Saint Joseph University of Beirut

Department of Computer Science

Bachelor Research Project

Report

Automated Web Testing Framework Using a Restricted Natural Language

Submitted by

Antoine Karam
antoine.karam3@net.usj.edu.lb

Ghady Youssef
ghady.youssef@net.usj.edu.lb

Submission Date: January 13, 2025
Supervisor: Mr. Maroun Ayli

Abstract

Automated testing is essential for maintaining software quality and robustness in modern web development, where applications are increasingly dynamic and complex. However, traditional script-based testing tools like Selenium pose significant challenges for non-technical users due to their reliance on fragile web element locators such as XPATH and CSS selectors. These locators are prone to failure when application structures change, requiring frequent updates and substantial technical expertise. To address these challenges, this paper introduces a Domain-Specific Language (DSL) designed to simplify web testing by enabling non-technical users to define tests using natural, declarative syntax. The system leverages Large Vision-Language Models (LVLMs) to dynamically locate web elements based on user descriptions, enhancing the resilience of test scripts and reducing the need for manual intervention. By abstracting over traditional web testing technologies and integrating modern AI techniques, this approach aims to make automated testing more accessible and robust, supporting both technical and non-technical stakeholders.

Keywords: Software Engineering, Web Development, Theory of Compilers, Large Vision-Language Models, Automated Testing

Contents

Chapter 1

Introduction

1.1 Background

Automated testing is critical for ensuring software quality, especially in the realm of web development, where applications are increasingly complex and dynamic. Traditional script-based web testing tools like Selenium require users to locate elements on web pages using XPATH or CSS selectors. This approach presents challenges for technical and non-technical users alike, as accurately identifying these elements can be complex and error-prone. Additionally, as [3] highlights, web element locators are often fragile, meaning minor changes to an application's code can render tests unusable. This fragility is especially problematic when locators depend on specific CSS classes or XPATH expressions, leading to frequent test failures and increased maintenance efforts.

1.2 Importance

The significance of automated testing cannot be overstated, especially in the context of continuous integration and deployment practices. Organizations depend on automated testing to deliver high-quality software rapidly while minimizing the risk of bugs and failures. Automated testing not only saves valuable time and resources as proposed by [6] but also enhances collaboration between technical and non-technical stakeholders by providing a common language for test creation and execution.

1.3 Problem

Despite advancements in automated web testing, significant challenges remain. [8] observed that traditional testing tools often require substantial technical knowledge, making them inaccessible to non-technical users. Furthermore, the reliance on fragile locators undermines test reliability, particularly when application updates render locators obsolete. As [3] notes, "No locator is guaranteed to remain unchanged when the code changes, requiring time-consuming updates by developers." These issues highlight the need for a framework that simplifies web testing, reduces dependency on complex locators, and makes the process more inclusive for non-technical stakeholders.

1.4 Solution

This paper presents a Domain-Specific Language (DSL) to address these challenges. The DSL allows users to define tests using a natural, declarative syntax, eliminating the need for technical knowledge of XPATH or CSS selectors. Instead of specifying how to locate web elements, users describe the elements, and the system leverages Large Vision-Language Models (LVLMs), namely SeeClick [4], to infer and locate them accurately. This approach builds upon prior work, such as the framework proposed by [6], which sought to automate test generation based on user requirements. By integrating modern AI techniques, the DSL enhances the accessibility, resilience, and efficiency of web testing. Moreover, the DSL abstracts over existing tools like Selenium, translating user input into executable commands compatible with web drivers.

Chapter 2

Literature Review

This paper presents a platform where both technical and non-technical users can create, execute, and monitor tests, viewing live results as tests run directly within the web interface. This tool integrates with Continuous Integration (CI) pipelines, making it suitable for large-scale web applications deployed by enterprises, where automated testing is critical for maintaining software quality over time. Extensive research has been done in attempt to improve automated testing. Main ideas tackle topics such as script-based web testing, AI-based approaches and frameworks that utilize different tools such as LLMs and data structures.

2.1 Script-Based Testing

Automated web testing has seen significant advances in recent years, with a focus on improving test generation, abstraction of browser drivers, and simplifying the process for engineers. Tools such as *Selenium*, *Puppeteer*, and *Sahi* allow users to create automated tests by interacting directly with browser drivers. However, these traditional tools often require substantial technical knowledge, as said by [7], who evaluated various automated testing frameworks and have pointed out that script-based approaches can be tedious, time-consuming, and difficult to maintain. The study highlights the challenges faced by script-based methods. This paper proposes a framework that mitigates these issues by migrating from script-based testing to a more generic method. However, traditional methods are limited in terms of accessibility: script-based testing remains difficult to non-technical users. The aim of our solution is to provide a layer of abstraction to web drivers like *Selenium*, ensuring a more natural user experience for the testers.

2.2 Curiosity-Driven Web Testing

[8] proposed an innovative solution that incorporates AI to automate the exploration of web pages during testing. Their proposed framework, *WebExplor*, uses curiosity-driven reinforcement learning (RL) to discover complex test cases more effectively. "Its goal is to automatically generate diverse sequences of actions to explore more behaviors of the web application under testing", as mentioned by [8]. By allowing an AI model to explore the web application autonomously, the framework can uncover hidden bugs that may be missed during traditional testing approaches. This method represents a shift towards using AI not just for test automation, but also for intelligently exploring web

applications to find potential issues. This paper also utilizes LLMs to directly identify and interact with web elements, in contrast to relying on AI models solely for guiding the test automation process.

2.3 Automated Testing Frameworks

The Software Automated Testing (SAT) framework, developed by [6], aims to automate the process of writing and having to maintain test scripts. It converts well-structured test case steps into code that can be executed using tools like *Selenium*. SAT abstracts away the technical complexity, enabling testers without a programming background to create and maintain automated test scripts. This framework translates high-level business requirements into technical specifications, automatically generating the corresponding HTML locators for the elements being tested. In addition, this significantly reduces the time and effort required to maintain test scripts, especially in large-scale applications where frequent updates are made. The proposed approach aims to automate the process of creating and generating web tests while our solution aims to enhance the robustness of the tests generated by our framework.

2.4 Structured Web Testing

Another notable advancement in automated web testing is presented by [2], a novel technique called *FormNexus*, which focuses specifically on the automated generation of web form tests. This approach addresses the complexities involved in interpreting the context of input fields. It achieves this by restructuring the Document Object Model (DOM) layout of forms into a more organized structure known as the Form Entity Relation Graph (FERG). This transformation clarifies the semantics and relationships of form elements, enhancing their suitability for machine interpretation. By analyzing the characteristics such as textual content and the position of each node within the DOM hierarchy, *FormNexus* identifies similarities among various HTML nodes and uncovers potential relationships. This process offers valuable insights into the semantics of individual inputs and their interconnections, contributing significantly to the field of automated testing. The proposed framework uses a different methodology from our solution. However, the work acheived by [2] demonstrates that web testing technologies can harness the structure present in web pages to effectively generate tests. Similarly, the solution proposed in this paper leverages specific web elements such as buttons, images, and inputs to accurately locate these elements.

Chapter 3

Methodology

3.1 System Architecture

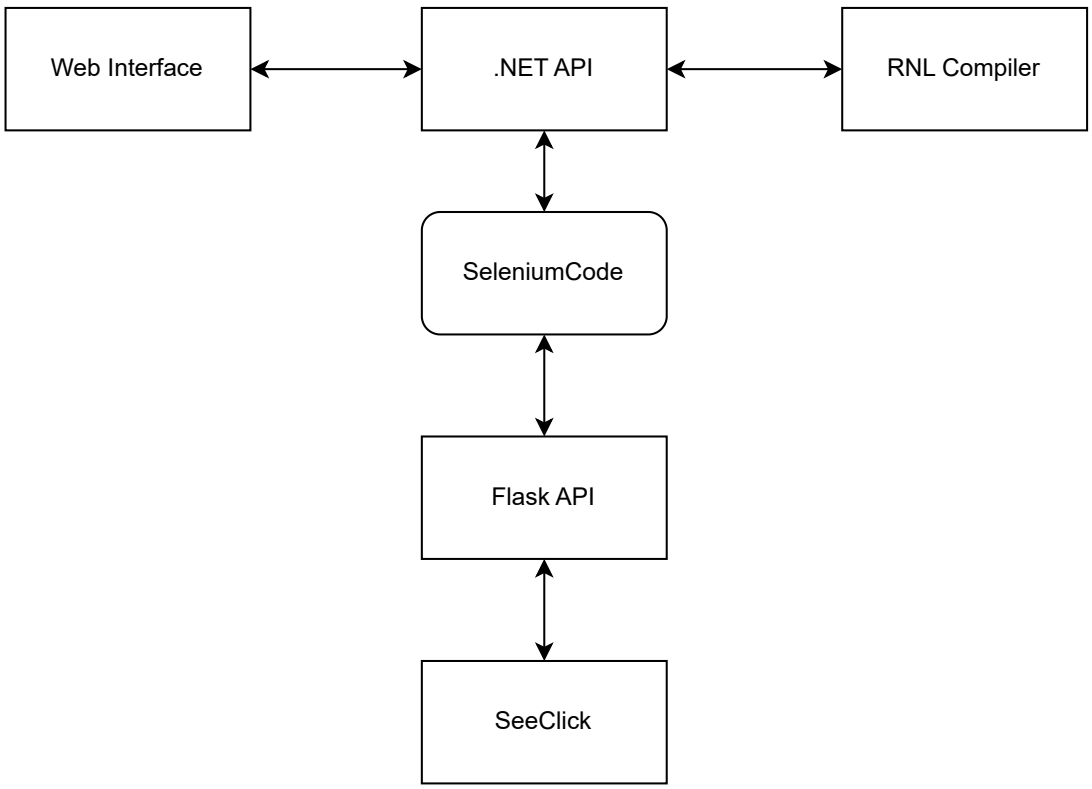

Figure 3.1: Block diagram showcasing the overview of the system

In this section, we explain how the various components of the system interact. The system is built around the core functionality of a restricted natural language (RNL) that translates test scripts into Selenium code.

The system begins with the **Client Application**, which serves as the user's primary interface. Users can interact with the application in two modes: a visual editor and a code editor. These modes allow them to create, edit, and manage test scripts, view a history of previous runs with detailed statistics (such as execution time and pass/fail status), and

rerun past test scenarios. Once a test script is prepared, the user initiates its execution.

Upon execution, the client establishes a **WebSocket Connection** with the server, which facilitates communication between the components. The server first calls the compiler to translate the restricted natural language (RNL) test script into Selenium commands. If the compiler encounters errors during this process, it sends detailed error messages back to the client via the WebSocket connection and terminates the interaction. In the absence of errors, the server wraps the generated Selenium code in WebSocket communication for real-time updates and executes the script.

The **Restricted Natural Language (RNL)** is the core component responsible for parsing and translating the user-provided test script. The compiler is implemented using an LL(1) parser, which generates an Abstract Syntax Tree (AST). This AST is then traversed by a visitor, implemented using the visitor design pattern [5], to translate the test script into Selenium JavaScript code. Error handling is managed through panic mode recovery [1], which employs a set of synchronizing tokens to recover from parsing errors and provide users with detailed error reports rather than halting at the first issue.

Once the RNL test script is translated, **Selenium WebDriver** automates browser actions based on the parsed commands. Selenium interacts with web pages to perform tasks such as clicking buttons, entering text, and verifying results. This ensures that the test scenarios specified in the RNL are accurately executed in the browser.

```
usjSignIn {
    visit "https://etudiants.usj.edu.lb"
    type "123456" on input with description "matricule field"
    type "password" on input with description "password field"
    type "123456" on input with description "captcha field"
    check if button with description "the login button" is displayed
    click button with description "the login button"
}
```

Listing 3.1: Example test using the restricted natural language

The system integrates with **SeeClick**, a model designed to gather specific coordinates based on a natural language description and a UI image. During test execution, Selenium captures screenshots at runtime, which are passed to SeeClick along with a description and type of the desired element. SeeClick [4], a model with over 9 billion parameters, processes this input to map the positions of web elements with precision, enabling accurate simulation of user interactions during tests.

Due to the high computational demands of SeeClick, the system utilizes **External Hardware** for processing. A high-performance machine equipped with a AMD EPYC 9124 processor (16 shared cores) and an NVIDIA H100 SXM GPU with 80GB of VRAM was rented. This setup ensures that SeeClick runs efficiently, supporting the system's ability to handle large-scale tests and dynamic web environments seamlessly.

3.2 Design and Implementation

The system was designed to be modular, with clear separation between the user interface, the restricted natural language compiler, Selenium, and SeeClick integration. This modularity allows the system to be flexible and scalable, making it easier to expand or modify individual components. The user interface was designed to support both a visual editor and a code editor, which enables users to interact with the system in the way that best suits their preferences.

The interaction between the client application and the server is established through a WebSocket connection, ensuring real-time communication between the components. The server handles the heavy lifting of compiling and executing test scripts, while the client is responsible for displaying the results and managing the user's interaction with the system.

3.3 Tools and Technologies

The following tools and technologies were used in the development of this project:

- **Selenium WebDriver:** Browser automation tool used to execute the commands parsed from the restricted natural language.
- **SeeClick:** An multi-modal model used to locate elements on a website by analyzing screenshots.
- **Python:** Used to encapsulate SeeClick within an API written in Flask.
- **JWT:** Used for authentication and authorization via web tokens.
- **.NET:** Framework used to implement the server.
- **C++ (Compiler):** Used to design and implement the restricted natural language parser, converting user input into executable commands for Selenium.
- **Rented High-Performance Machine:** Utilized for testing purposes, providing the necessary computational resources for SeeClick.
- **WebSocket Communication:** Enables real-time communication between the client and server during test execution.
- **Client Application:** Built using React and TypeScript with the `shadcn` component library for a user-friendly interface.
- **JavaScript (Mocha):** Used to execute Selenium code and perform assertions during tests.
- **Docker and Docker Compose:** Used to containerize the services for seamless deployment.

3.4 Testing and Validation

During the validation phase, the system was tested in multiple environments to ensure correct functionality. A rented machine was used to simulate real-world conditions, allowing us to run tests at scale and observe how the components interacted.

Several test cases were developed to verify the system's behavior. These included tasks such as clicking buttons, filling forms, and navigating between pages. Each test case was designed to confirm the system's reliability, with any failures analyzed to determine root causes.

The system's performance under load was also validated to ensure stability during concurrent test executions. The WebSocket connection between the client and server was rigorously tested to verify that live updates were correctly transmitted, allowing users to monitor test execution in real time.

$$
\begin{aligned}
\langle program \rangle &::= \langle test \rangle\ \langle program \rangle \mid \epsilon \\
\langle test \rangle &::= \texttt{TEST\_NAME}\ \{\ \langle body \rangle\ \} \\
\langle body \rangle &::= \langle action \rangle\ \langle body \rangle \mid \epsilon \\
\langle visit \rangle &::= \texttt{VISIT URL} \\
\langle click \rangle &::= \texttt{CLICK}\ \langle elem\_type \rangle\ \texttt{WITH\_DESC NLD} \\
\langle check \rangle &::= \texttt{CHECK\_IF}\ \langle elem\_type \rangle\ \texttt{WITH\_DESC NLD}\ \langle state \rangle \\
\langle type \rangle &::= \texttt{TYPE CONTENT ON}\ \langle elem\_type \rangle\ \texttt{WITH\_DESC NLD} \\
\langle state \rangle &::= \texttt{DISPLAYED} \mid \texttt{HIDDEN} \\
\langle action \rangle &::= \langle click \rangle \\
&\quad \mid \langle check \rangle \\
&\quad \mid \langle type \rangle \\
&\quad \mid \langle visit \rangle \\
\langle elem\_type \rangle &::= \texttt{BUTTON} \\
&\quad \mid \texttt{LINK} \\
&\quad \mid \texttt{TEXT} \\
&\quad \mid \texttt{IMAGE} \\
&\quad \mid \texttt{INPUT}
\end{aligned}
$$

Figure 3.2: The LL(1) grammar for the restricted natural language

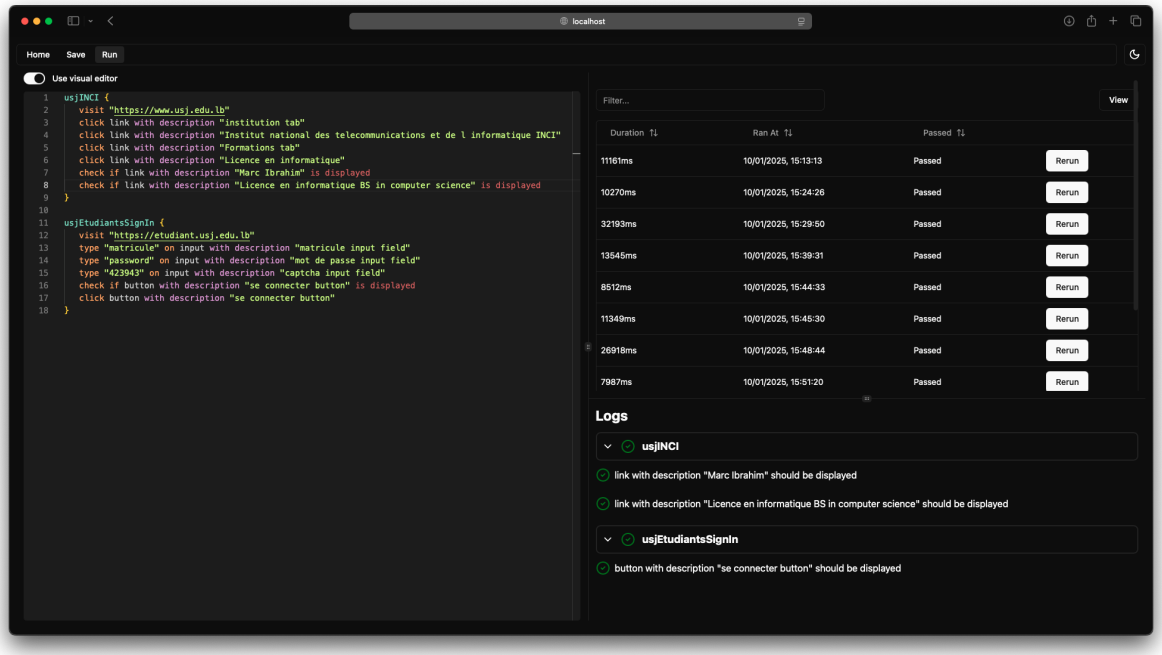

Figure 3.3: Screen with code editor

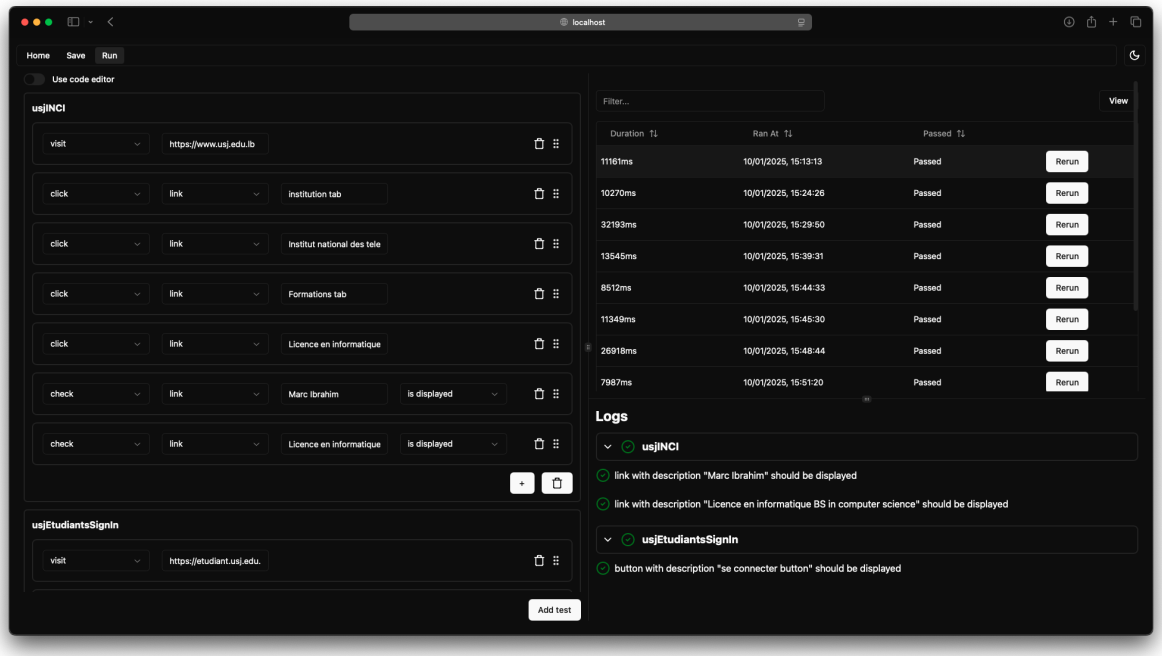

Figure 3.4: Screen with UI-based editor

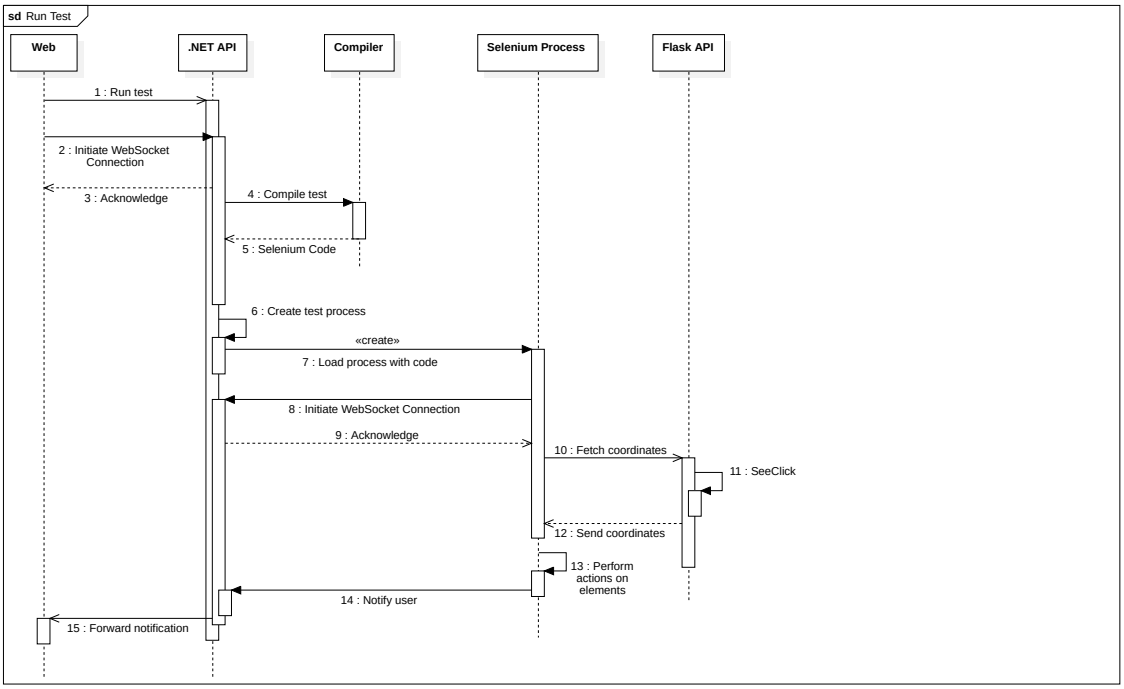

Figure 3.5: Sequence diagram for the run test action

Chapter 4

Results

4.1 Evaluation

The implementation of the restricted natural language compiler and its integration with Selenium has shown promising results. A variety of test cases were executed, focusing on the interaction between the Selenium commands and web elements using the SeeClick model. This functionality has been thoroughly tested across different dynamic web elements, and the accuracy of detecting and interacting with such elements was found to be consistently high. The interface reliably clicks on elements that meet the visibility conditions, even under varying network loads and page render times.

4.2 Performance

In terms of performance, the SeeClick inference pipeline was evaluated for its response time and efficiency. On average, the time for generating coordinates and executing the click interaction ranged from 0.5 to 1.5 seconds. This latency is considered an acceptable overhead given the complexity of ensuring that dynamic and hidden elements are properly detected before interaction. Considering the robustness of the solution, this performance tradeoff is deemed a reasonable price to pay for the increased reliability and scalability in testing web applications.

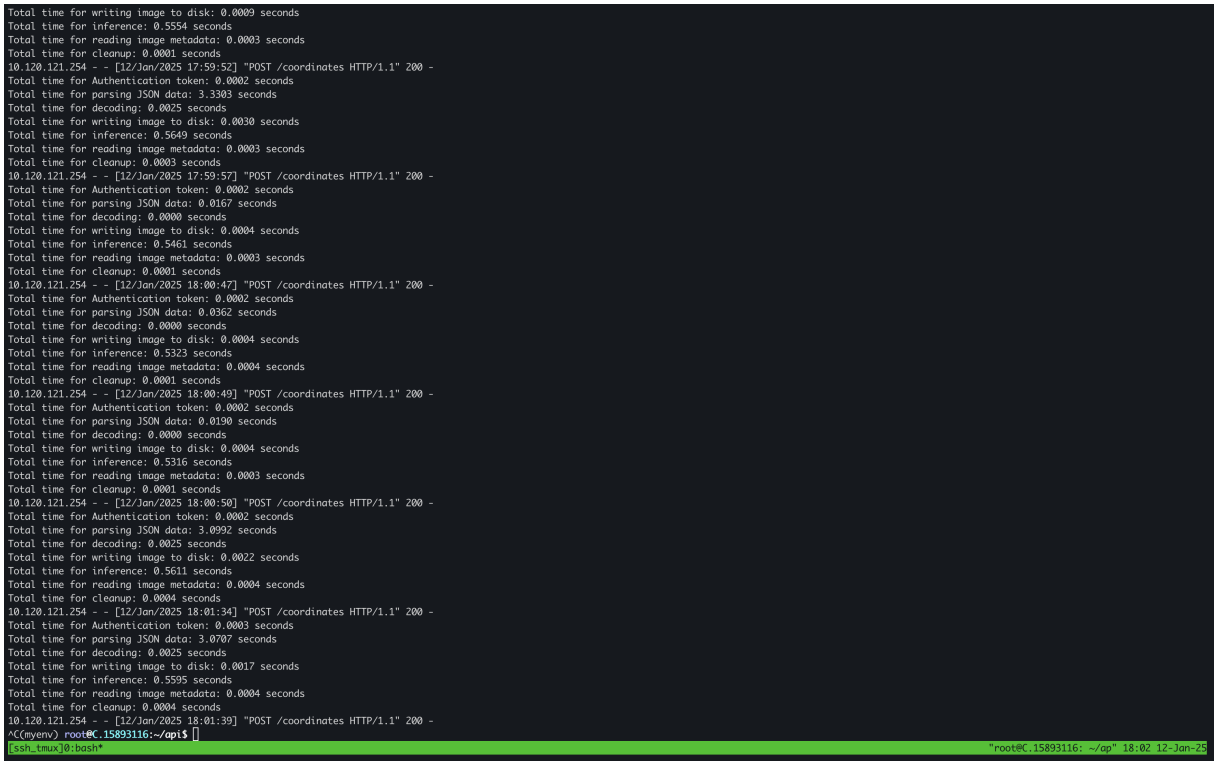

Figure 4.1: Benchmark of the SeeClick coordinate generation pipeline

Chapter 5

Discussion

This study demonstrates several important outcomes regarding the development and usage of the web-based platform for automated web testing.

5.1 Efficient Test Creation

Firstly, the average time required to create a test was significantly reduced due to the intuitive design of the restricted natural language and the embedded text editor. In addition, users without technical expertise could efficiently create tests, using the simplified syntax and drag-and-drop features, without needing knowledge of CSS or XPATH selectors.

5.2 Improved Test Robustness

In terms of success rate, the platform showed a high level of accuracy in identifying and interacting with web elements. The restricted natural language, in conjunction with dynamic resolutions powered by LVLMs, ensured that test scripts accurately described the intended interactions, even in complex scenarios.

5.3 Optimized System Performance

From a performance standpoint, the system demonstrated impressive efficiency. The execution times were within acceptable ranges, even under various load conditions. The platform maintained a high throughput and low response time, which is essential for handling multiple test scenarios concurrently without significant delays.

Overall, the results suggest that the platform provides an effective, friendly, and reliable solution for automated web testing, offering significant improvements in terms of speed, accuracy, and performance when compared to traditional testing methods.

Chapter 6

Conclusion

In conclusion, this study demonstrates the successful development of a web-based platform aimed at simplifying automated web testing for both technical and non-technical users. By introducing a restricted natural language (RNL) for test creation, alongside an intuitive user interface, the platform allows users to define and execute tests without the need to understand complex programming concepts like XPATH or CSS selectors.

The system's design, which integrates modern Large Vision-Language Models (LVLMs) such as SeeClick, significantly enhances the resilience and flexibility of the testing process by dynamically locating web elements based on user descriptions. This abstraction over traditional testing technologies, combined with the use of AI, minimizes the need for manual updates and reduces the complexity of test maintenance, ensuring that the system is robust to changes in web application structures.

The performance of the system has been validated through rigorous testing, where it successfully handled various test scenarios with low execution times and high throughput. The WebSocket communication allowed for real-time updates, making the testing process transparent and interactive for users. Additionally, the system's ability to simulate user interactions with precision further contributes to its reliability and usability.

Overall, this study provides evidence of the platform's potential to streamline and accelerate the web testing process, making it more accessible to a broader audience. By simplifying test creation, enhancing test resilience, and integrating cutting-edge AI, this platform represents a significant advancement in the field of automated web testing, offering substantial benefits to both technical and non-technical stakeholders.

Chapter 7

Future Work

The following areas for improvement are identified:

- **Scalability and Distributed Execution:** Develop a service to handle text execution for scalability and distribute the execution across multiple machines to speed up the process. Utilizing cloud infrastructure or distributed systems would be beneficial in handling large-scale tests.
- **Expansion of Selenium Interface:** Extend the Selenium interface by adding additional classes and support for complex interactions, including a dedicated method for handling the ‘SeeClick‘ model.
- **Compiler Enhancements:** Improve the restricted natural language (RNL) compiler by adding support for more actions.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] Parsa Alian, Noor Nashid, Mobina Shahbandeh, and Ali Mesbah. Semantic constraint inference for web form test generation, 2024.

[3] Maroun Ayli, Youssef Bakouny, Nader Jalloul, and Rima Kilany. Enhancing the resiliency of automated web tests with natural language. In *Proceedings of the 2024 4th International Conference on Artificial Intelligence, Automation and Algorithms*, AI2A '24, page 63–69, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9313–9332, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[6] Milad Hanna, Amal Elsayed Aboutabl, and Mostafa-Sami M Mostafa. Automated software testing framework for web applications. *International Journal of Applied Engineering Research*, 13(11):9758–9767, 2018.

[7] Mohamed Monier and Mahmoud Mohamed El-mahdy. Evaluation of automated web testing tools. *International Journal of Computer Applications Technology and Research*, 4(5):405–408, 2015.

[8] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. Automatic web testing using curiosity-driven reinforcement learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, pages 423–435. IEEE Press, 2021.