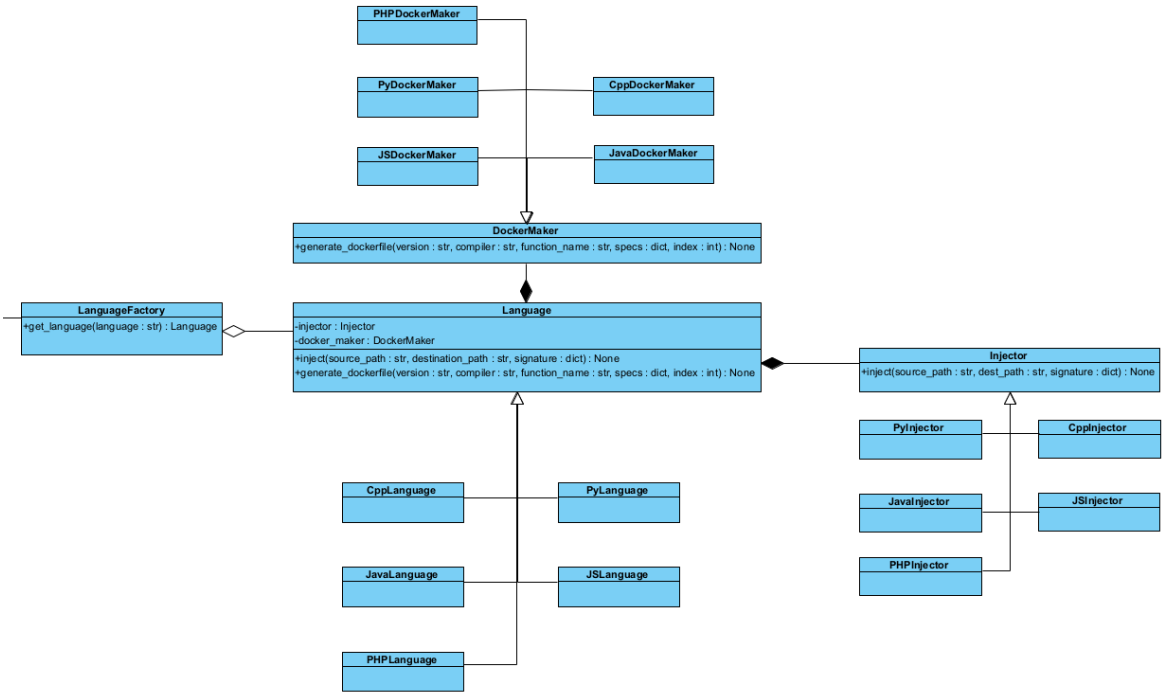

Figure 5: Simplified Class Diagram of Language Component

By collecting all language-specific information into a single file, extending becomes relatively easy, depending on the additional language chosen. To keep the rest of the system as language agnostic as possible, a LanguageFactory class has also been created. The expectation is that throughout the rest of the system, the language to be used is passed as a string object representing the name of the language. This name string is passed to the factory class to retrieve a language instance representing that specific language.

This approach provides several benefits:

- **Functionality abstraction**: The rest of the system need not know how a language is implemented, just that it exists and what its interface is, so it can use it.
- **Dependency collection**: Since many working parts of the system depend on language-specific functionality, having everything centralized makes it easier to change or repair in case of issues.
- **Extendability**: This is an extension to the previous point, as if everything relating to a language is present in a single, dedicated file, it is accessible to add more languages. One only has to create a new class inheriting from the Language class, with its respective DockerMaker and Injector classes. These classes already provide most functionality for how internal processing is supposed to happen, so generally only a small number of functions need overriding to implement a language. Additionally, a new case has to be added to the factory for additional languages, to let the rest of the system know that they are available.

5.1.10 Dockerfile Creation

An image sits at the base of a container. A Docker image is to a container as a class is to an object in an Object-Oriented Programming language; the image essentially is a blueprint for how to build a container entity. In order to create an image, one needs a Dockerfile that specifies how to create an image. The Dockerfile tells Podman what to install into an image; thus, it is also related to the language to be used, as different languages have different requirements in order to run (**Must 1.2, 5**). Luckily, Docker offers a large repository of base images which Podman can use. In our use case, the base image is generally used to provide the compiler or interpreter of the language to be used. Specific base images also come with useful tools, such as the "time" command, which allows for timing and metrics recovery of a process.

Each Language instance holds a DockerMaker instance of its specific language. The DockerMaker needs an associated mounting folder in which files are present that need to be provided in order to run the program. Additional helper files may be added in the language file as necessary. The DockerMaker then takes as input information about the environment to be built and creates an appropriate Dockerfile. In order to keep the containers reusable, the program that is executed by default inside a container is "sleep" for a long time. This allows the container to keep running as long as needed, execute programs, and swap them out appropriately.

5.1.11 Code Injection

The system can take in as an input program a single function. In most programming languages, this is not enough to run a piece of code. Generally, one needs to have a function call at least. In many languages, however, a code file needs to also have a main function or some similar entry point to be able to run. To handle this situation, code injection is done before passing the code file to the container to be compiled and run.

The general assumption of the system is that a program takes in a set of command-line arguments as input and prints the result of calling the function. In order to ensure that this assumption is always met, the Injector takes in a code file as input and adds in the necessary lines of code, outputting a new valid code file. The Injector interface does this step under some general assumptions of how

a piece of code might look for a variety of programming languages. The injection process is done imperatively through a highly granular set of functions that dictate how each substep is done. As such, one can implement additional programming languages by overriding constants and individual functions as necessary to produce valid code (**Should 10.1, 10.2**).

5.2 Functionalities

5.2.1 Writing in a Code Cell

After passing the Home Page, the user will be prompted with two initial code cells (**Must 1**). These code cells can be configured according to the user's liking, and their content can also be edited. To offer full comfort to the user, they can write code in a cell from scratch, directly copy-paste code, or upload a code file that is saved locally on their computer (**Must 3**). In order to offer full code clarity, the code written will have syntax highlighting according to the language the user has selected (**Could 16**).

5.2.2 Code Cell Configuration

In order to run a code cell successfully, the user needs to specify its configuration. By cell configuration, we understand the language, version, compiler, and function signature of that specific cell. We will go over each task individually.

From the drop-down configuration menu specific to each cell, the user can select the language in which the code is inputted. After selecting the language, the menu will showcase the available version and compiler options for each language, ensuring no invalid combinations are selected by the user (**Must 1.2, 5**) (i.e., selecting the Python language and clang compiler). When showcasing the possible options of each language, the default configuration is also highlighted for the user (e.g., when selecting Python, version 3.10 is chosen automatically).

To specify the cell's function signature, the user needs to travel to the Settings page. Function signatures are specified in the following way: there is a "Global" signature container where argument and return types are specified and then an individual container for each cell's specific configurations: library dependencies and function name.

Further, to ease the process of testing, the user has the "Run as is" option for each cell, which means that they need to write their own main and only need to specify dependencies for the cell.

This design is made with the assumption that all code cells, regardless of different configurations, will have the same arguments and return types. However, if users wish to overwirte it, certain workarounds have been identified and will be discussed in Chapter 9.3. With this edge case in mind, it was decided that a global signature would be more efficient and enhance the user experience, as compared to needing to specify each cell signature individually. This choice was also influenced by client feedback, which supported the use of a global signature rather than specifying each one individually.

5.2.3 Settings Configuration

As mentioned in Chapter 4 of this report, all relevant information for customizing result showcasing can be found in the Settings Page. The user should be able to customize the application's output so that they can only view relevant information. Therefore, the Settings Page contains a section where the user can select which metrics are shown after the code is finished running (**Must 2**).

5.2.4 Manual Testing

In order to test their code, the user needs to input their test cases in the Settings Page, giving the user the freedom to test specific edge cases or unusual behavior. When choosing manual testing, after running the code, the "Differential Results" tab will showcase for each test case inputted what the expected output was and which cells did not "pass" along with the value they outputted. To ease the process of testing, the option of automatic testing is also available and will be elaborated on in the next section.

5.2.5 Generating Test Cases Automatically

This feature simplifies and automates the task of creating test cases in order to validate a piece of code, as per requirement **Could 12**. It makes use of the signature the user filled in to generate sets of tests randomly. The generator supports all primitive types and lists or nested lists. The user only needs to select the "generate test cases" option and specify the number of tests to generate. Additionally, this feature may help the user find unusual behaviour in their code. When choosing automatic testing, the user only gets a set of inputs for the program; expected outputs are unavailable.

5.2.6 Adding and Removing Code Cells

As mentioned, the Code Page gives the user two initial code cells (**Must 1**). Of course, in most scenarios, the user should not be restrained to just two cells, so the feature of cell addition and removal has been implemented. The way the application works remains the same no matter the number of cells the user decides to have.

A scenario that has been taken into account during design is the one where the user wants to have a large number of code cells, for example, over 15. The interface that the product offers allows the user to have an unlimited number of cells, but the user experience will be lacking when the number of cells exceeds a certain threshold. In order to serve the users who want to test multiple code cells, such as 50, we have integrated the option of creating and uploading a custom project (**Could 13**). This way, the user can import a project (a JSON file) into the application based on the structure explained in the User Manual, thus eliminating the task of manually adding and filling in 50 different code cells.

5.2.7 Running a Code Cell

This is the core functionality of the system. Users start by entering a piece of code and selecting the programming language (**Must 3**). They can also optionally choose from supported versions or compilers for that language (**Must 5**). To run the code, users must specify the entry point—namely, the function to be called—along with its signature. If the code requires external libraries, these can be added as well. Finally, users can manually define test cases or enable automatic test case generation (**Must 4, Could 12**).

After pressing the "Run" button, all cells will run in parallel (**Must 1.1**), and the application will showcase the final results when all have finished running. If a piece of code takes too long to run, the user can select a Timeout from the Settings Page.

5.2.8 Comparing Test Results

This feature makes it easy to compare the outputs of multiple code cells. The "Differential Results" screen highlights discrepancies, helping users quickly identify which cell failed and which test case caused the failure. If a test is "matched", then it means that all cells had the same output/behaviour (**Should 9**). Further, the user can also clearly view each cell's performance metrics and output by accessing an overlay specific to each code cell (**Must 2.1, 2.2**).

5.2.9 Saving and Loading a Project

This feature allows the user to work on the same project in different sessions. The project is saved in JSON format, which can also be easily modified in an editor if the user wants to create a custom setup. The user can then reload the project in the app at any time (**Could 13, 14.1**).

As mentioned prior, this feature allows for more flexibility in using the app and facilitates cases where the user wants to work with a large number of cells or speed up their testing process.

5.2.10 Uploading Code and Input/Output Files

This feature allows the user to simply upload their files in the project instead of manually writing or copy-paste them in the correct place (**Must 3.2**), thus speeding up the testing process. For the input/output test file, a certain format described in the User Manual is expected to correctly introduce the data (**Must 4**). For code files, only files with correct extensions are accepted (e.g., *py, js, cpp*).

5.2.11 Extending Features

One requirement of our system is that it be scalable and extendable. In doing so, we offer the trade-off that the user does not have a specific "Add Plugin" button but can extend the application by following the instructions given in the Developer Manual. Therefore, the user can add other metrics and languages to the application according to their needs (**Should 10, Could 15**).

6 Testing

The scope of this chapter covers the testing of the system to ensure that both the system and user requirements are fulfilled, as previously agreed with the clients. To test the app's functionality on the Backend, we will use Automated testing, namely Unit testing and Integration testing. We will run Usability tests to ensure that the interaction with the users is proper and that there is a smooth flow of actions. There is no need for Security tests since the client was not concerned with this aspect, and it was not a requirement to provide security to the application.

Throughout the Development and Testing phase, we will use Manual testing to check the high-level functionality, but we never rely solely on this practice to decide whether a piece of code is working as expected.

6.1 Test Methodology

6.1.1 Unit testing

During Unit testing, we will ensure that all the small code units work accordingly and look for the correct functionality. The blocks of code will be tested in isolation, and when a connection with an external component is needed, that will be mocked through a stub implementation or the Mock library from the unittest framework provided in Python. Here is where the basic operation and logic of the code will be tested using the unittest framework, checking that the functionality is kept under multiple different conditions, including edge cases.

6.1.2 Integration testing

During Integration testing, we will check whether all the system components work well together and whether the communication between them is properly handled. We will use Postman to test the communication between the Frontend and Backend. We will also check that the setup is correctly executed, if the requests are valid, and if errors are properly handled.

Four main components need to be tested in the container creation and execution pipeline. We tested the communication between two components in a bottom-up approach, following the natural order of the pipeline. First, the Podman interface was tested to ensure proper communication with the Podman application and correct error handling. Afterwards, the Container-Podman interface was tested to ensure proper container creation and subsequent commands. Finally, we tested the Container Manager-Container communication to verify proper container creation and the efficient reutilization of containers.

6.1.3 Usability Testing

To ensure that the users fully understand the interface and have an intuitive flow of actions, a set of users will have to attempt to complete tasks. The test scenarios can be found in Appendix B. The team will observe and evaluate the users' behaviour, and we will also ask for feedback. The usability of the interface will be measured by 3 usability testing metrics, which were mentioned in UXtweak [6]:

1. **Effectiveness**: It describes whether a user can complete a particular task and how effectively they can do it. Effectiveness is measured by *Success Rate*. It is represented by the number of users who succeeded in completing a task out of all the users.
2. **Efficiency**: It describes how easy it is for a user to reach the endpoint of a task and how much

effort is needed for that. To measure efficiency, *Time on task* will be used, tracking how fast a user performs a certain task.

3. **Satisfaction**: It describes how satisfied a user is with the product's experience. To measure satisfaction, the users who participate in the testing will receive a Likert scale, in which each number represents a level of satisfaction with the interface. The Likert scale can be seen in the Appendix G.

6.2 Risk Management

This section will present and assess the risks of the entire testing process, along with mitigations for each risk. The risks were assessed before the entire Testing phase started, to ensure that everything would work properly and that there would be no oversights.

6.2.1 Risks

- **Requirements change frequently**: Since the team took an Agile approach in the development of the project, requirements may change during this phase, which could lead to incomplete test coverage and not enough time for testers to test the app.
- **Ambiguous requirements**: The requirements could be unclear, leading to gaps in understanding, functionality, and, therefore, in the tests.
- **Edge cases are not discovered**: Since edge cases are very specific to each project, they cannot be entirely discovered during the Planning and Testing phase. This might also be a consequence of inexperienced developers unfamiliar with software testing.
- **Incomplete Test Coverage**: This is a consequence of not testing all the functionalities and not discovering all the edge cases. It could lead to errors in the code that are not that obvious, bugs not detected, and gaps in understanding the system's behavior.
- **Time Constraint**: Since the Testing phase for the entire project is only five weeks, it might not be enough to fully test everything and check that all the components are successfully integrated. This might lead to incomplete testing, gaps in functionality, and errors.
- **Changes in the code base**: The code base could be changed frequently. Even minor changes can influence the test's ability to measure the expected functionality of the code.
- **Integration issues**: If the dependencies are complex and the components rely a lot on each other to offer the full functionality of the system, superficial integration testing could lead to the failure of the entire system. Since one component having issues can bring down the entire system, this is a critical concern. Furthermore, since we are using Podman and relying on a third-party source, there might be the problem of not being able to perform Integration testing thoroughly.

6.2.2 Mitigations

Nr.	Risk	Mitigation
1	Requirements change frequently	1. Communicate with the stakeholders from the beginning, ensuring that the developers understand entirely what the stakeholders want. 2. Start the project by asking the stakeholders many questions so the team can gather many details from them.
2	Ambiguous requirements	1. Regular meetings with the stakeholders and continuously ask them if the requirements and progress meet their expectations.
3	Edge cases are not discovered	1. To discover as much as possible, include every developer, not only the testers, in thinking about edge cases. 2. Ask the stakeholders/users to use the product even if it is not ready, to give the possibility of discovering a new edge case. 3. Analyse other similar projects and check if they have any undiscovered edge cases that could apply to this app as well.
4	Incomplete Test Coverage	1. Update the test cases often to ensure that everything is tested. 2. Use tools to analyse the test coverage and discover what has not been tested enough. 3. Make a comprehensive test methodology.
5	Time Constraint	1. Focus on the most important features and test them first. 2. Start testing early, when the development starts. 3. Allocate enough testers so the entire testing process can be done in the time given.

Nr.	Risk	Mitigation
6	Changes in the code base	1. Encourage a Test-Driven Development Approach. 2. Develop the tests vague enough that if there are small changes in the code, they do not affect all the tests, but thorough enough that all the functionality is still tested. 3. Make the system architecture so that not many components rely on one another.
7	Integration issues	1. Run integration tests after each feature is developed.

6.3 Timeline

In this project, the Testing phase started one week later than the Development phase. (Week 5) This was because in Week 4, we decided from the beginning that we might still be working on the Design of the system and also starting the Development phase, so testing started in Week 5 to ensure that the basic functionalities needed were met. From then on, every time a functionality was added, unit and integration tests were made to ensure that the functionality was properly developed. Usability tests were performed at the end (Week 8), after the entire product was a unified system, to make sure that if we need to change anything further, we can do so in the remaining time.

6.4 Results

6.4.1 Unit Testing

In total, 235 tests were written. In each class, external dependencies on other classes were mocked to test only the functionality and flow of that specific class. For each method, all possible branches were tested, including the raising of errors and exceptions.

Almost all of the classes were tested during this phase, except a couple of them, such as: CellSim, LanguageFactory, LoggerConfig, Session, and SessionManager. CellSim is not tested because it is just a simulation of a cell in order to be able to send the results from Manual testing to the Frontend, so the Differential tab can be loaded. LanguageFactory is just identifying which Language class is needed for the programming language used. All of the extended Language classes (PyLanguage, JavaLanguage, etc.) are tested, so the Factory class doesn't need checking for functionality. LoggerConfig is primarily used as a debug tool during the Development phase, and since it doesn't add to the functionality of the end system, it is not tested. Session and SessionManager are classes that only have a constructor and some empty methods, since they are meant to be used in a further Development phase of the program, so the application can have multiple sessions run at the same time. This functionality is not yet implemented, and it does not need testing.

All of the other classes that add to the logic functionality of the Backend are tested through unit tests.

During Unit testing, some gaps in the Backend logic were eventually solved. One of them is that in the ContainerManager, the way we removed containers was not correctly implemented, but it was later solved.

6.4.2 Integration Testing

A total of 43 tests have been written, testing each method that uses at least two components from the execution pipeline. The tests aim to cover both normal execution, but also error handling and correct clean-up. As a result of the integration tests, the following bugs/oversights have been discovered:

1. **File clean-up in container**: In the container reutilization procedure, before new files are copied, the old ones are deleted. The subprocess that ran the delete command did not properly delete the files inside, but this did not create any errors, because if a file with the same name would be inserted, it would overwrite the old one. If the file had a new name, the old ones remained unused in the container, without causing any errors.
2. **Verification of the maximum number of containers**: When the system compared the number of active containers with the maximum number of containers allowed, the comparison operation used was equality, not larger-or-greater. This was an oversight, but it did not produce any errors, as the comparison happens with a thread lock, ensuring that no concurrency issues should arise from increasing the container count before another thread checks the container count.

Fortunately, neither of these oversights caused errors, but they have both been resolved to ensure code quality and mitigate unintended behaviour.

6.4.3 Usability Testing

In total, 6 technical users completed the usability testing, from which 4 were new time users, and 2 of them were the clients, who did know how the app works because we showed it to them multiple times during the sprints. For the first-time users, we gave some guidance, such as what tasks they should do and, if necessary, where some buttons could be found (i.e. the Configure Cell drop-down). Even if we guided them, they were strongly encouraged to check the User Manual that provided all the information, but sometimes, because of the time limitation, we provided them with instructions.

1. **Metrics**
   (a) *Effectiveness* - Success rate
   It is measured by taking the total number of tasks that were successfully done out of the total number of tasks performed by all the users. Out of 204 tasks performed by all the users who participated, 199 tasks were successfully completed, which resulted in 97% of participants who managed to finish a task. In UXtweak (n.d.) [6], it is mentioned that an average result is 78%, so the desired outcome was achieved.
   (b) *Efficiency* - Time on task
   Since there is no specific recommendation of how much time each task should take because each system is different and the time depends on the complexity of it, we measured the times for each task that was performed for each user. Most of them were completed in less than 10/15 seconds, which is also due to the fact that the tasks were granular (the list of tasks can be found in Appendix B). The tasks that took the longest were: "Write code in cells", "Configure cells" and "Fill in signatures". This is due to the fact that even though the interface is designed to look like an IDE or code editor, usually in those interfaces, you

don't need to select the input and output types and specify the name of the functions in order to run the code. This was expected for first-time users, but we still consider that the app is efficient because after doing these tasks once, when the users needed to repeat them again, they did it faster.

(c) *Satisfaction* - Likert Scale The feedback given by the clients regarding the Likert scale can be found in Appendix G, along with the other results. On average, the level of satisfaction expressed by the users is 4, which on the scale presented, is the equivalent of "Satisfied".

2. **Success Criteria**

In some cases, there were task failures, which happened because there were functionality gaps in the Backend. This does not mean that the interface is not working accordingly, because each time that happened, the errors could have been seen in the terminal of each cell, and the user knew it was not the cause of an incorrect input or so, but a shortcoming on the system side. The task was then eventually completed after trying a second time.

The issues encountered were for the following tasks:

(a) *Upload a code file*

This was because a part of the code that was written in the code file provided had an issue. After the code was repaired, it worked.

(b) *Run code cells*

After trying to run two cells, in which one was Python code, it gave an error that the Python image could not be pulled from the repository. Even though we couldn't pinpoint the source of the error, it is likely to be due to an internet connection issue that in a few seconds was solved, and the system worked again.

(c) *Input timeout period / Write code in cells*

The user was curious to see what would happen if a function that was not supposed to terminate was inserted in a code cell. The expected behaviour was that the timeout functionality (timeout was set on 60 seconds) would interrupt the program and the cells would output a timeout error. This did not happen, and the system froze for a bit. After reopening the app, the issue was fixed. After noticing this issue, the problem was solved in the Backend, but not implemented in the final version of the system to make sure that it would not lead to other unexpected outcomes. The partial solution for this issue is on the "experimental features" branch in the GitHub repository.

3. **Feedback**

The overall feedback from the users was a positive one. They were mostly satisfied with the design and functionality of the interface.

Some of the features that the users appreciated the most when using the app were:

(a) Most of the steps are intuitive.

(b) The icons are clear and intuitive.

(c) It is very helpful that you can see how the overlay will look based on what you choose to display in the "Settings" tab.

(d) The functionality of saving and importing the projects as JSON files is very useful.

(e) Designwise, the interface looks good and using it for a long time would not be tiring.

(f) The functionality of being able to press Ctr + F in the Code Cell to search for a certain word/character.

Users said that the workflow was intuitive enough, but for all of them, the steps of Configuring a cell and Fill in the signatures were slightly confusing. This was again due to the fact that in a normal IDE/ code editor, the user doesn't need those additional steps to run code. In order to overcome this, we asked the users what they would see as a solution.

Some of the other improvements that the user suggested are:

(a) The logo icon located on the top left could be used as a "Go back" button since they expected that it would take them from the Settings page to the code page.

(b) The Configure Cell tab could have a "Save" button since it is confusing if the configurations are saved or not without it.

(c) Change the icon for the Differential tab since it is not very suggestive.

(d) Add an alert that would specify when the tests have finished running.

(e) Add the input for each test in the overlay with the results of each cell.

(f) Implement key shortcuts (i.e. Ctrl + S).

(g) Add tool tips (i.e. when hovering on a button, tell the users in text what that button represents or what the next step would be in order to get to running the code).

(h) Add the possibility of changing the font size and theme of the interface.

Since this information can already be found in the user manual that can be accessed at any point when interacting with the app, adding these features is a future improvement, and more information about them can be found in Chapter 9.3.

7 System Limitations

Throughout the Development phase of the project, there are some system limitations that have been identified. This list is based on developer observations and testing results, but may not be comprehensive.

7.1 Pathing

The application must be run from a location that has a path that has no whitespaces. If using the installer, this should be done automatically, but may not depending on the operating system and disk distribution. This also applies when running the Backend individually or building the system from source.

7.2 Internet Access

Internet access is required to download base images for Podman containers. Downloading happens when the system tries to use an image not already stored locally by Podman, so it is necessary at least the first time the code is executed from the application. Locally stored base images can still be accessed without internet access. It is also possible for the Docker repository (from which base images are downloaded) to be down. This seems unlikely, but it is nevertheless possible.

7.3 Windows

During our test trials, we found that on certain Windows machines, problems can occur with spawning a subprocess, which is necessary for executing Podman queries. This seems to be caused by different shell configurations on the executing machine. One potential fix to this issue can be to manually set the SHELL constant in sources/Podman.py to a different value and then rebuild the system from the source.

7.4 Mac

Minimal testing has been done on Mac, due to limited access to a reasonably powerful Mac machine. The Frontend of the application runs properly on Mac machines (including Electron packaging), but the Backend may have issues. With the current system design and implementation, the entire system does theoretically run. However, Podman had issues running on our particular Mac machine, thus the rest of the system could not be properly tested on this operating system.

7.4.1 Podman

Podman is a central dependency to the Backend of the system. During our testing trials, we had issues getting Podman to work properly on our Mac machine. Installing Podman through Brew seems to lead to a faulty installation. Installing Podman through the installer provided on its home website also seems to have issues, but that could be potentially solvable by downgrading to an older stable version. There seem to be some virtualization issues on certain combinations of processors and operating systems.

7.5 Linux

The system should run on a Linux machine; however, we did not have access to one to test, so we cannot guarantee functionality for this OS.

7.6 Java Language Support

The particular syntax of Java and how projects are created and executed made it hard for some features to be implemented for the Java Language. With the current implementation, the List class is the only allowed data type for array-like structures. The method that should be tested should also be static. Another limitation was the choice of compilers, as OpenJDK is now deprecated on the Docker Repository, so only Amazoncorretto and Temurin are available.

7.6.1 Additional Dependencies

Java does not work with normal package managers, and it requires dependency specifications, building, and packaging. This makes it quite hard to implement dynamic project building with dependencies. Further development is needed to achieve support for dependencies.

7.6.2 List of Strings Input

Because the package that allowed the parsing of input strings that represent a list of strings had to be added dynamically, we also parsed the input as a normal string, splitting at the ‘ , ‘ character. This does not allow for the string in the list to contain that character, as it will split the string into two different ones.

7.7 C++ Language Support

C++ is a large, high-level language with many multi-paradigm features. As such, the system only supports a small subset of C++ language features.

7.7.1 Raw Array Types

Due to the lower level memory management quality of raw pointer types in C++, these are not natively supported by the system. Function interfaces must use non-array, non-pointer primitive data types. Beyond that, std::string and std::vector are supported.

7.7.2 Additional Dependencies

Additional libraries must be available on the Alpine Package Keeper. There is currently no way to add dependencies beyond what is available on the APKmanager. When adding these dependencies on the specs list of a code cell, the name must match that of the package on APK.

7.8 Language Extension

The user might want to extend the system with additional programming languages. It is possible that certain programming languages have features which cannot be supported based on the system architecture. The system design assumes a programming language has a specific structure, which to our knowledge should be fairly general, but it is nevertheless possible that some languages have vastly different structures and constructs.

8 Conclusions

8.1 Discussion about Execution

During the project's research phase, we came across multiple ways of implementing the functionality of running code. One of the ideas that we wanted to implement was using LLVM IR and injecting code at that level. We did not fully experiment with the idea, but we were certain that it would work, as we tried with lower-level languages like C. After deciding and presenting the idea, getting feedback on it, we started to experiment with how higher-level languages are compiled into IR. It turned out to be more complicated than we first thought, and we decided that implementing this would be outside the project's scope. A better approach would have been to test LLVM IR code and code injections for multiple languages before proposing the idea and presenting it.

One recurring theme through the development of the system design has been extensibility. We wanted to ensure, to the best of our abilities, that the system is sufficiently modular and can be extended with features as the user sees fit.

8.2 Final Requirements Implementation

In this section, we go over the System Requirements (Chapter 3.4) once again, highlighting what has and has not been achieved throughout the duration of the project. These are based on associated User Requirements and Use Cases (Chapter 3.3), as signed off by the client.

Must

1. The system interface must have 2 code cells.
   - 1.1 The two code cells can be run at the same time. **- Done**
   - 1.2 Each code cell should have an individual configuration chosen by the user (e.g., language, version, external libraries). **- Done**
   - 1.3 Each code cell should run on the specified test cases. **- Done**
2. There must be a configurable dashboard based on multiple tiles. **- Done**
   - 2.1 The user should be able to choose the metrics and outputs showcased in the code comparison. **- Done**
   - 2.2 The system should showcase at least the following: performance metrics (e.g., execution time, memory usage) and test suite results. **- Done**
3. The user must be able to upload code to the dashboard's code cells. **- Done**
   - 3.1 The user should be able to change the input program within the code cell. **- Done**
   - 3.2 The user should be able to upload a code file. **- Done**
   - 3.3 The user should be able to insert code in the code cell. **- Done**
4. The user must be able to manually upload an input and output file for specific test cases. **- Done**
5. The system must be able to choose a compiler according to the programming language chosen by the user. **- Done**
6. The system must be user-friendly and intuitive. **- Done**
   - 6.1 The user should be able to learn how to use the system within 5 minutes.

6.2 The buttons should be easily recognizable by their icons.

6.3 The user should easily distinguish from which code cell the results are coming from.

7. The system must be a standalone app. **- Done**
8. The user must have a clear and concise set of instructions for using the app. **- Done**

All requirements under the Must section have been completed.

Should

9. The system should be able to compare the outputs of the program with the expected outputs from the test cases. **- Done**
10. The system should be easily scalable and extensible. **- Done**

    10.1 The code base of the system is well-encapsulated and uses interfaces.

    10.2 The system architecture allows for the addition of new features.

11. The system interface must be visually accessible. **- Not Done**

    11.1 The user has the option of changing the font size.

    11.2 The user has the option of choosing between dark mode and light mode.

Accessibility features under Should have not been implemented due to time constraints and relatively low feature value. These should have realistically been classified lower on the prioritization scale.

Could

12. The system could be able to generate test cases based on the given code and function signature. **- Done**
13. The user could be able to save their projects locally. **- Done**
14. The user could be able to close the app and not lose their local project. **- Not Done**

    14.1 There should be a warning when closing the app without saving.

15. The user could be able to write and add plugins to the application. **- Reformed**
16. The text within the code cells could have syntax highlighting. **- Done**

There is an alert for saving the current working project when switching to the HomePage, but no alert will be given when directly closing the application.

Plugins have not been explicitly integrated into the system. There is currently no way of adding a piece of user code (e.g., Python script, *.dll* file) into the system and have it run as is. It is possible, however, to add new functionalities to the system and rebuild from sources with relative ease.

Won't

17. The application won't be integrated as an extension to an already existing IDE. **- Not Done**

The application is not integrated as an extension for an IDE, but integration could be possible. Further details on this topic are discussed in the next chapter of this report.

Furthermore, based on the meetings with our clients, all use cases have been approved.

9 Future Improvements

9.1 Implementing the "Won't" Requirement

During the Design phase, one requirement was deliberately placed in the "Won't" category: developing the project as an extension for an IDE, such as Visual Studio Code. While this feature was not implemented within the scope of the current project, the system has been designed to make such an extension technically feasible in the future. With appropriate integration layers and API exposure, the current architecture could support this direction with manageable effort.

9.2 Extensibility and Scalability

A core design goal of the system was to ensure it remains extensible and scalable. Supporting additional programming languages has been made straightforward—adding a new language typically requires implementing a single configuration or execution file. Clear documentation and existing examples are available to guide developers through this process.

On the scalability front, although the current system runs locally, it is built with the potential to migrate to a remote server setup. This transition would require only minimal changes, as partial support for remote execution is already in place. This opens the door to deploying the system in distributed or cloud-based environments to support more intensive workloads or multiple users.

9.3 Frontend Improvements

While the Frontend fulfills its core purpose, several opportunities exist to enhance the user experience. A key improvement would be to give users greater control over the app's appearance. Potential features include a light mode, the ability to customize fonts and font sizes, and other accessibility options. These additions would make the app more user-friendly and adaptable to individual preferences.

An additional improvement would be changing how the status bar is shown to the user. Currently, each cell shows its own status bar, even though there is one progress bar for all code cells. Some users have reported confusion caused by this because they would believe that each code cell has its own separate status bar. An improvement would be to have only one status bar over the whole screen. Other improvements would be to allow the user to dynamically adjust the width of code cells in order to make the code less/more visible, but also to add tooltips to ensure there is no confusion on what each button represents.

As mentioned in Chapter 5.2.2, code cell signature is declared globally. Therefore, it is currently not possible for the user to declare individual cell signatures. In the case the user wants to do this, given the current system, the user would be able to do so by uploading a custom configuration file. From the development side, this would only require changes at the level of communication between Frontend and Backend.

Regarding the buttons, a nice addition to the interface would be a "Save" button in the Configure Cell drop-down of each code cell. The users expressed their confusion when having to select the configurations because they had no way of being certain that what they selected was actually saved, and adding a button would solve their problem. Users were also confused that there was no button for going back to the previous page and that the logo from the Code Page does not have that functionality, since it looks like the "Go Back" arrow. An improvement would be to make the logo have that functionality. Another thing that was mentioned by the users was that the "Differential" tab icon is not very suggestive and can be changed into something else. This could be an icon that would suggest comparing results more.

Regarding functionality, users suggested having the input for each test in the overlay with the results of each code cell. As it is implemented now, the inputs are shown in the "Differential" tab only if some tests did not match, but it was suggested that having inputs for each code cell for each test case would make the process of understanding which test failed and how the code behaved more efficient. Another improvement that would make the workflow more efficient would be to add an alert that says when the tests are done. Right now, the progress bar shows how much the running process has advanced, but the users cannot differentiate between the case in which the tests actually finished running and the one in which an error was encountered. Adding such an alert would solve this issue. The final improvement suggested by the users to achieve a smooth workflow was to implement key shortcuts (i.e. Ctrl + S) that are available in an IDE/Code Editor since they are used in every app, and everyone is used to them.

9.4 Copy Cell Functionality

Having a copy cell functionality would make it much easier for a user to add a large number of similar code cells quickly. This functionality would allow a user to copy a code cell alongside its contents and configurations.

9.5 Configuring Automated Tests

As it is currently implemented, the user doesn't have control over the generation of test cases, except for the number of test cases. In the Backend, there are intervals for generating the values; for example, numbers can take values from -200 to 200. This can cause problems, especially if the user code doesn't expect negative numbers, for example. As a possible improvement, the user should be able to configure all intervals to match the requirements for testing.

References

[1] Agile Business Consortium. *MoSCoW Prioritisation*. https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html. Accessed: 2025-04-11. n.d.

[2] Y. Dang et al. "Optimizing Program Repair via Learned Heuristics". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21)*. Association for Computing Machinery, 2021, pp. 1327–1339. DOI: 10.1145/3468264.3473124. URL: https://doi.org/10.1145/3468264.3473124.

[3] IBM Corporation. *Get It Right the First Time: Writing Better Requirements*. https://www.ibm.com/docs/en/SSYQBZ_9.6.1/com.ibm.doors.requirements.doc/topics/get_it_right_the_first_time.pdf. Accessed: 2025-04-11. n.d.

[4] S. Li et al. "CompDiff: Data-driven Comparative Debugging for Program Transformations". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. Association for Computing Machinery, 2023, pp. 830–845. URL: https://shao-hua-li.github.io/assets/pdf/2023_asplos_compdiff.pdf.

[5] Scrum.org and ScrumInc. *The Scrum Guide*. https://scrumguides.org/scrum-guide.html. Accessed: 2025-04-12. 2020.

[6] UXtweak. *Usability Testing Metrics*. https://www.uxtweak.com/usability-testing/metrics/. Retrieved [April 11, 2025]. n.d.

A Appendix - Use Case descriptions

No.	Use Case	Description
1	Open app	The user opens the app by clicking on the shortcut on the desktop. After opening the app, the user is prompted to choose between opening an existing project or creating a new one.
2	Open existing project	The user can open an already existing project by selecting a previously saved app project.
3	Create new project	The user can create a new project with no configurations or code.
4	Save project	The user is able to save the current project locally in a location of their choosing by selecting the "Save Project" option.
5	Configure project	The user can specify necessary information in order to run the piece of code they wish to test. This information includes things such as programming language, language version, necessary dependencies, and function signature.
6	Upload configuration	When creating a new project, the user is able to upload a configuration file that sets the parameters of the program: language, version, compiler, function signatures.
7	Upload file	a. The user can upload a code file (e.g., a ".py" file). b. The user can upload a file with test suites containing desired inputs and outputs. c. The user can upload a file containing environment requirements (e.g., a file specifying each code cell's specifications).
8	Write code in cell	a. When in a project, the user is able to write new code in a code cell or modify existing code. b. The code written by the user has syntax highlighting.
9	Generate test suite	The user can ask the system to generate a suite of test cases based on the provided function signature with which to run both code cells.
10	Run code cells	The user can press a button and execute the code present in the code cells.
11	Choose output type	a. The user is able to choose to see the output of each cell after selecting "Terminal". b. The user is able to choose to see the performance metrics of each cell after selecting "Metrics". c. The user is able to choose to see all the results from all the code cells in a plot after selecting "Plots".
12	Analyze results	a. The user is able to compare outputs from the tests for the provided code snippets by seeing which tests matched, which did not, their inputs, and respective outputs. b. The user is able to compare the performance metrics of each cell. c. The user is able to compare metrics and outputs through visual plots and charts.

No.	Use Case	Description
13	Press “Help” button	The user has the ability to view a detailed manual containing all the instructions needed to utilize the program.
14	Press settings	The user can access and modify different system settings through the “Settings” interface.
15	Select ”Interface Settings”	The user can configure the visual aspect of the system through the “Interface Settings” element.
16	Change interface aspect	The user is able to configure the visual aspect of the system through settings such as font size and color scheme.
17	Save changes	The user can save the settings that they have changed.
18	Select location in computer	The user can choose where on their machine the project is saved.
19	Extend feature	a. The user is informed through the Developer Manual how to successfully add a new language by extending the classes Language, DockerMaker, and Injector and informing the user how the systems manage a new language. b. The user is informed through the Developer Manual how to successfully add extra metrics by adding it in the ResultParser class and finding a way to measure it.

Table A.1: List of Use Case Descriptions

B Appendix - Test Scenarios

B.1 Navigability and Learnability

Objective: Verify that new users can learn how to navigate the system within approx. 5-7 minutes.

Tasks:

1. Ask the user to open the application.
2. Create a new project.
3. Add code to the 2 cells.
4. Configure the cells.
5. Fill in signatures and function names.
6. Select the type of testing.
7. Run the code.
8. Analyse the results from each cell and the diff tab.
9. Save the project.
10. Close the app.
11. Reopen the project that was just saved.

Success Criteria:

1. The errors, if encountered, are clear to the user.
2. The system alerts the user whenever something happens: a file is saved or uploaded, or changes are saved.
3. The system provides visual support through a status bar that informs the user how much progress the system made with running the code cells.
4. The system can be navigated for a first time user within 5-7 minutes, without including the time of running the code.

Related requirements:

1. User requirements: 7.
2. System requirements: 6.1, 6.2, 7.

B.2 Running Multiple Cells

Objective: Check if the user can get to run multiple code cells and understands the core functions of the app.

Tasks:

1. The user writes code in the cells.
2. The user is asked to configure the cells.
3. The user is asked to find the Signature tab from the Settings page and input the Function Signatures and the testing values for Manual Testing.
4. The user is asked to run the code cells.
5. The user is asked to analyse the default option of "Results" only with the outputs of each cell.
6. If the user runs only 2 cells, they are asked to add another cell and repeat the steps from 1 to 5.
7. Ask the user to add a new cell, in which they would upload a code file.
8. Ask the user to repeat the steps from 1 to 5.
9. Ask the user to delete a cell.

Success Criteria:

1. All the buttons related to running a code cell, configuration of a code cell, and analysing the results are easily recognisable.
2. The user can get to the endpoint of the tasks, which is analysing the results for more than 2 cells, in approximately 5 minutes (not counting the time of the code running).
3. The user understands the flow of actions needed for running code cells and is not confused throughout the process.
4. The user is able to add and delete a cell.
5. The user can upload a code file.
6. The File Manager of their computer opens automatically when selecting "Upload Code".

Related requirements:

1. User requirements: 1, 2, 4, 5, 7, 14.
2. System requirements: 1.1, 1.2, 3.1, 3.2, 3.3, 5, 6.2, 6.3, 16.

B.3 Explore the "Settings" Page

Objective: Check if the user can select different types of output and settings of the terminal and can easily navigate through them.

Tasks:

1. The user is asked to try to select the options that would allow them to see the execution time and memory usage of each cell.
2. The user is asked to give a timeout period for the program.
3. The user is asked to run the code cells.

Success Criteria:

1. All the buttons related to metric, terminal, and system settings are easily recognisable.
2. All the fields and buttons related to setting the signature of the program are easily recognisable and easy to navigate to.
3. The button for the Settings page is easily recognisable.

Related requirements:

1. User requirements: 7.
2. System requirements: 2.1, 2.2, 6.2.

B.4 Generation of Tests and Comparison of Test Results

Objective: Check if the generation of test cases flow is intuitive and that the visualization of results is clear.

Tasks:

1. Ask the user to upload a code file (`ex_list.py` in `examples/sample_functions` folder of the project).
2. Ask the user to navigate to the Signature tab, where the Testing Values should be input.
3. Ask the user to:
   (a) Generate Test Cases Automatically
   (b) Perform Manual Testing and provide input and output values
   (c) Perform Manual Testing, where one output is intentionally wrong.
4. Upload a Test file which is previously provided by the testers (`test_lists.txt` in `examples/sample_input_output_fi` folder).
5. Ask the user to run the code cells and analyze the results of the tests from the Diff tab.

Success Criteria:

1. The user is able to go through all of the 3 scenarios of tasks and perform them within 5 minutes.
2. The system generates the test cases correctly.
3. If the outputs differ, the system alerts the user about that.

Related requirements:

1. User requirements: 6, 10, 11.
2. System requirements: 1.3, 4, 9, 12.

B.5 Saving and Importing Projects

Objective: Check if it is intuitive and easy for the user to save and reopen a project.

Tasks:

1. Ask the user to save the newly created project.
2. Ask the user to reopen the project from the "Home" page.
3. Ask the user to make some changes to the project and save it.
4. Ask the user to close the app and then reopen it and open their modified project.

Success Criteria:

1. The user can easily recognise the buttons related to saving a project and opening an already existing one.
2. The file containing an already existing project is successfully uploaded and opened on the system.
3. The system notifies the user when they want to close the app without saving the project.

Related requirements:

1. User requirements: 12.
2. System requirements: 13, 14.1.

B.6 Help Accessibility

Objective: Check that the user has access to the user manual and that it is understandable.

Tasks:

1. Ask the user to access the User Manual.
2. Ask the user to check the indications for a specific task in case that was unclear before.

Success Criteria:

1. The User Manual is easily accessible and easy to read for the user.
2. The button that leads to the user manual is intuitive and clearly recognizable.

Related Requirements:

1. User requirements: 8.
2. System requirements: 8.

C Appendix - Complete Class Diagrams

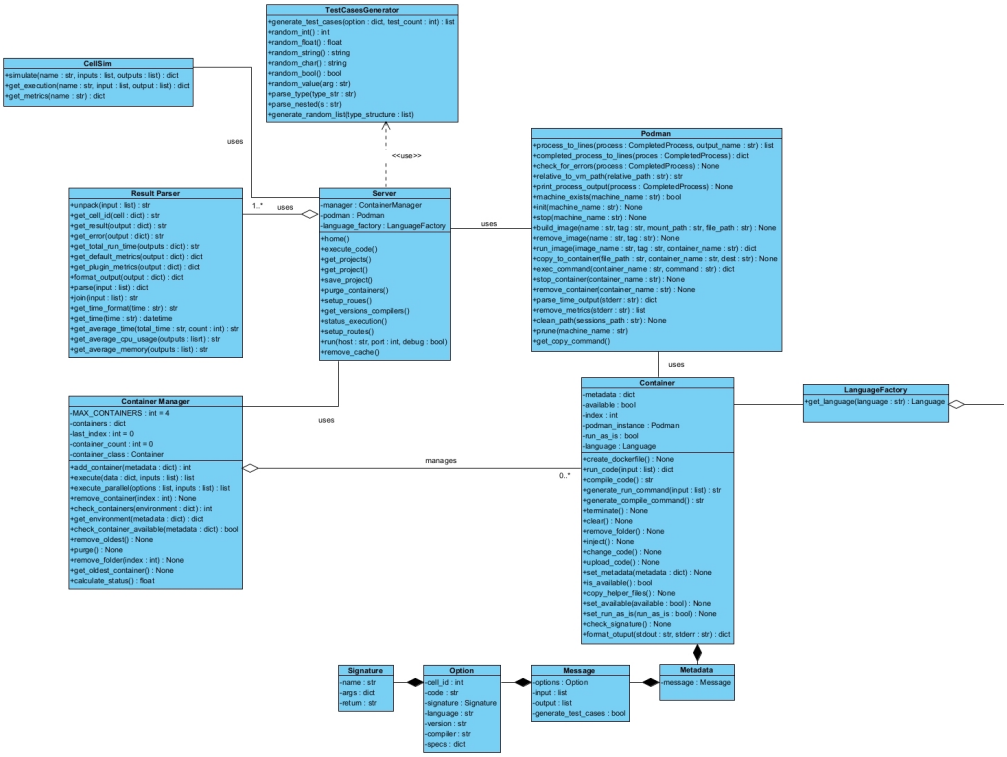

Figure C.1: Full Class Diagram For Server Component

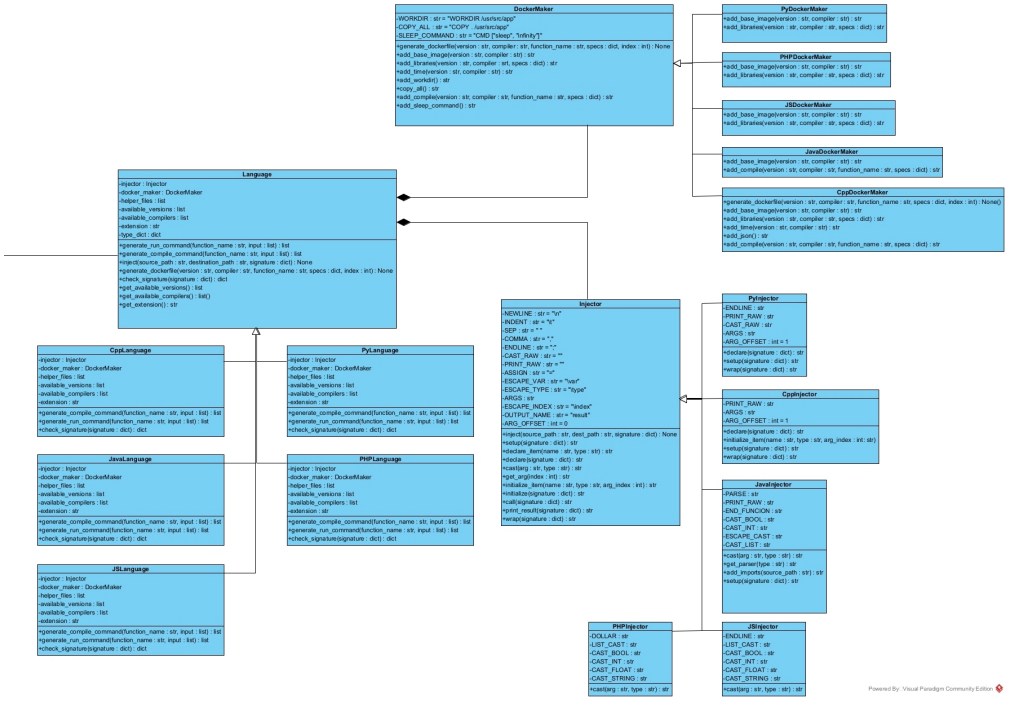

Figure C.2: Full Class Diagram of Language Component

D Appendix - Summary of Usability Tests Results

Name	**Task**	**Time**	**Done (Y/N) / Errors**
Navigability and Learnability	Open app	1.83s	6Y
	Create new project	1.83s	6Y
	Configure cells	11.2s	6Y
	Fill in signatures	25.05s	6Y
	Select type of testing	5.6s	6Y
	Run code	3.2s	6Y
	Analyse results	6.2s	5Y - 1N
	Save project	3.5s	6Y
	Close app	1.8s	6Y
	Reopen project	2.2s	6Y
Run multiple cells	Write code in cells	35s	6Y
	Configure cells	6.5s	6Y
	Complete the signature tab	19.5s	6Y
	Run code cells	3.2s	6Y
	Add another cell	2.2s	6Y
	Upload a file in the new cell	8s	6Y
	Delete cell	2.3s	6Y
"Settings" page	Select execution time and memory usage	5.2s	6Y
	Input timeout period	2s	6Y
	Run code cells	4.2s	6Y
Generation of tests	Upload a code file	8.7s	6Y
	Go to the signature tab	5.7s	6Y
	Generate Test Cases	6.7s	6Y
	Manual Testing right	26.2s	6Y
	Manual Testing wrong	28.7s	6Y
	Run the code cells and go to the Diff tab	7.7s	6Y
Saving and importing projects	Save the project	4.7s	6Y
	Reopen project from the "Home" page	7.7s	6Y
	Make changes and save it	15.7s	6Y
	Close app and reopen project	4.3s	6Y
Help Accessibility	Access the User Manual	4.2s	6Y
	Check the instructions for a task that was unclear before	27.1s	5Y - 1N
Likert Scale	Average of 4		

Table D.1: Summary of Usability Tests Results

E Appendix - UI/UX Wireframes

E.1 First Wireframe

Figure E.1.1: Initial Home Page

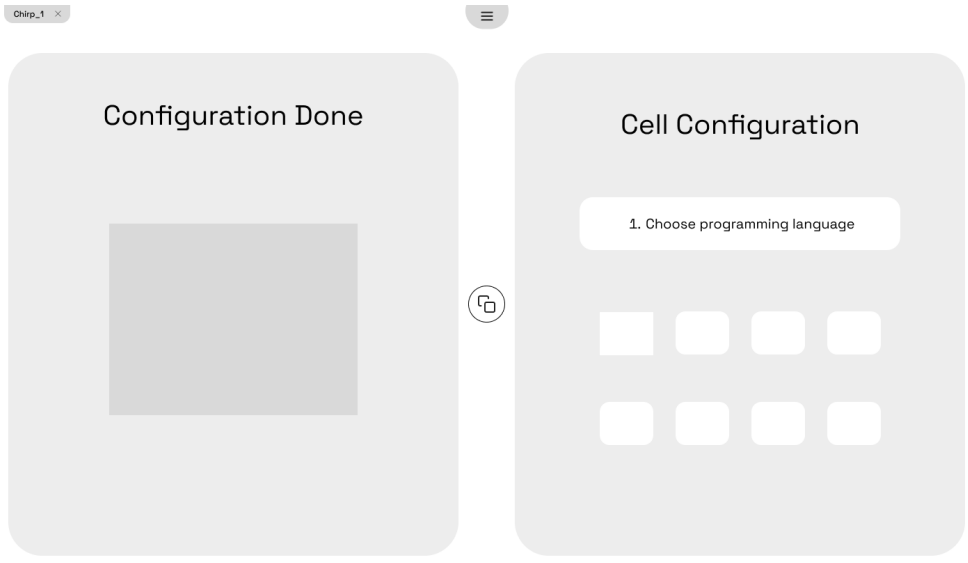

Figure E.1.2: Initial Add Cell Configuration

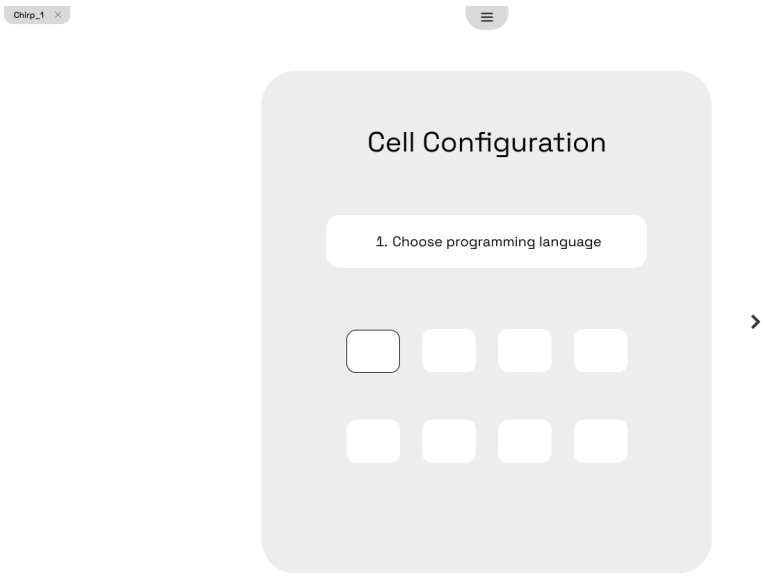

Figure E.1.3: Initial Code Cell Configuration

Figure E.1.4: Initial Code Page

Figure E.1.5: Initial Results Overlay

E.2 Second Wireframe

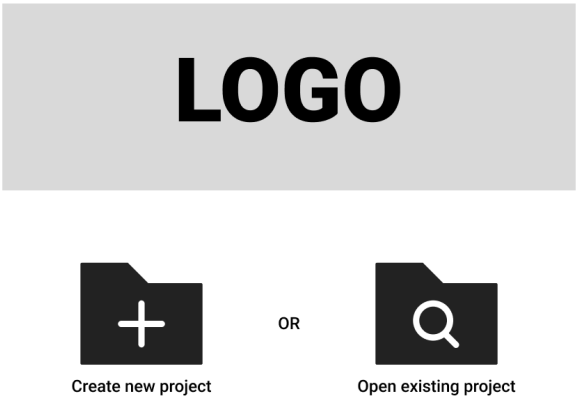

Figure E.2.1: Old Home Page

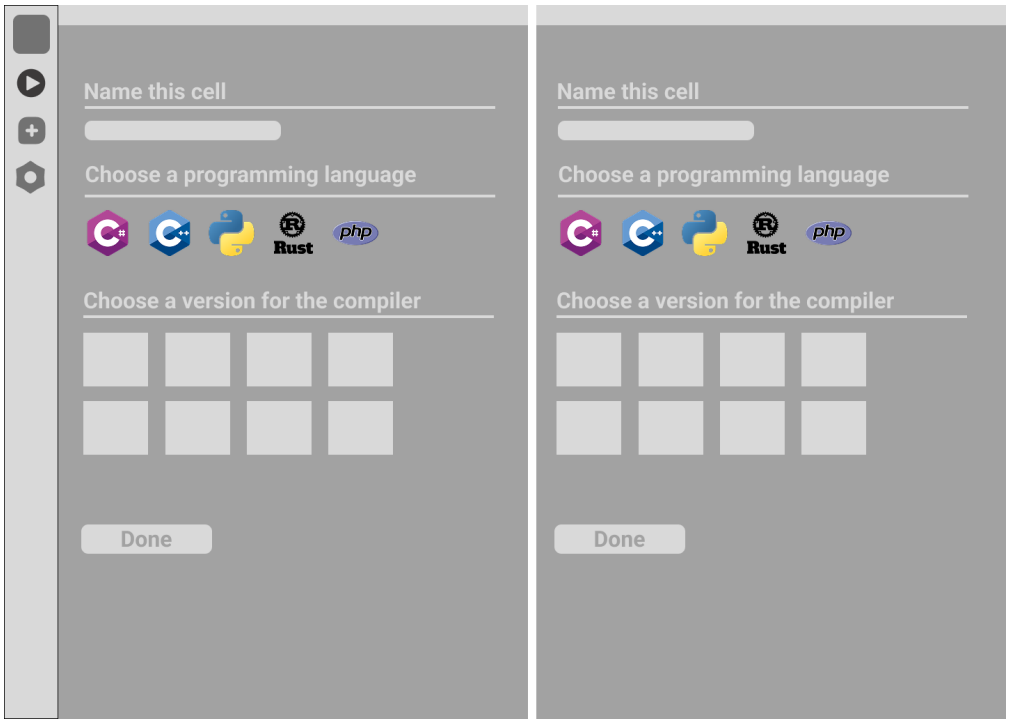

Figure E.2.2: Old Cell Configuration

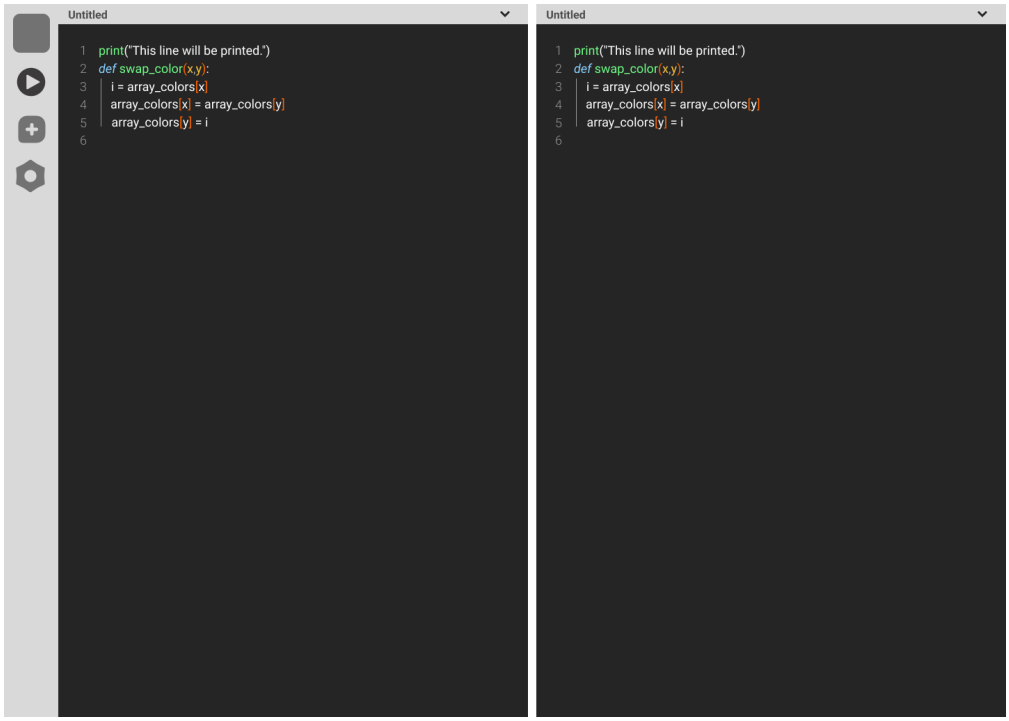

Figure E.2.3: Old Code Page

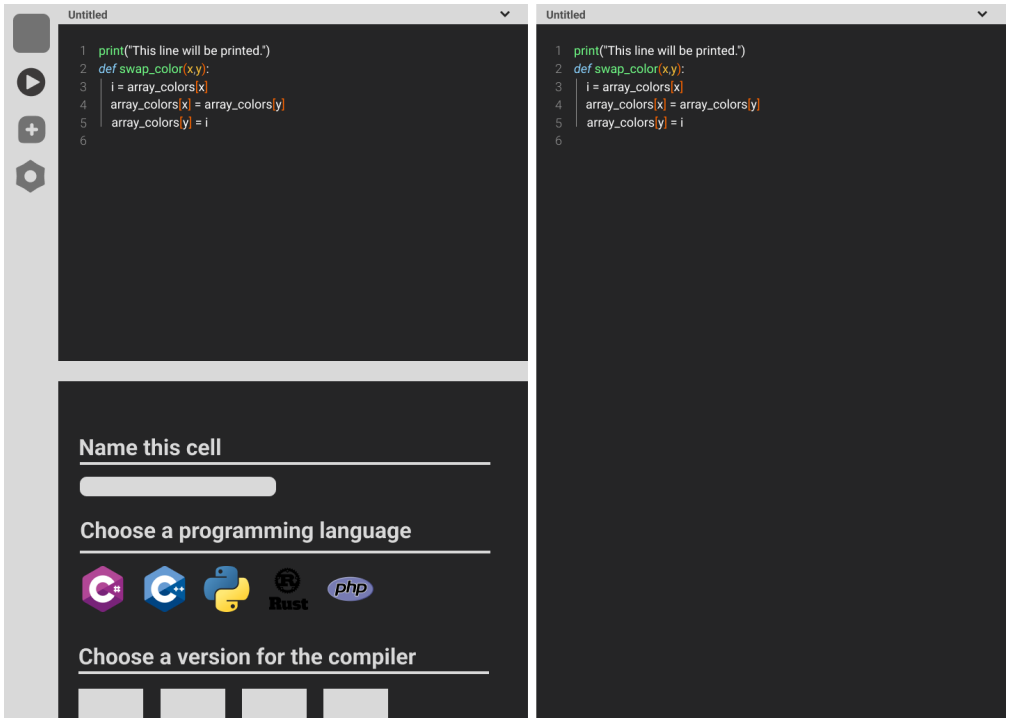

Figure E.2.4: Old Third Cell Layout

Figure E.2.5: Old Cells Result Overlay

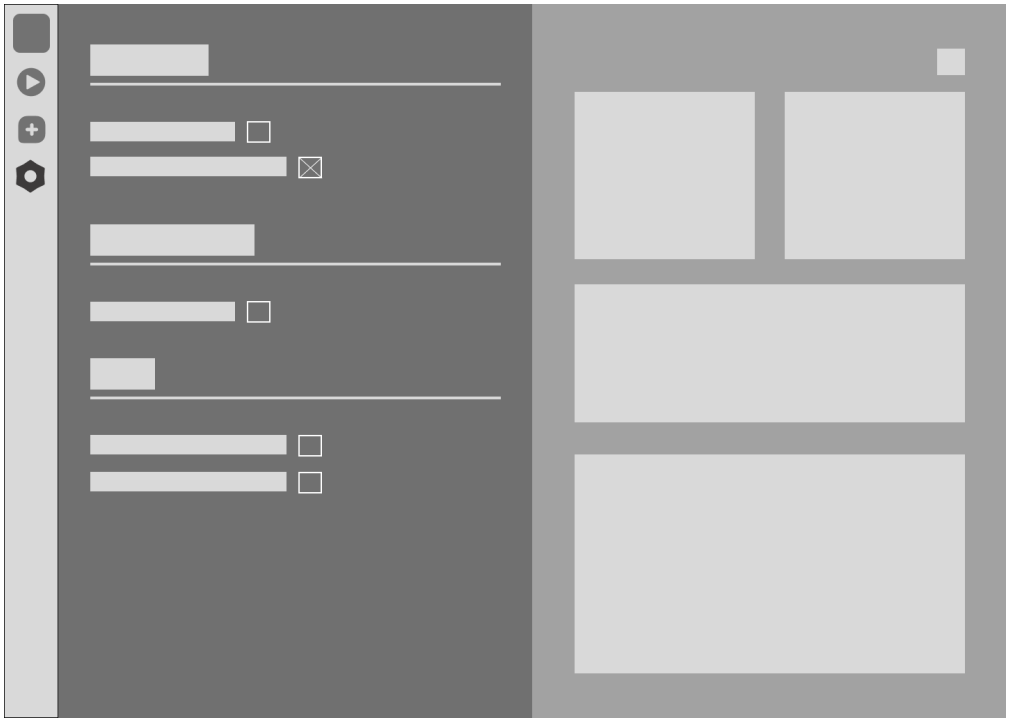

Figure E.2.6: Old Settings Page Structure

E.3 Final Prototype

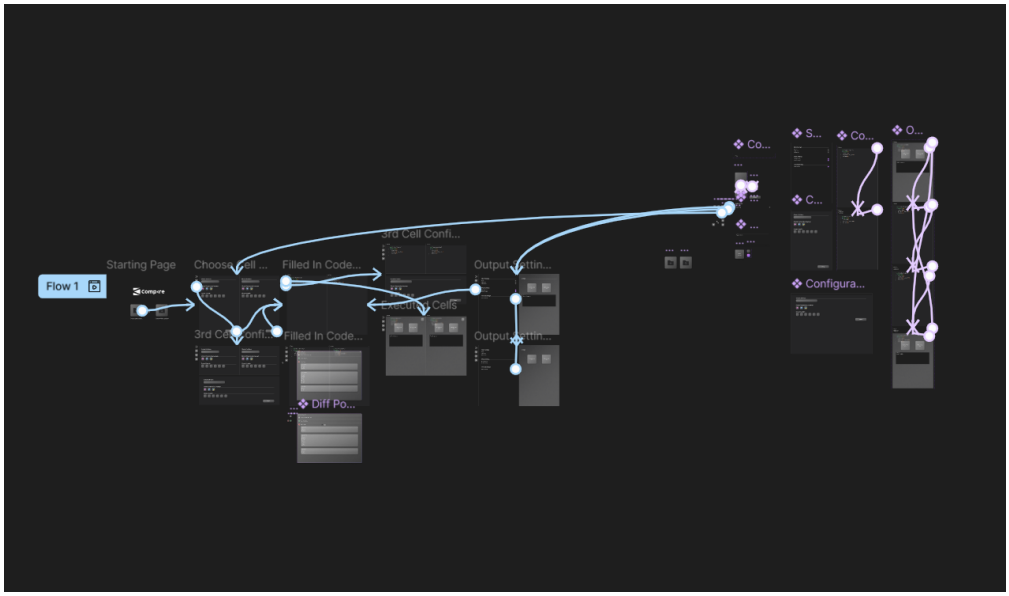

Figure E.3.1: Full Figma Prototype

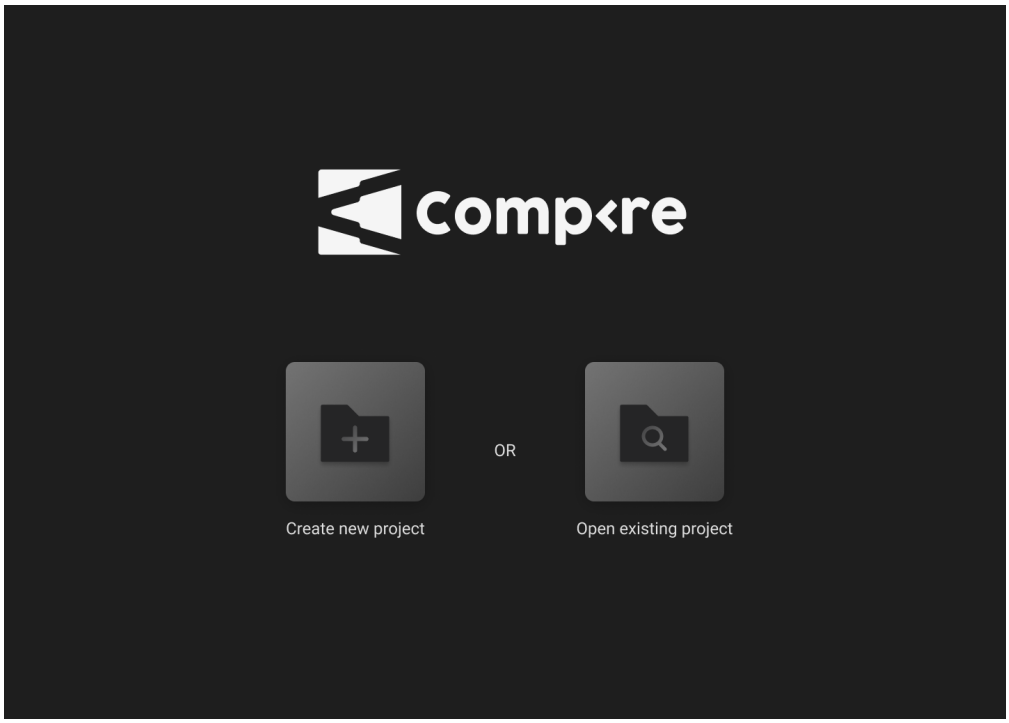

Figure E.3.2: Home Page

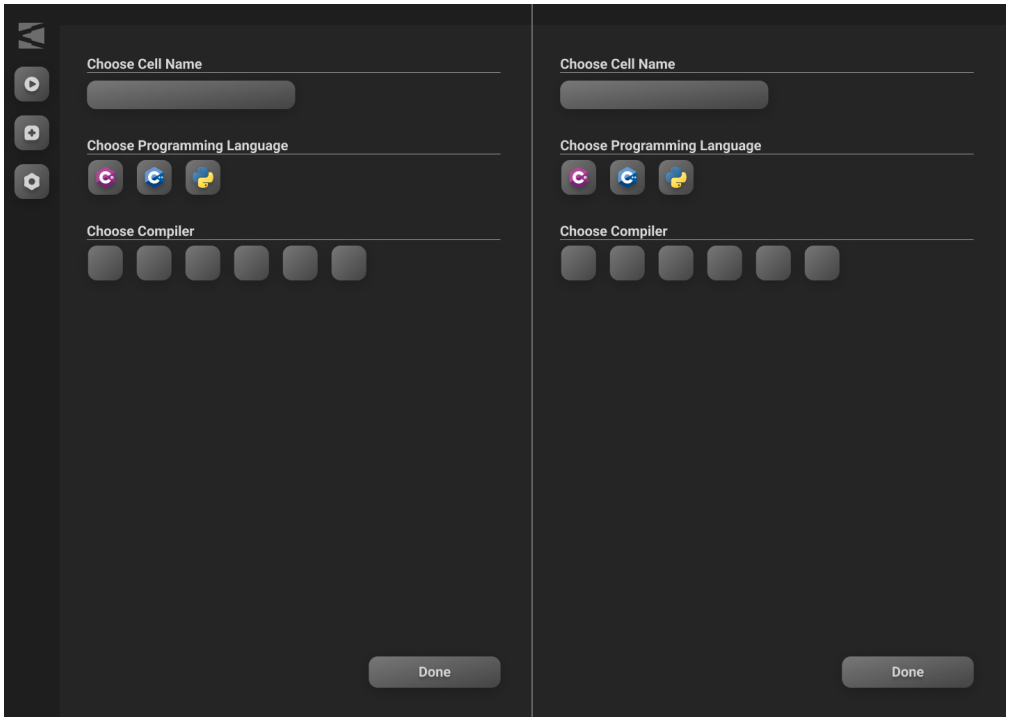

Figure E.3.3: Choose Cell Configuration Page

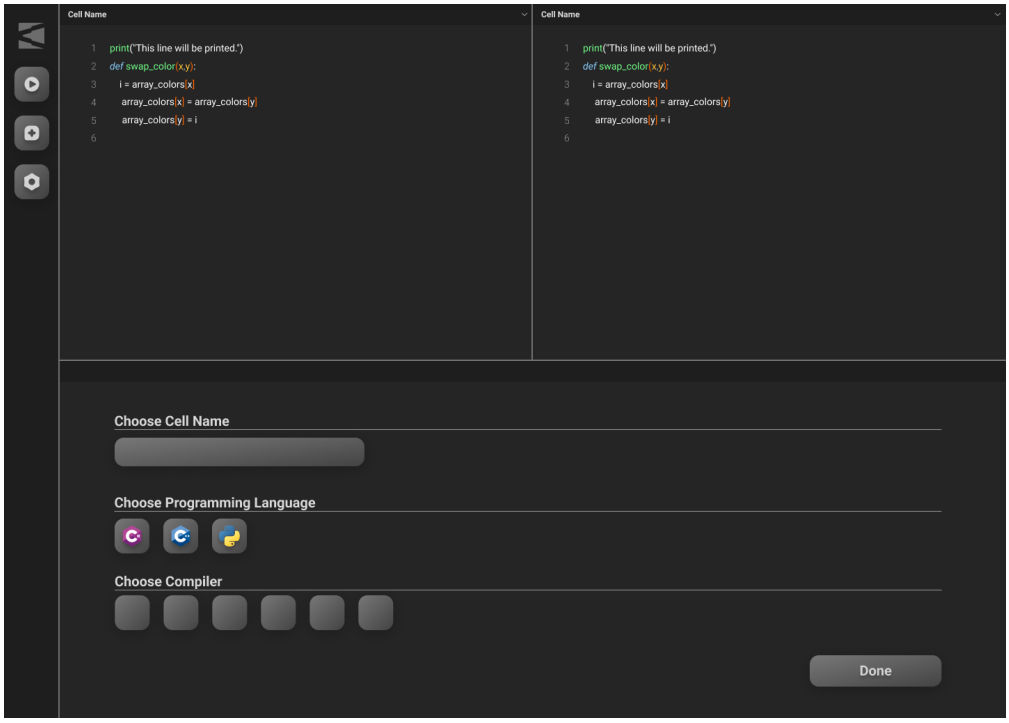

Figure E.3.4: Third Cell Layout

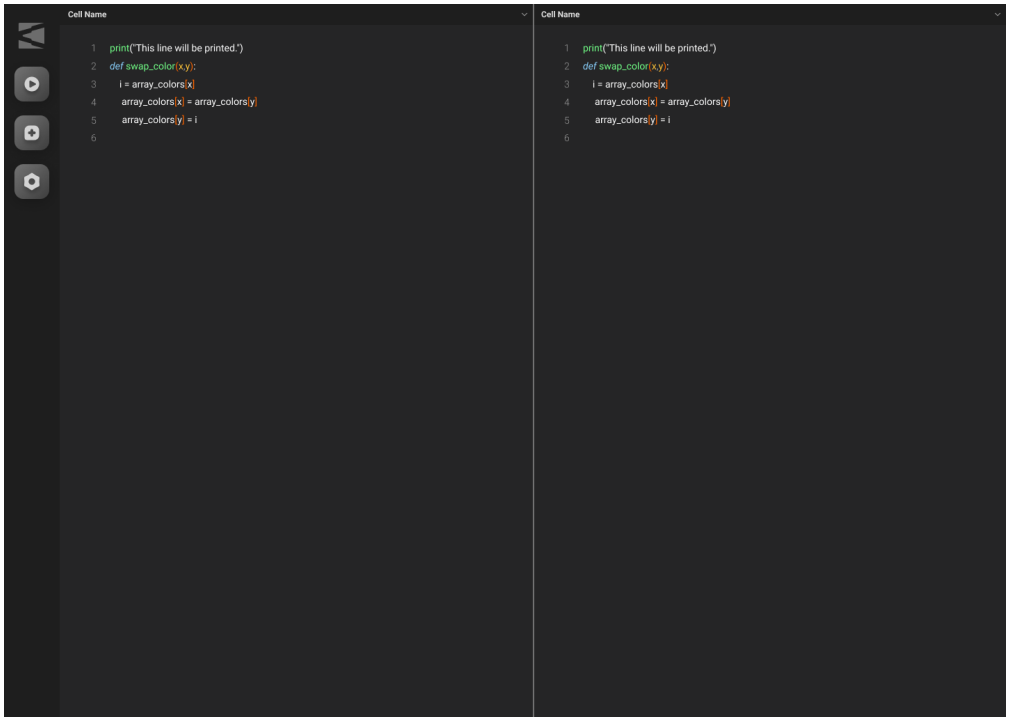

Figure E.3.5: Code Page

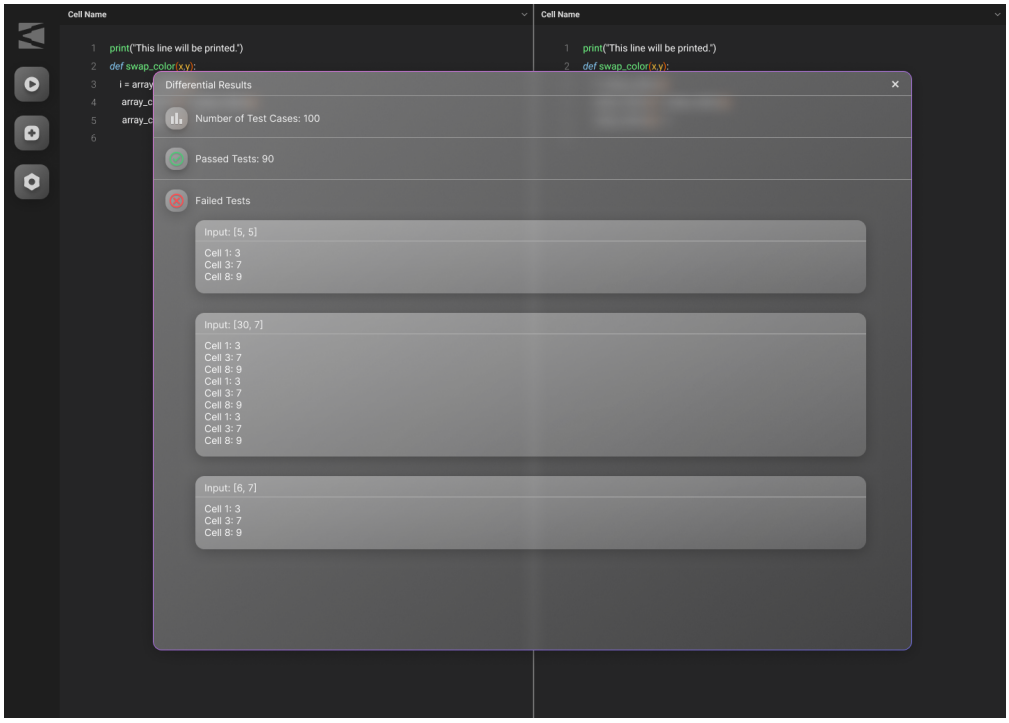

Figure E.3.6: Differential Results Modal

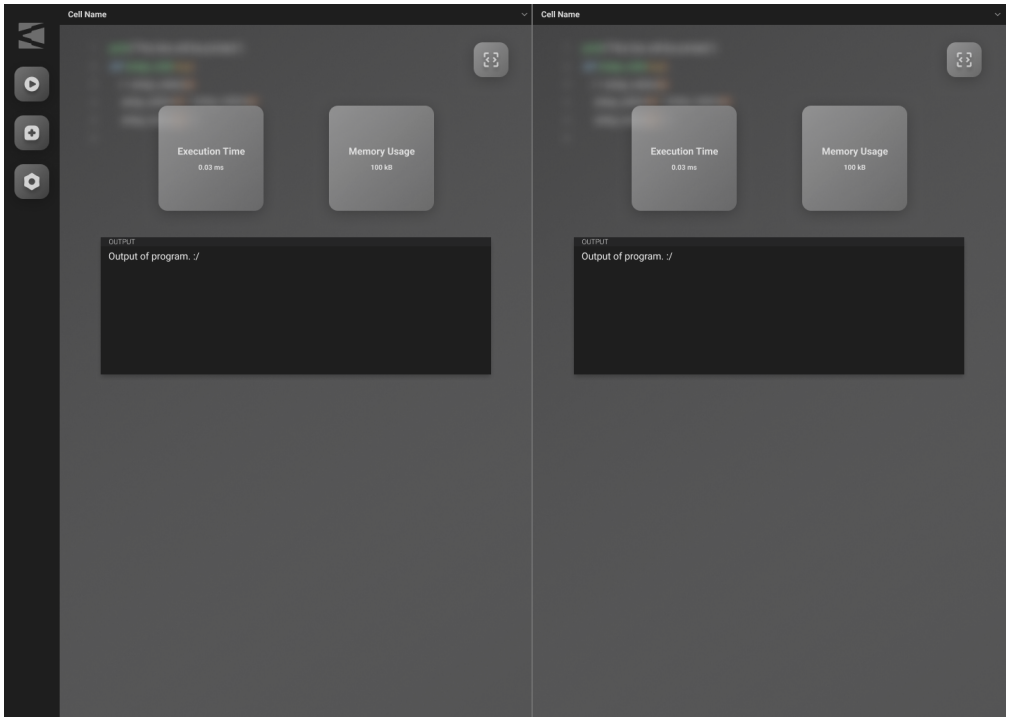

Figure E.3.7: Individual Cell Results Overlay

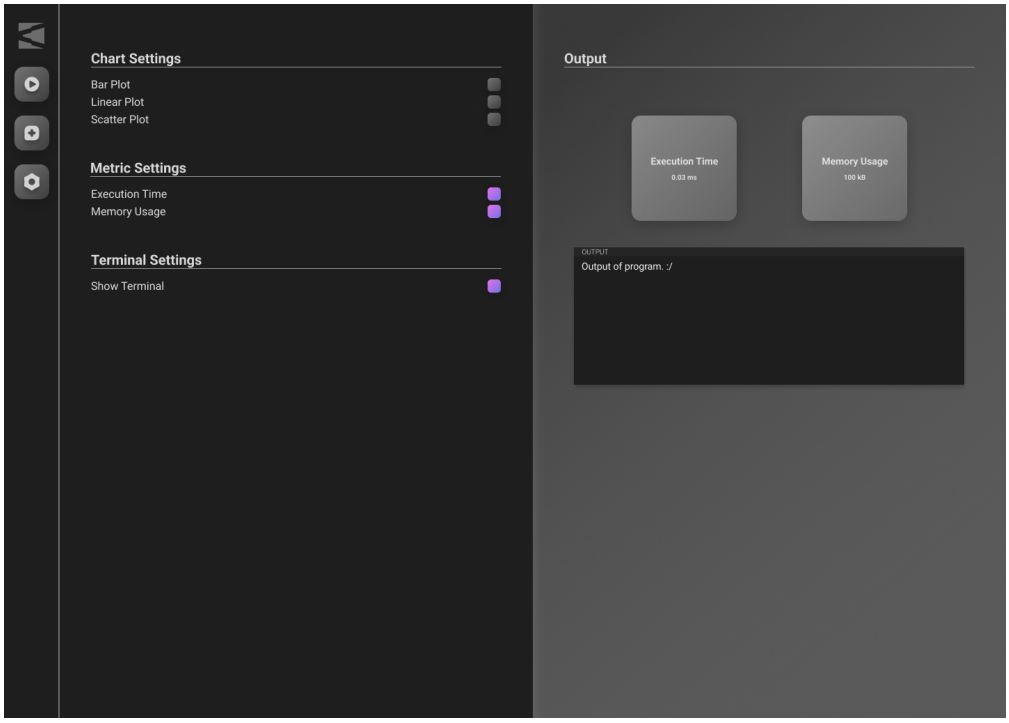

Figure E.3.8: Output Settings Menu

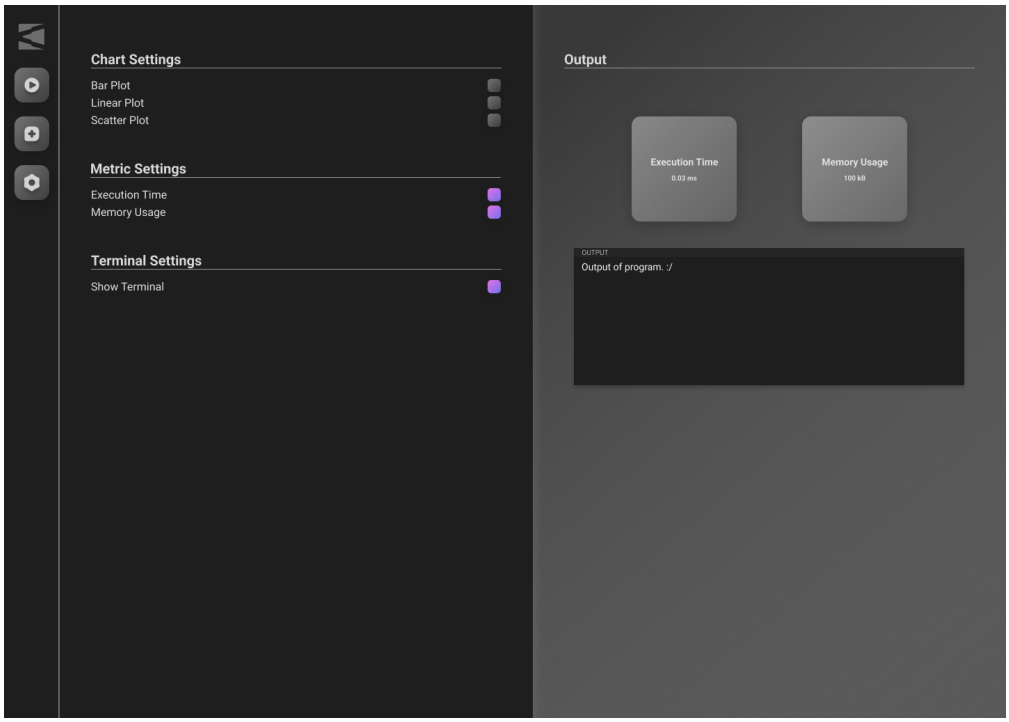

Figure E.3.9: Output Settings Menu

F Apendix - Final UI Look

Figure F.1: Home Page

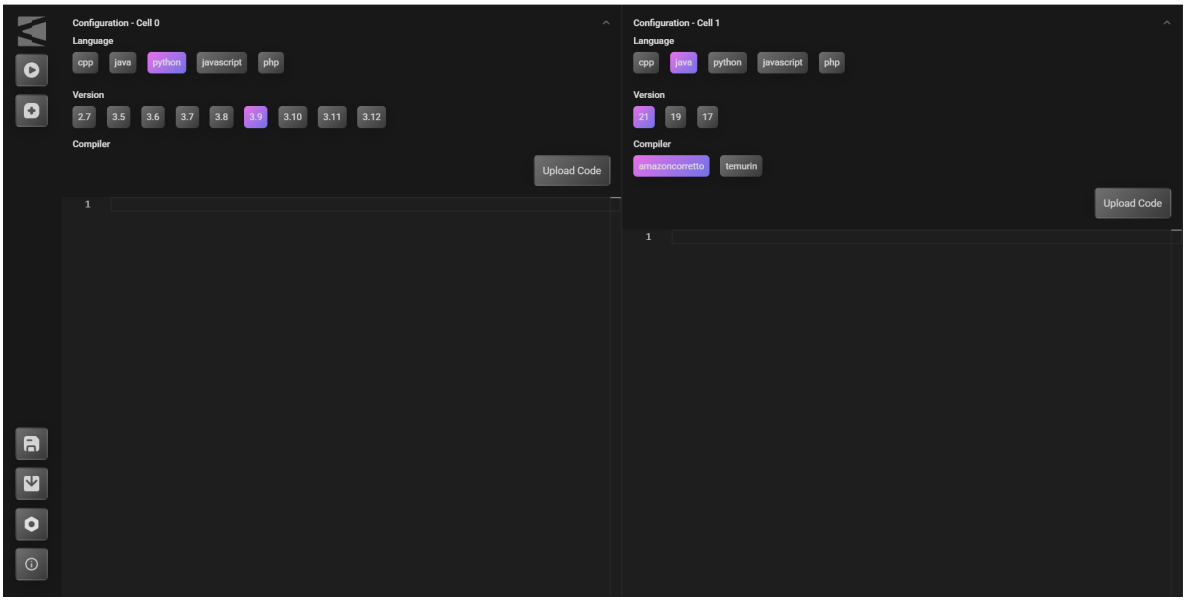

Figure F.2: Cell Configuration

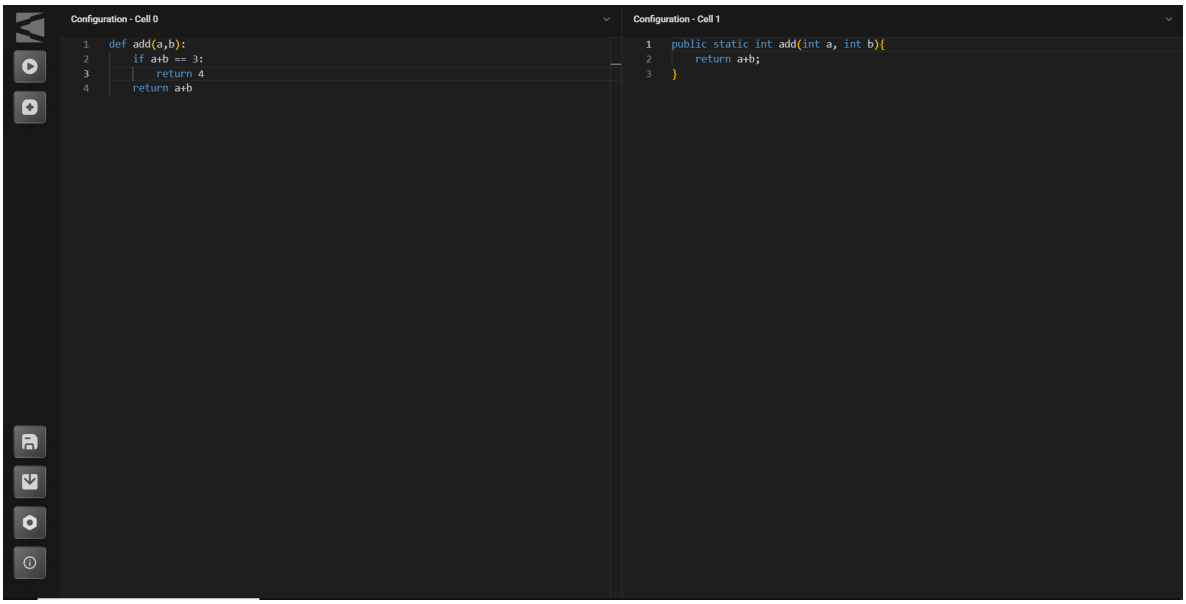

Figure F.3: Code In Program

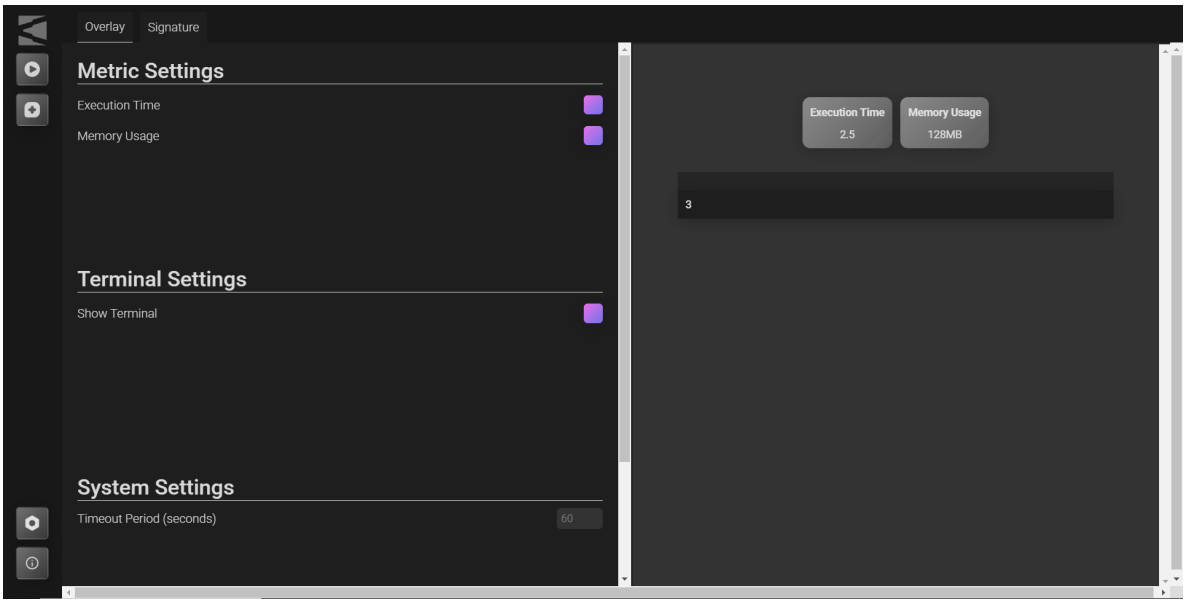

Figure F.4: Settings Page

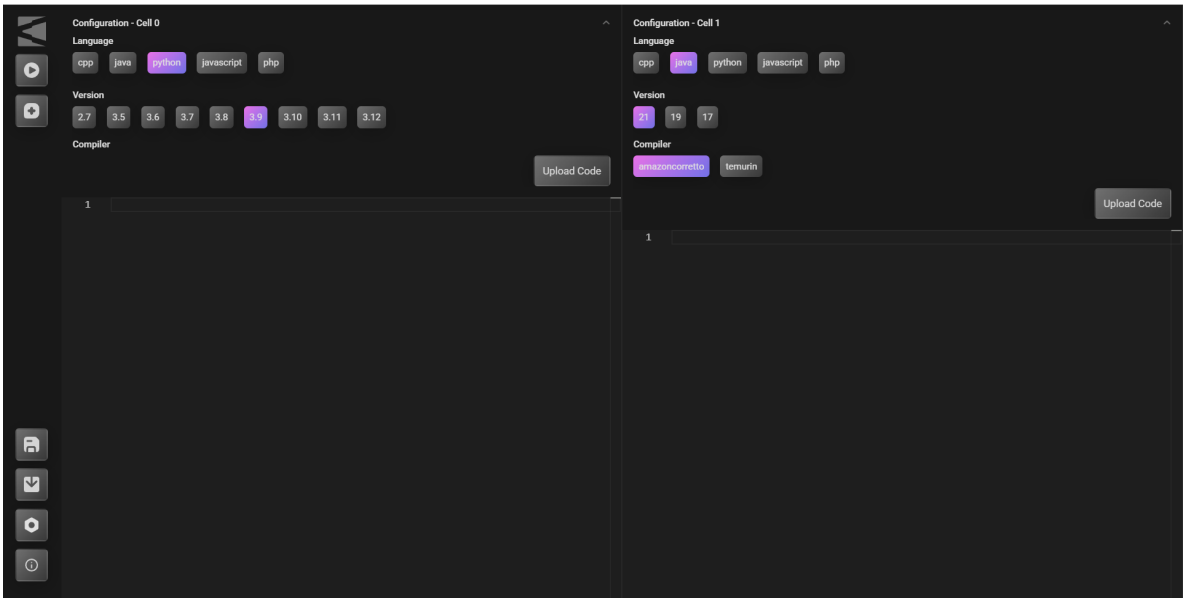

Figure F.5: Cell Configuration

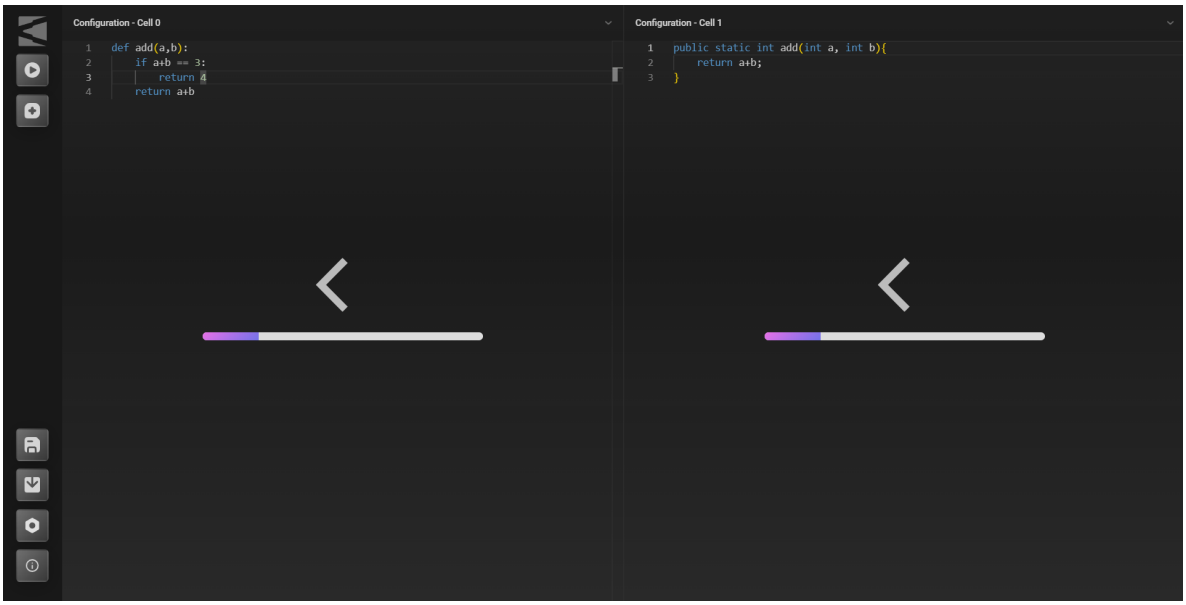

Figure F.6: Loading Screen

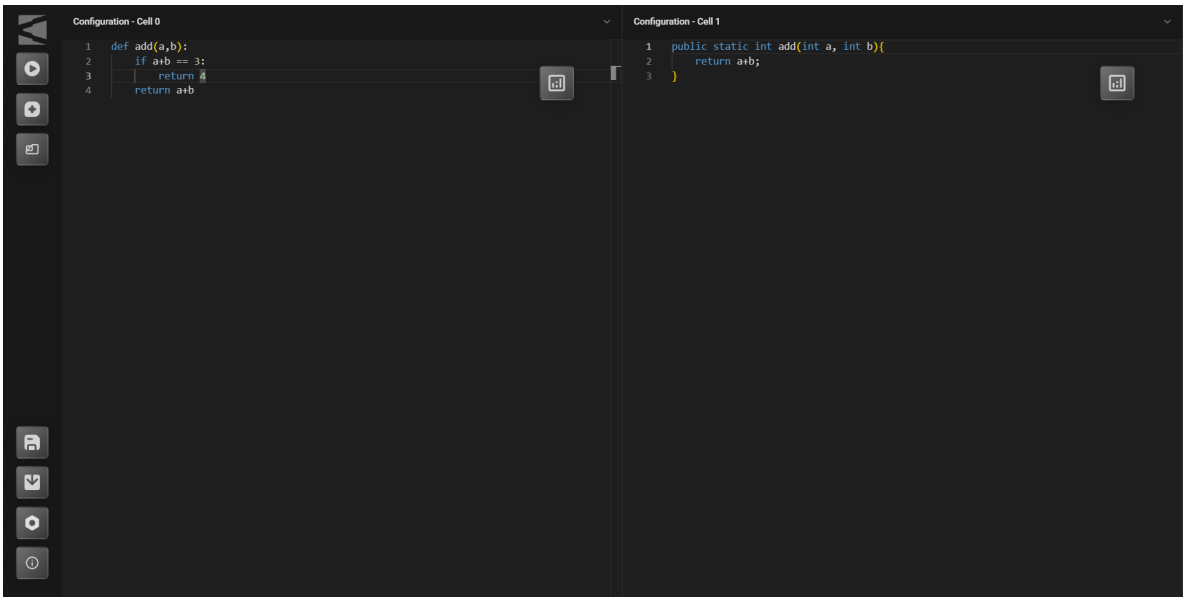

Figure F.7: Code Finished Execution

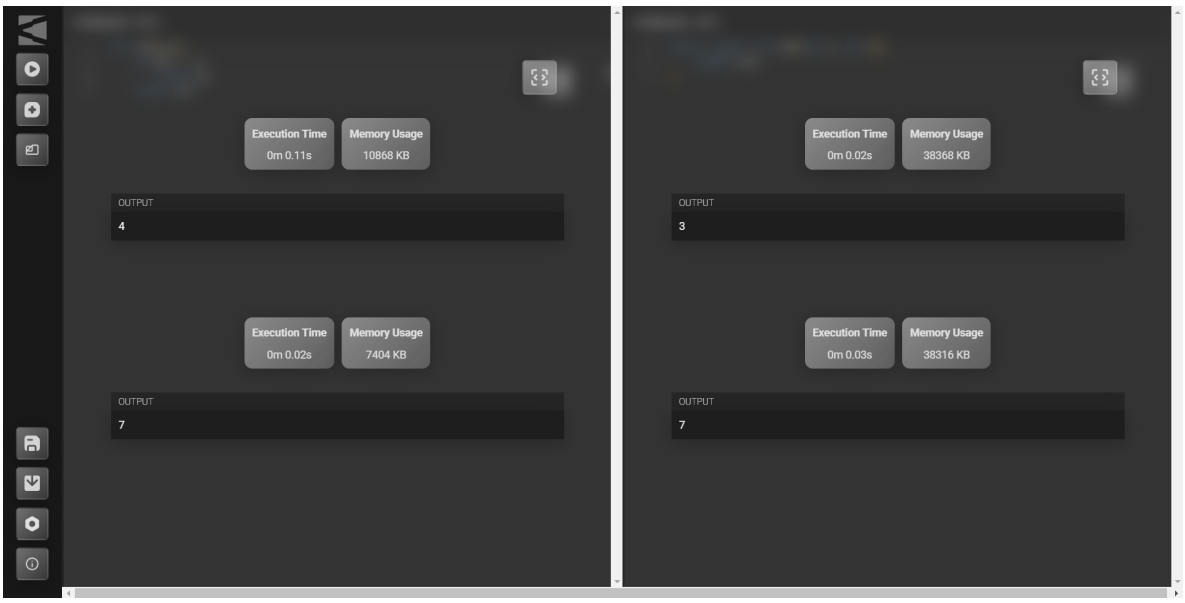

Figure F.8: Outputs and Metrics After Execution

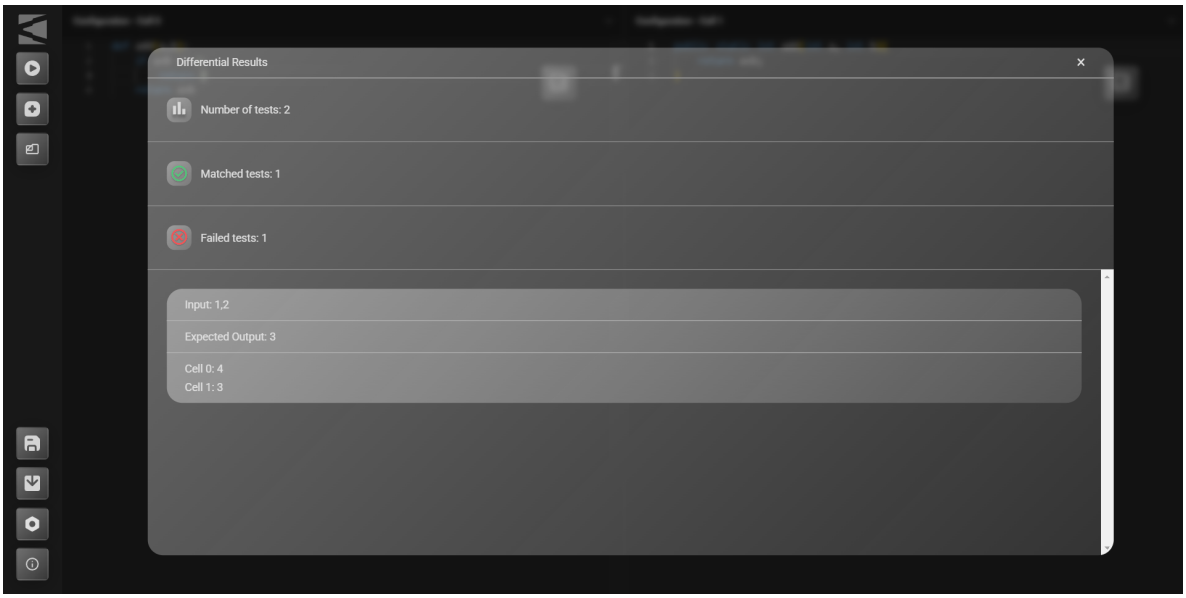

Figure F.9: Differential of Outputs

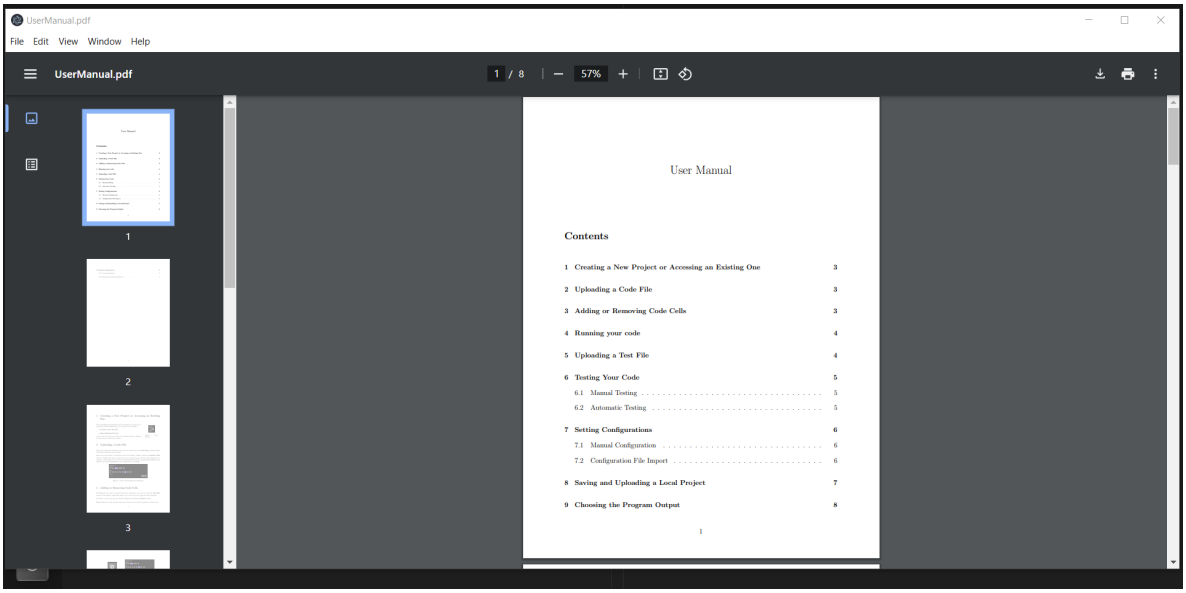

Figure F.10: User Manual

G Appendix - Likert Scale

What is the level of satisfaction regarding the interface of the system?

1. **Very Dissatisfied**
   I am not satisfied with the interface; it does not meet my expectations.
2. **Dissatisfied**
   I am not very satisfied with the interface; there are several usability issues.
3. **Neutral**
   I neither agree nor disagree; my satisfaction level is moderate.
4. **Satisfied**
   I am satisfied with the interface; it meets my expectations for usability.
5. **Very Satisfied**
   I am extremely satisfied with the interface; it exceeds my expectations.

H Appendix - Individual Contributions

Diana Bîrjoveanu - Unit Testing, Usability Testing
Ana Gavra - Frontend Developer, UI/UX, Project Manager
Ovidiu Lascu - Frontend Developer, UI/UX
Florin Priboi - Backend Developer, Integration Testing
Răzvan Ștefan - Backend Developer, Diagrams, Git Maintainer
Alexandru Zambori - Backend Developer, System Design and Integration