Self-Healing in Selenium Automation: A Comparative Study of Healenium and a custom Large Language Model-Based Approach

Daniel Keiss [1]

Abstract: For decades, the Selenium open-source framework has been widely used for regression testing of web interfaces. Many companies have a large inventory of tests and face the challenge of investing significant time in maintaining their test automation. This effort arises from frequent rework, caused by changes in web interfaces or unexpected delays in accessing web elements. This paper examines how self-healing approaches using large language models can help reduce manual effort. An example implementation compares the Selenium self-healing solution, Healenium, with a newly implemented LLM-driven solution, AiCurator.

Keywords: Selenium, Test Automation, Self-Healing, Large Language Model, Artificial Intelligence, Healenium, AICurator

1 Introduction

Selenium's long history is dating back to 2004 when Jason Huggins initially implemented it at ThoughtWorks in the form of *JavaScriptTestRunner*. This early version lays the foundations for WebDriver and Selenium Grid, which are still in use today [Coa]. In 2018, the W3C standard for WebDriver was published, and browsers began offering a standardized interface for automation [Cob].

Thanks to W3C standardization, cross-browser support, and a well-established open-source community, Selenium is widely used in enterprises for a long time. Many companies have accumulated a large inventory of Selenium tests in the last years. This inventory requires continuous maintenance, especially as user interfaces evolve throughout the lifecycle of a project. Therefore, the locators with are used to access the web elements, often need to be updated to reflect the changes [Le23].

To reduce maintenance efforts, the self-healing tool Healenium attempts to repair failed locators during runtime. Selenium stores every successful locator, whether it worked initially or was successfully healed, in a PostgreSQL database. Healenium uses a heuristic approach, which identifies the next most likely locator in comparison to the failed one. If everything goes well, Healenium provides a healed locator during runtime and test continues. This approach aims to minimize the manual efforts required for test maintenance. In case of success, developers know that the locator works and only needs to verify its functional correctness before integrating it into the code [Kh23].

This paper compares Healenium, based on the mentioned heuristics, with a custom-built solution based on a large language model (LLM) called AICurator. All experiments and final results are based on the use of AICurator with Claude3. The model was selected as the primary LLM for this work because it is the fastest, and it delivers the best coding results according to general tests [Mi].

The first goal of this work is to examine whether the LLM approach is capable of healing locators. Following this, Healenium and AiCurator are compared directly with each other. The AiCurator solution is expected to compensate its speed and cost disadvantages, with its reasoning capabilities regarding the underlying context of the web page. Additionally, potential benefits of the AI approach, such as the automated generation of pull requests across multiple layers, will be explored.

1.1 Motivation from Business

In their study, *Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing*, Alégroth et al. [AFK16] report that up to 60% of the effort planned for test automation is used for the maintenance of test scripts. There are multiple non-technical and technical factors that influence the cost of maintenance, such as developer knowledge and the stability of the test environment and simulators. Since Selenium is sensitive to changes

1 Wilhelm Büchner Hochschule, Fachbereich Informatik, Hilpertstr. 3, 64295 Darmstadt, Deutschland, daniel.keiss@gmail.com, https://orcid.org/0009-0009-8864-1040

in the user interface (UI), partially automated maintenance may reduce the cost to provide a return on investment (ROI).

Selenium uses Locators to identify elements on the web page, such as buttons, input fields, or links. Locators are defined by attributes like ID, name, class name, tag name, link text, and partial link text. A large part of the maintenance effort in Selenium tests results from failing locators, which are often broken when minor changes occur in the user interface. Slow response times in the frontend or changes in system messages can disrupt automated tests too [Kh23].

The argument that testability issues are more about poor application and architecture design than flaws in the testing approach is well-known. In many cases, recreating such applications, similar to rewriting the tests, would not be economically feasible. Additionally, it should be noted that applications generating dynamic content, such as content management systems (CMS), are inherently more challenging to test [Ch21].

In this context, self-healing tools can be highly beneficial. This tools aims to reduce the maintenance effort by automatically repairing failing Locators during test execution. Test reports show which Locators were healed, and developers can review and integrate them into the codebase. In addition, automatically adding these changes to the codebase, for example, by a pull request, can further streamline the maintenance process.

1.2 Objectives and Research Questions

The objectives and research questions of this paper are to determine whether a solution using a generic LLM for self-healing Selenium tests is comparable to even better than Healenium. This includes assessing whether the AICurator solution is as effective at healing tests during runtime as Healenium.

Large language models are known for their reasoning capabilities, which are mostly informal reasoning based on experience and common sense [HC23]. It is of interest if these reasoning capabilities can provide an advantage over Healenium's probability-based approach in our specific case. Would the reasoning improve the accuracy and adaptability of automated test healing? Does it potentially reduce manual effort more effectively than Healenium, for example, by automatically creating pull requests?

1.3 Structure of the Paper

Chapter 2 Terminology & Definitions provides an overview of the key terms used in this paper.

Chapter 3 Selenium & Healenium introduce the fundamentals of Selenium and Healenium. It describes which Selenium components we focus on, the architecture we use for the test setup, and how Healenium extends this to implement the self-healing mechanism.

Chapter 4 Cucumber presents Cucumber, Behaviour-Driven-Development and Gherkin. It explains how these concepts are used for frontend testing and how step reusability is applied to test healing capabilities.

Chapter 5 Large Language Models briefly explains what LLMs are. It describes the reasoning of LLMs and what it means in the context of this work. Following that, the LLM variants tested upfront are introduced, and why Claude3 was selected as the primary LLM.

Chapter 6 Prompt Definition explains how the prompt for healing the locators was created. It also details the structure of the parameters and how the prompt was eventually improved.

Chapter 7 Implementation describes how the demo web application and the LLM-based self-healing tool AiCurator solution was implemented.

Chapter 8 Experiments details the experiments conducted to compare Healenium and AiCurator.

Chapter 9 Summary presents the results of the experiments and discusses the findings.

2 Terminology & Definitions

This paper repeatedly uses terms from the fields of frontend testing with Selenium, test definition with Cucumber, and large language models. This chapter provides an overview defining the key terms used.

Selenium is an open-source framework for automating web browsers [Ga20b].

Locators are an abstraction used to identify web elements in Selenium. There are different strategies available, such as identifying by ID, by name, by link text, and so on [Ga20b].

Page Object Model (POM) is a design pattern that models web pages using an object-oriented class. These page objects contain all relevant locators in one place to ease test maintenance and reduce code duplication [Ga20b].

Healenium is a self-healing tool based on Selenium that aims to reduce maintenance effort by repairing broken locators during test execution [Ch21].

AICurator is a self-healing build for this paper based on a large language model.

Cucumber is an open-source testing framework that supports Behavior-Driven Development. It allows users to write automated tests in a human-readable language, using a Given-When-Then structure, defined in the Gherkin language [So].

Behavior-Driven Development (BDD) is a software methodology designed to improve collaboration between developers, testers, and business stakeholders. The purpose is to build a shared understanding of how the system should behave [SM23].

A **Large Language Model** (LLM) is a deep neural network designed for modeling and generating text. An LLM is able to generate coherent and contextual answers for a given task or question [ka23].

Reasoning, in general, is the process of thinking about something in a logical and systematic way, using evidence and experiences to reach a conclusion or make a decision. In this paper, in the context of LLM, reasoning always refers to informal reasoning, which is a less structured approach that relies on intuition, experience, and common sense to draw conclusions and solve problems [HC23].

3 Selenium & Healenium

As mentioned in the introduction, Selenium is a widely used open-source framework for automating tests of web interfaces. It supports all common browsers like Chrome, Firefox and others [Ga20b].

Selenium has several components like the WebDriver, SeleniumGrid and SeleniumIDE. We will focus on the WebDriver, since this component is used to instrument the browser and interact with the web page. The WebDriver communicates directly with the web browser using the W3C protocol. The communication happens via the JSON Wire Protocol, building the bridge between the Selenium API with the browser's automation capabilities. Each browser has its own WebDriver implementation, like chromedriver for Chrome and geckodriver for Firefox [Cob; Ga20b].

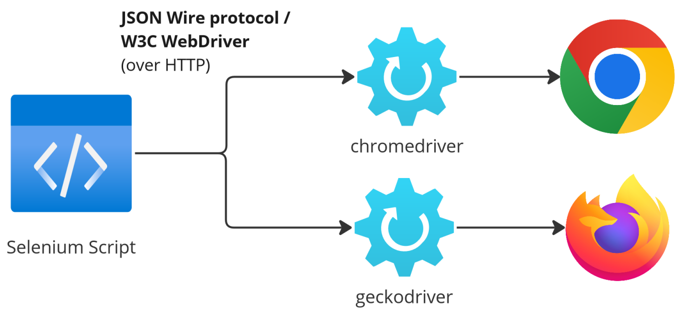


Fig. 1: Selenium WebDriver Architecture [Ga20b]

To interact with the web page, Selenium uses Locators. Locators identify elements on the web page, such as buttons, input fields, or links. The most commonly used locators are ID, name, class name, tag name, link text, or partial link text. ID and name are considered robust because they are unique and less likely to change. However, for various reasons, developers often have to use class names or XPath, which are more likely to change. This can increase the maintenance effort for the tests. Additionally, many pages use dynamic content and therefore additional checks must be implemented in Selenium, which can be a cause of unstable tests [Ga20b].

A common pattern to interact with the web page is the Page Object Model (POM). A POM contains all locators and methods to interact with the specific web page. Depending on the test architecture, POM can represent page fragments too [Vi16]. POM is an object-oriented approach to encapsulate pages or fragments. When locators are changing, only the POM needs to be updated and not the tests themselves [Ga20b; Le23].

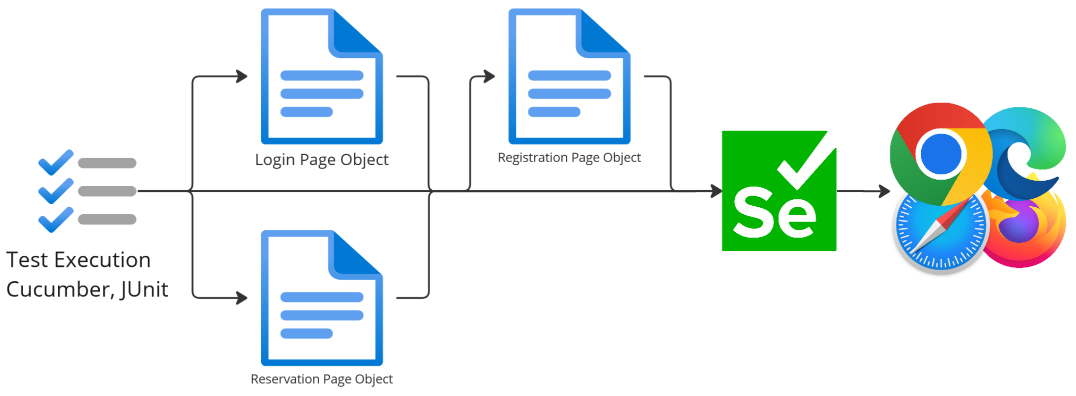


Fig. 2: Selenium PageObjects Architecture Example Project

3.1 Healenium

Healenium aims to address the challenge of broken locators during test execution and helps to reduce maintenance effort. When an automated test cannot find a web element, Healenium tries to find an alternative during runtime. Healenium uses a modified Longest Common Subsequence (LCS) algorithm with weighting factors to analyze the Document Object Model (DOM). It considers attributes like tags, IDs, classes, and values to find the best match for a missing element and generate a new locator. The locators are stored in a database and can be reviewed by the developer and used for manual code changes, for example, with help from the Healenium plugin for IntelliJ IDEA [Ch21]. Healenium does not provide a fully automatic repair, for example, creating a pull-request with the necessary changes.

How Healenium works [Ch21]:

1. Baseline Creation: During successful test runs, Healenium stores the web element locators as a reference for future tests in the database.
2. Exception Handling: If an element isn't found, Healenium catches the NoSuchElementException from Selenium and starts the self-healing process.
3. Self-Healing Process:
    - Compare the current DOM with the stored locator via modified LCS algorithm.
    - Generates a list of potential alternative locators with confidence scores.
    - Select the best match to replace the failed locator.
4. Test Continuation: The test proceeds by using the new locator if working.
5. Reporting: After the test run, Healenium provides a report of the healed locators.

4 Cucumber

Cucumber is an open-source testing framework that supports Behavior-Driven Development (BDD). It allows users to write automated tests in a human-readable language, using a Given-When-Then structure, defined in the Gherkin language. In Cucumber, Gherkin scenarios are stored in feature files. These files are split into scenarios, which consist of a series of step definitions. A step definition maps the Gherkin steps to the code that performs the actions in the system. With this approach, feature files serve as an executable specification for specific functionality within the application [So].

4.1 Behavior-Driven Development (BDD)

Behavior-Driven Development (BDD) is a software methodology designed to improve collaboration between developers, testers, and business stakeholders. The purpose is to build a shared understanding of how the system should behave. BDD using the Gherkin language to specify the behavior. The process follows a structured approach called Given-When-Then, where:

- Given describes the initial setup
- When outlines the action taken
- Then defines the expected outcome

[SM23].

```
Scenario: Login
  Given the login page is open
  When the user logs in
  Then the reservation page is shown
```

Fig. 3: Gherkin Scenario for Login

4.2 Cucumber for Frontend Testing with Healing

For frontend testing, Cucumber is used in this work with the following concept. The features contain scenarios that describe the expected behavior of the application. The steps contain the code to interact with the pages, create test data using the API, and maintain the context. The pages contain the locators and methods to interact with the web elements. The healing process occurs within the pages.

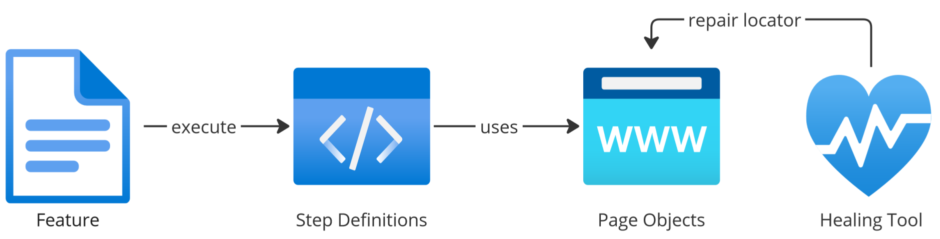


Fig. 4: Feature with Steps and Pages with Healing

4.3 Why choosing Cucumber

To test whether Selenium locators can be healed using LLMs, simpler JUnit tests could have been used. So, why was Cucumber chosen for this work instead?

The main reason is that step definitions in Cucumber are easily reusable. To compare the Healenium approach with the LLM approach, it simply needs to copy the scenarios and execute them with each method. In the test report, both results can be directly compared. Another reason is that step definitions describe the functional actions being performed. This provides the LLM a broader context for analysis.

5 Large Language Models

A Large Language Model (LLM) is a deep neural network designed for modeling and generating text. The most common LLMs, like ChatGPT, use the Transformer architecture with self-attention mechanisms. LLMs are pretrained on extensive text datasets like Wikipedia and arXiv to learn statistical patterns and representations of natural language. After pretraining, the model is fine-tuned on specific tasks or datasets to improve its performance. The model's knowledge is stored in its parameters. During text generation, the LLM uses the input prompt and the already generated text as part of its context. This mechanism enables the model to generate coherent and contextual relevant text. Modern LLMs consist of billions of parameters, with models like BERT Large containing 340 million parameters and GPT-3 containing 175 billion parameters [ka23]. Based on unofficial information, the next generation of LLMs like GPT 4 already uses 1.8 trillion parameters [ka23].

5.1 Reasoning

Reasoning is the process of thinking about something in a logical and systematic way, using evidence and experiences to reach a conclusion or make a decision There is no clear definition for large language models, what Reasoning exactly means. In many cases, it is an informal reasoning, which is a less structured approach that relies on intuition, experience, and common sense to draw conclusions and solve problems [HC23]. This informal reasoning is meant when we talk about reasoning in the context of this Paper.

The size of the model corresponds to its Reasoning capabilities. It is an inherent property of LLMs that depends on the training data [Ku24]. Based on a study that extracted the training data of GPT-3, it was found that at least part of the training data was the open-source dataset The Pile [An24]. The Pile contains a diverse set of data sources, including books, articles, and GitHub [Ga20a]. Statistics from GitHub as of October 8, 2024, show that there are 373,986 repositories for React projects (github.com/topics/react). For Selenium, there are 14,041 repositories (https://github.com/topics/selenium). Therefore, we can assume that Selenium tests for React frontends using HTML and JavaScript are well-represented in the training data. If we use a new test framework, the LLM might not have as broad a background, and its reasoning is likely to be worse with a higher chance of errors. For example, as of October 9, 2024, there are only 2,102 projects available for Playwright (github.com/topics/playwright).

In the context of this paper, there were only generic and not fine-tuned LLMs used. Since fine-tuning is work intensive, expensive to train, and multiple times more expensive in use than a generic models, it was out of scope for this paper [Di23].

5.2 Selection of the Large Language Model

For the implementation, three different LLMs were tested upfront. The idea was to test implementations for the LLM with the best coding capabilities, for the most widely used LLM, and for a local variant. At the time of this work, the best coding ability was attributed to Claude 3.5 Sonnet from Anthropic [Sa24]. The most widely used LLM is ChatGPT from OpenAI. Here the GPT 4o Mini was selected because it provides a structured output [Op]. The best locally runnable variant on the given machine was Mistral 8x7B [Ji24].

The experiments and final results are based on *claude-3-5-sonnet*. It was finally selected for the experiments because it is the fastest LLM of all tested, and it delivers the best coding results according to general tests [Mi]. In the following section, you will find a brief overview of advantages and disadvantages of the models in relation to each other as they pertain to this work.

5.2.1 Claude 3.5 Sonnet

The main Model used for all experiments in this paper is *claude-3-5-sonnet*. It is faster than OpenAI GPT 4o Mini and is expected to have better results in code generation [Sa24]. Unlike OpenAI, Claude 3.5 cannot use specific formats for the output. However, there was never a case, where the format was incorrect.

The temperature of the model controls the randomness of predictions during text generation. Higher temperatures lead to more creative outputs, while lower temperatures are more deterministic [Ana]. Since the goal is to heal locators, the temperature was set to 0.4 (range between 0 and 1), providing mostly straight forward answers with little creativity.

5.2.2 GPT 4o Mini

ChatGPT is one of the most known LLMs with broad support, e.g., as a service in Azure. In this work, the Model *gpt-4o-mini* was tested. A plus of this Model is that it can use defined formats for the return value, which is currently not possible with Claude 3. OpenAI guarantees that the answer is in the defined format [Op].

Like Claude 3.5 Sonnet the temperature was again set to 0.4. Some exploratory tests showed very good results in auto-healing. However, since Claude3 is faster, it was finally selected for the structured tests

5.2.3 Local Mixtral 8x7B

The last LLM tested in this paper is a locally hosted Mixtral 8x7B model. This model is probably the strongest for coding that can be run on the available local machine [Ji24]. The local machine using an NVIDIA RTX 3090 and simulated OpenAI API by LM-Studio (lmstudio.ai).

Unfortunately, it is too slow on the given machine to perform healing at runtime. Furthermore, it has issues returning responses in the defined JSON format. Nevertheless, the reasoning capabilities of the model allow it to generate healed locators in many cases, which is a good result given that Mixtral 8x7B uses only 14 billion active parameters [Ji24].

To develop a working prototype, the decision was made to stop the investigation at this point. If you wish to use a self-hosted solution, the open-source LLM variant would still be an option with more powerful hardware and enhanced response extraction.

6 Prompt Definition

To heal the locator, a suitable prompt is required. Following the guidelines from the Anthropic prompt engineering, first a clear definition of the success criteria for the success case is necessary. In our case, the most important success criteria is that the test runs successfully with the new locator [Anb]. First, an own prompt was created and further developed based on the guidelines. This prompt was then entered into Anthropic prompt generator and expanded automatically.

6.1 Prompt Parameters

Before we proceed to the prompt definition, it is first necessary to determine which data are available for repairing failed locators. When Selenium cannot find an element, it throws an ‘ElementNotFoundException‘. This exception contains the failed locator and is intercepted before the prompt. Additionally, we have access to the body of the current page, which can be loaded via the WebDriver. To provide better context for the prompt and enable automatic code repair, the stack trace is analyzed. From it, we can deduce which ‘PageObject‘ class contains the incorrect locator and by whom it was invoked. In our case, the ‘PageObject‘ class is called by a ‘Step‘ class. The methods involved represent the functional context of the test. Finally, the format is defined. This must necessarily be specified in the prompt only if no format can be set beforehand, as is the case, for example, with ChatGPT 4.0 mini.

6.2 First Prompt

The experiments started with this Prompt, and it already generated perfect results. However, since the number of experiments is rather limited and some best practices of prompt engineering are not covered in the first prompt, the prompt was reengineered.

Parameter	Description
seleniumExceptionMessage	The error message from the Selenium Element Not Found Exception. This message contains the failed locator.
htmlBody	The HTML body from the page under test.
pageSourceCode	The code of the page object.
callMethodPage	The method in the page object.
stepsSourceCode	The code of the step class to give more context about the purpose from the test.
callStepMethod	The method in the step class.
format	The JSON format of the return value given by SpringAI.

Tab. 1: Description of Parameters

```
Your task is to resolve an "element not found" exception in \Gls{Selenium} by providing a working locator.
The expection is {exception}.
You will be successful when the test runs successfully with this locator.
To find the locator, you will examine the HTML body {htmlBody}, the \Gls{Selenium} page object's code {
     pageSourceCode} was called in method {callMethodPage}.
The Cucumber step code {stepsSourceCode} that calls the \Gls{Selenium} page object from method {callStepMethod}
     gives more inside to understand the test context.
Provide a locator that is unique and stable in the attribute "locator".
Update the page source code with the changed locator for merging in the attribute "pageSourceCode".
Provide a concise explanation of the changes made to the source code and the locator in the attribute "explanation".

{format}
```

6.3 Prompt created with Anthropic Prompt Generator

Studies have demonstrated that prompt engineering enables models to better understand the context and nuances of the tasks they are given. Using clear instructions and role-prompting, combined with advanced techniques like chain-of-thought and self-consistency, significantly enhances LLM capabilities. A simple example of this is giving an LLM time to think to provide a better answer. This can be achieved by breaking the task into steps and defining what is expected at each step [Ch24].

Since Anthropic offers a prompt generator that applies the well-known best practices of prompt engineering, it was used for the first iteration to improve the existing prompt The Anthropic Prompt Generator enhances the first prompt a lot, for example, by providing information to attributes that are unlike to change for locator. Furthermore, the extendability of this prompt is much better.

In detail, these are the improvements in comparison from the first prompt based was identified. Each technique is linked to the Anthropic guidelines. You will find the similar descriptions like role prompting or using prompting chains in this study.

Technique	Description
Role Prompting	To set a role, the prompt starts with: "You are a QA automation expert. . . "
Clear and specific instructions	Specify the goal with the following command: "Your goal is to analyze the given information, create a unique and stable locator, and update the page source code accordingly."
Chain complex prompts	Provide six steps to ensure focus on each step, starting with: "1. Analyze the exception message to understand which element is not being found."
Using structured format for input	Use XML tags to define input data.
Explicit output instructions	Provide output instructions in one section. This was adapted from the original Anthropic Prompt Generator response since SpringAI provides this in the format parameter.

Tab. 2: Prompt Engineering Techniques used by the Anthropic Prompt Generator

The experiments showed that the output for the locator had to be refined to ensure it could be parsed into a By object at runtime. This already demonstrated the benefit of the extensibility of the generated prompts. The following prompt was used in the experiments.

```
You are a QA automation expert tasked with resolving a \Gls{Selenium} "element not found" exception by providing a
     working locator. Your goal is to analyze the given information, create a unique and stable locator, and update
      the page source code accordingly.

Here's the \Gls{Selenium} exception message you need to resolve:
<seleniumExceptionMessage>
{seleniumExceptionMessage}
</seleniumExceptionMessage>

To find the locator, you will examine the following HTML body:
<htmlBody>
{htmlBody}
</htmlBody>

The \Gls{Selenium} page object's code that was called in method {callMethodPage} is:
<pageSourceCode>
{pageSourceCode}
</pageSourceCode>

The Cucumber step code that calls the \Gls{Selenium} page object from method {callStepMethod} is:
<stepsSourceCode>
{stepsSourceCode}
</stepsSourceCode>

Follow these steps to resolve the issue:

1. Analyze the exception message to understand which element is not being found.
2. Examine the HTML body to locate the element in question.
3. Review the page source code and Cucumber step code to understand the context of the test.
4. Create a new, unique, and stable locator for the element. Consider using ID, name, or a combination of attributes
      that are unlikely to change.
5. Update the page source code with the new locator. Complex locators should use XPath.
6. Provide a brief explanation of the changes made and why the new locator is more reliable.

Return the following attributes in the response:
1. "explanation": A brief explanation of the changes and reasoning
2. "locator": The new, working locator you've created. Use the format "By.<locatorType>(<locatorValue>)".
3. "pageSourceCode": The updated page source code with the new locator

{format}
```

7 Implementation

To compare the AI approach based on AICurator with the Healenium approach, a sample web application was created. The application provides login, registration and reservation processes. For each of these processes, different version of the frontend page where implemented. The test for each process only covers the first version of the page, the other versions required healing. With this approach, the healing process is repeatable and AICurator and Healenium can be compared to each other.

7.1 Design Demo Web Application

The web application allows users to share a taxi for a specific route. It simulates processes like determining the route, validating it, and calculating prices for the reservations. A key feature is that the cost of a reservation decreases as more users join the ride. The users get updates when the cost changes. The login and registration functionalities are fully implemented since they are easily to implement and serve as useful examples.

The application follows a single-page application (SPA) architecture for the frontend and a RESTful backend. This separation allows that the API used by the frontend can be applied in testing scenarios, such as creating a user via the API. The architecture with SPA fronend combined with a RESTful backend mirrors a dominant pattern in recent years [Mu]. This should allow adopting the experimental results better to real-world scenarios.

The frontend is built using React, with content dynamically loaded. For the design, Bootstrap is used. React was chosen because it allows multiple versions of the same page to be implemented without duplicating logic. This is particularly useful when, for instance, modifications to the reservation logic need to be applied without having to update a second version manually.

The backend is developed using Spring Boot and includes an OAuth server for login. The APIs provide functionalities for registration and reservation. The interfaces can be used by test, which allows it to create test data directly and not via the frontend.

To illustrate the interaction between the frontend and backend, here is a description of how the login and taxi search processes work:

1. The frontend sends the login data to the backend.
2. The backend processes the login data, creates a JSON Web Token (JWT), and returns it to the frontend.
3. The frontend sends a request to search for taxis, including the JWT received from the login. The backend validates the JWT to ensure the user is authenticated.
4. The backend searches for available taxis and returns the results to the frontend.

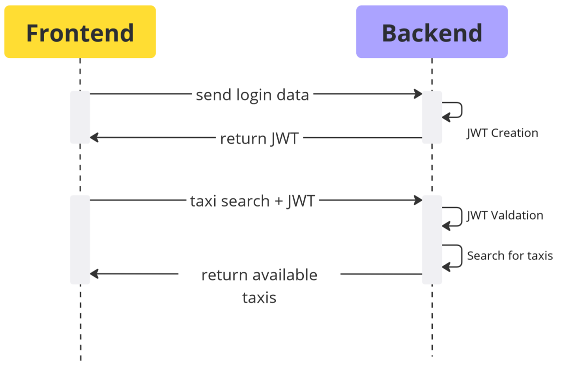
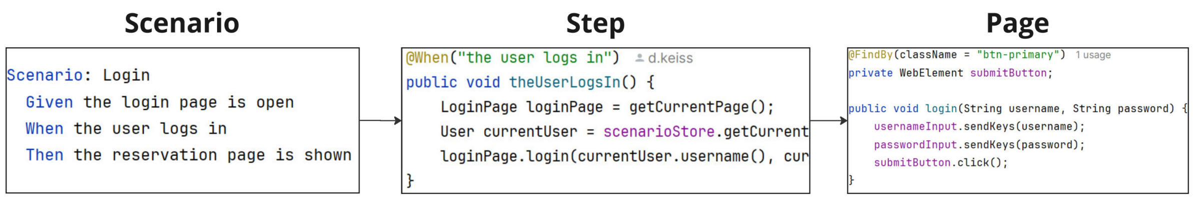


Fig. 5: Taxi Share Login and Reservation

7.2 Design Test Project

The tests were created using Cucumber and are divided into steps and pages. The Cucumber features describe the expected behavior of the application. The steps contain the code to interact with the pages, create test data using the API, and maintain the context. The pages contain the locators and methods to interact with the web elements. Runtime healing occurs within the pages.



```
Scenario: Login
  Given the login page is open
  When the user logs in
  Then the reservation page is shown
```

```
@When("the user logs in")  d.keiss
public void theUserLogsIn() {
    LoginPage loginPage = getCurrentPage();
    User currentUser = scenarioStore.getCurrent
    loginPage.login(currentUser.username(), cur
}
```

```
@FindBy(className = "btn-primary")  1 usage
private WebElement submitButton;

public void login(String username, String password) {
    usernameInput.sendKeys(username);
    passwordInput.sendKeys(password);
    submitButton.click();
}
```

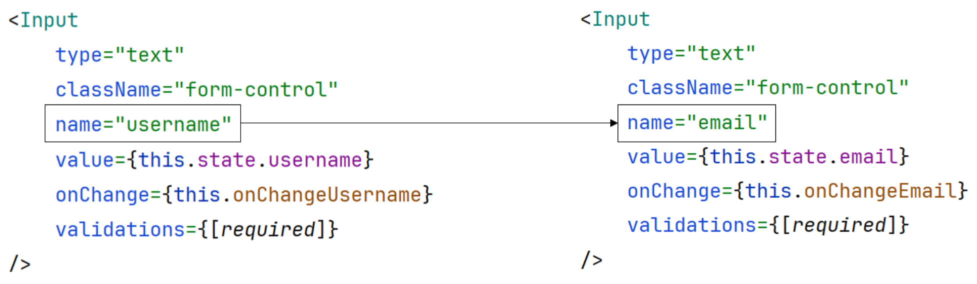
Fig. 6: Scenario, Step, and Page for Login

7.3 AICurator

The custom solution based on an LLM named AICurator shares the same primary objective as Healenium: If a locator cannot be found, a healing should be done at runtime. To achieve this, like in Healenium, the Selenium exception is caught, and the healing process is initiated.

During the healing process, information for the prompt is collected first. This includes the failed locator, which is identified in the exception, as well as the body of the current HTML page. This information are the base, to find an alternative locator. The base information are supplemented with the code of the PageObject and the called methods from the test code. The purpose of this information is to provide the LLM an extended context and the capability to generate a pull request for the repaired test code.

The following example of the AICurator healing process illustrates a scenario in which the element's name attribute changed from "username" to "email". The test can remain semantically the same, for example, if the username is identical to the email address. In this situation, the web page was adjusted without the test being updated accordingly. Such changes would usually occur during the development process, and this example demonstrates a meaningful update in the test. Since we have only two elements, it made sense at this point to change the locator to the email attribute, as it is the closest match semantically. This example also highlights that changes from the automatic healing process should always be reviewed thoroughly before being applied.



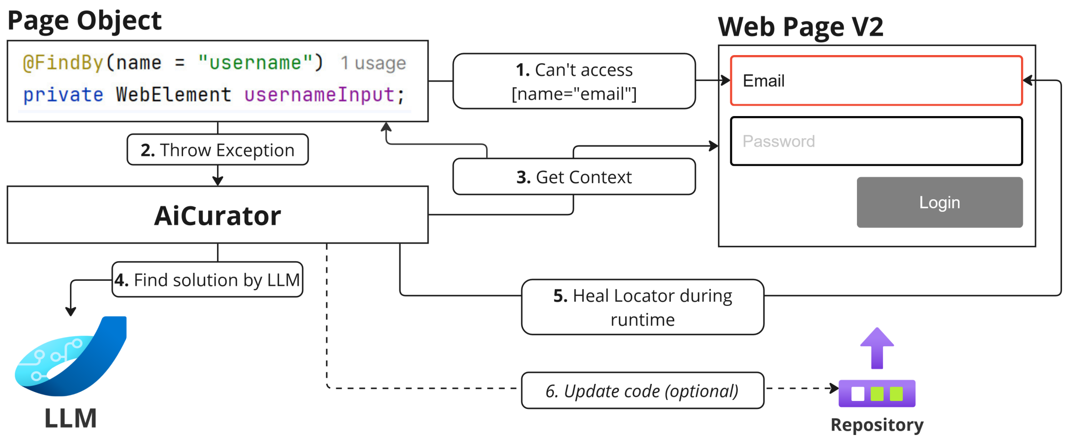
```
<Input
    type="text"
    className="form-control"
    name="username"
    value={this.state.username}
    onChange={this.onChangeUsername}
    validations={[required]}
/>
```

```
<Input
    type="text"
    className="form-control"
    name="email"
    value={this.state.email}
    onChange={this.onChangeEmail}
    validations={[required]}
/>
```

Fig. 7: Name attribute changed from username to email

Step-by-Step example

1. Can't Access email: The test execution starts, but the web page throws an error because it cannot access the element identified by email.
2. Catch Exception: When the web element (in this case, the email input field) cannot be accessed, AiCurator catches the exception in the code, typically due to a locator failure.
3. Get Context: AiCurator retrieve the body of the web page, the code of the PageObject and the called methods from the test code to get a context for a proper healing and the capability to generate a pull request for the repaired test code.
4. Find Solution by LLM: AiCurator consults the large language model (LLM) to find a solution based on the page body and the error. The LLM is used to reason about the structure of the page and suggest an alternative locator or solution.
5. Heal During Runtime: AiCurator applies the solution found by the LLM in real time, effectively "healing" the page by dynamically adjusting the locator or method of interacting with the web element.
6. Pull Request (optional): After successful healing, AiCurator sends an event with the updated test code, which the test project can use, for example, to create a pull request for its repository.



```
@FindBy(name = "username")  1 usage
private WebElement usernameInput;
```

Fig. 8: AICurator flow when a web element is not found

7.4 Realisation

The example project can be found here: `https://github.com/dkeiss/aicurator`. This project includes the demo application with a React frontend and a Spring Boot backend, as well as the self-healing Cucumber test with the AICurator framework.

8 Experiments

To compare AICurator in realistic scenarios with Healenium, three Cucumber features were created for registration, login, and reservation. These features don't contain error cases to keep clarity. The features automatically execute the scenarios from top to bottom. For this purpose, corresponding Cucumber test steps were implemented using Selenium.

8.1 Test concept

The concept of repeatable and repairable test cases is as follows:

- Each feature initially includes a scenario that does not require locator healing (V1).
- This scenario is then copied and adjusted to open a different version of the web page (V2).
  - Additionally, the login feature includes additional version to test further specific cases (V3,V4).
- All versions after V1 required repair:
  - For this, the generated prompt from Chapter 6 is used in AICurator. See "Prompt created with Anthropic Prompt Generator" for more details.
  - For Healenium, one successful test run (V1) is needed, so that the locator is stored in the database.

The type of frontend changes that require healing are:

- Fields have been replaced while keeping semantic equivalence.
- The attributes of the fields have been modified.
- An additional field is added.
- The layout has been adjusted, which does not require locator changes but might increases the complexity for LLM processing, as the entire page must be analyzed.

8.2 Execution

By default, tests are executed with AICurator. For experimentation, the tests were run multiple times in a row, as LLMs may produce varied outputs. Mostly, the outputs are consistent and used as the presented result.

To execute the features with Healenium, the VM parameter `-Dhealing.approach=healenium` must be set. This was done once since Healenium exhibits consistent behavior. A prerequisite for Healenium testing is a prior successful test to store the working locators in the Healenium database.

The subsequent sections describe the features in detail. The final results documented here are from November 18, 2024.

8.3 Registration Feature

The registration feature tests the happy path of user registration. It includes one scenario, which was copied and adapted to require new locators. The new version use a modified layout.

This screenshot illustrate the changes in the registration user interface.

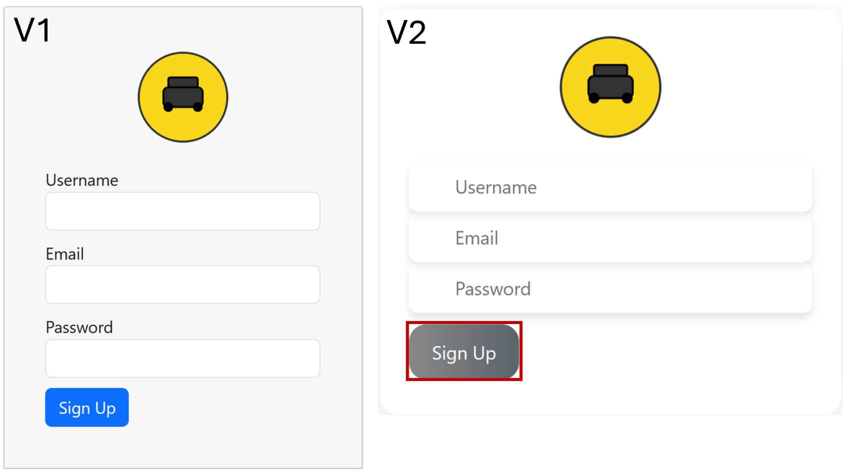


Fig. 9: Changes in Registration UI

The following registration feature was executed by Cucumber. It ran the automated test using Selenium against the different versions of the registration UI.

```
Feature: Registration
  As a user
  I would like to register at the website,
  to make reservations for shared taxis.

  Background:
    Given self-healing for locators is enabled

  Scenario: Registration (V1)
    # This scenario doesn't need self-healing
    Given the registration page is open
    When the user registers
    Then the registration is successful

  Scenario: Registration (V2)
    # This scenario need self-healing
    # 1. The style is changed
    # 2. The button uses different classes
    Given the registration page v2 is open
    When the user registers
    Then the registration is successful
```

List. 1: Registration Feature

8.3.1 Results

In the registration feature, both AICurator and Healenium were able to heal all tests. The solutions are different, but neither seems to be better than the other, as the class `fancy-button` is the only attribute of the button.

Scenario	**Changes**	**Result AICurator**	**Result Healenium**
Registration (V1)	Original Scenario	Test successful	Test successful

Registration (V2)	1. The style is changed
	2. The button uses different classes
		Healing and Test successful
		Changed locator from:
		className = `"btn-primary"` to
		cssSelector = `"button.fancy-button"`
			Healing and Test successful
			Changed locator from:
			className = `"btn-primary"` to
			cssSelector = `"button"`

Tab. 3: Registration UI changes and healing results

8.4 Login Feature

The login feature tests the happy path of the user login. It includes one scenario adapted for four different webpage versions. Versions V1 and V2 share the same layout but use different fields that need to be treated semantically equivalently. Versions V3 and V4 use a new layout and contain additional IDs for the buttons. These IDs should be used if possible. In one case, it was used to fail Healenium on purpose and validate if the LLM finds a better solution here.

This screenshot illustrate the changes in the login user interface.

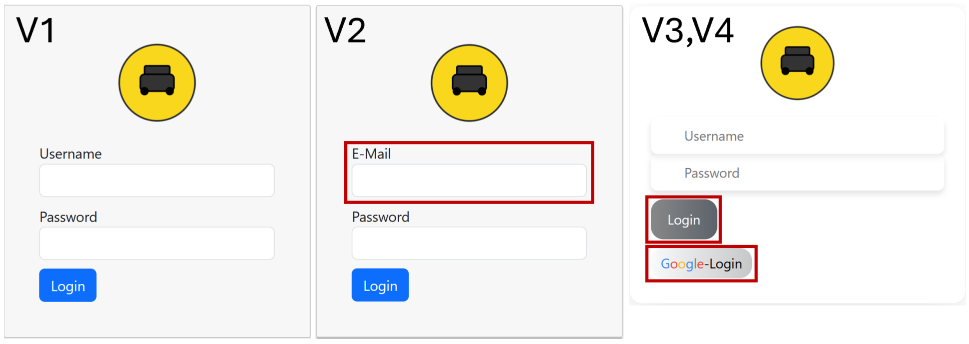


Fig. 10: Changes in Login UI

The following login feature was executed by Cucumber. It ran the automated test using Selenium against the different versions of the login UI.

```
Feature: Login
  As a user
  I would like to log in with an existing account,
  to make reservations for shared taxis.

  Background:
    Given self-healing for locators is enabled
    And a new registered user
    And a new browser session

  Scenario: Login (V1)
    # This scenario doesn't need self-healing
    Given the login page is open
    When the user logs in
    Then the reservation page is shown

  Scenario: Login (V2)
    # This scenario need self-healing
    # 1. The username field name is changed to email
    Given the login page v2 is open
    When the user logs in with email
    Then the reservation page is shown

  Scenario: Login (V3)
    # This scenario need self-healing
    # 1. The style is changed
    # 2. Login button uses a different class and has an ID which should be used
    # 3. Google login button with ID is added
    Given the login page v3 is open
    When the user logs in with email
    Then the reservation page is shown

  Scenario: Login (V4)
    # This scenario need self-healing
    # 1. The style is changed
    # 2. Login button uses a different class and has an ID which should be used
    # 3. Google login button without ID is added
    # This is an adjusted scenario designed to fail Healenium
    # It will give the Google button a higher rank because of the missing ID
    Given the login page v4 is open
    When the user logs in with email
    Then the reservation page is shown
```

List. 2: Login Feature

8.4.1 Results

In the login feature, both AICurator and Healenium were able to heal all regular tests.

The final scenario was intentionally adjusted to fail for Healenium. The scenario has the inconsistency that only the login button contains an ID. This inconsistency led the heuristic to select `cssSelector`-based as closer solution because the last successful locator used the type className. Furthermore, the solution from Healenium, `cssSelector=button#login-button`, is not optimal because, in this case, only the ID is required.

Scenario	Changes	Result AICurator	Result Healenium
Login (V1)	Original Scenario	Test successful	Test successful

Login (V2)	1. The username field name is changed to email	Healing and Test successful
		Changed locator from:
		name = `"username"` to
		name = `"email"`
			Healing and Test successful
			Changed locator from:
			name = `"username"` to
			name = `"email"`

Login (V3)	1. The style is changed
	2. Login button uses a different class and has an ID which should be used
	3. Google login button with ID is added
		Healing and Test successful
		Changed locator from:
		className = `"btn-primary"` to
		id = `"login-button"`
			Healing and Test successful
			Changed locator from:
			className = `"btn-primary"` to
			cssSelector = `"button#login-button"`

Login (V4)	1. The style is changed
	2. Login button uses a different class and has an ID which should be used
	3. Google login button without ID is added
	This is an adjusted scenario designed to fail Healenium.
	It will give the Google button a higher rank because of the missing ID, similar to the last successful locator.
		Healing and Test successful
		Changed locator from:
		className = `"btn-primary"` to
		id = `"login-button"`
			Healing and Test failed
			className = `"btn-primary"` to
			cssSelector =
			`"button.fancy-button-google"`

Tab. 4: Login UI changes and healing results

8.5 Reservation Feature

The reservation feature covers searching for taxi connections, making reservations when no connection is found, and joining rides. The challenge in these tests lies in the changing visibility of the elements. Additionally, the healing requires complex locators that combine type and name attributes.

This screenshot illustrate the changes in the reservation user interface.

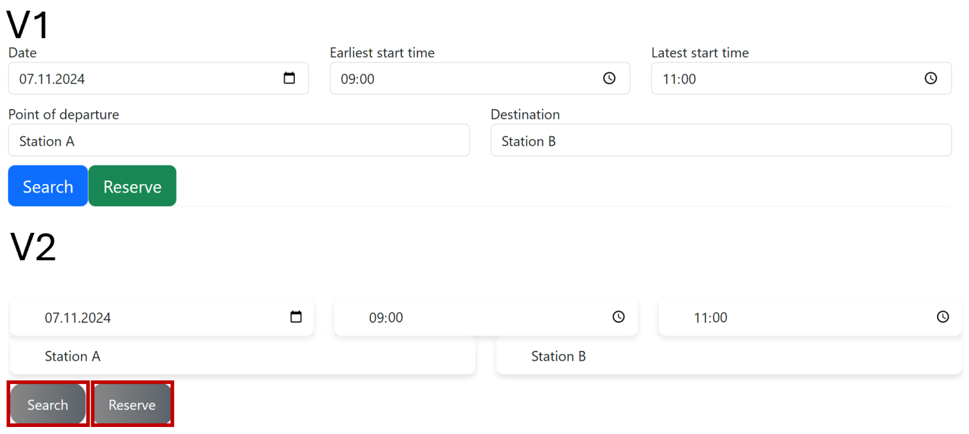


Fig. 11: Changes in Reservation UI

The following reservation feature was executed by Cucumber. It ran the automated test using Selenium against the different versions of the reservation UI.

```
Feature: Reservation
  As a logged-in user,
  I would like to be able to make reservations for collective tickets on certain routes during specific
       time periods
  in order to take advantage of any discounts.

  Background:
    Given self-healing for locators is enabled
    And a logged-in customer

  Scenario: Made a reservation and get updates (V1)
    # This scenario doesn't need self-healing
    Given no reservations exist
    And the reservation page is shown
    When searching for a taxi
      | departure | destination | date       | earliestStartTime | latestStartTime |
      | Station A | Station B   | 07.11.2024 | 09:00             | 11:00           |
    Then no taxis are available
    And the reserve option is visible
    When a reservation is made
    Then the reservation is shown in the list
    And the join option is invisible
    When somebody joins the reservation
    Then a join notification is shown

  Scenario: Made a reservation and get updates (V2)
    # This scenario needs self-healing
    # 1. The ID for the search button is no longer available; only type and name are present.
    # 2. The ID for the visible reserve button is no longer available; only type and name are present.
    Given no reservations exist
    And the reservation page v2 is shown
    When searching for a taxi
      | departure | destination | date       | earliestStartTime | latestStartTime |
      | Station A | Station B   | 07.11.2024 | 09:00             | 11:00           |
    Then no taxis are available
    And the reserve option is visible
    When a reservation is made
    Then the reservation is shown in the list
    And the join option is invisible
    When somebody joins the reservation
    Then a join notification is shown

  Scenario: Search for available taxis and join (V1)
    # This scenario doesn't need self-healing
    Given no reservations exist
    And there are available reservations from another user
      | departure | destination | date       | startTime |
      | Station A | Station B   | 2024-11-07 | 10:00     |
    And the reservation page v2 is shown
    When searching for a taxi
      | departure | destination | date       | earliestStartTime | latestStartTime |
      | Station A | Station B   | 07.11.2024 | 09:00             | 11:00           |
    Then taxis are available
    And the reserve option is invisible
    When joining the first reservation
    Then the current user is added to the participants

  Scenario: Search for available taxis and join (V2)
    # This scenario needs self-healing
    # 1. The ID for the search button is no longer available; only type and name are present.
    # 2. The ID for the invisible reserve button is no longer available; only type and name are present
         .
    Given no reservations exist
    And there are available reservations from another user
```

```
      | departure | destination | date       | startTime |
      | Station A | Station B   | 2024-11-07 | 10:00     |
    And the reservation page v2 is shown
    When searching for a taxi
      | departure | destination | date       | earliestStartTime | latestStartTime |
      | Station A | Station B   | 07.11.2024 | 09:00             | 11:00           |
    Then taxis are available
    And the reserve option is invisible
    When joining the first reservation
    Then the current user is added to the participants
```

List. 3: Reservation Feature

8.5.1 Results

In the reservation feature, both AICurator and Healenium were able to heal all tests. The challenge lies is that the reservation button and the search button are visible at different times. In a few cases, AICurator generated the selector `cssSelector = "button.btn.btn-success.btn-lg.fancy-button.hidden"` as a solution. This made the test run, but it was incorrect. In approximately 80% of the cases, the noted correct solution was generated. Here, adjustments will be required for future applications.

Healenium technically healed the test cases, but the created locator is incorrect like the failed on from AICurator. Here we see the limitation that Healenium can't create combined locators.

Scenario	**Changes**	**Result AICurator**	**Result Healenium**
Made a reservation and get updates (V1)	Original Scenario	Test successful	Test successful

Made a reservation and get updates (V2)	1. The ID for the search button is no longer available; only type and name are present.
	2. The ID for the visible reserve button is no longer available; only type and name are present.
		Healing and Test successful
		Changed locator from:
		id = `"searchButton"` to
		xpath = `"//button[@type='submit' and text()='Search']"`
		Changed locator from:
		id = `"reserveButton"` to
		xpath = `"//button[@type='submit' and text()='Reserve']"`
			Healing and Test technical successful
			Changed locator from:
			id = `"searchButton"` to
			cssSelector = `"button.fancy-button.btn-primary.btn-lg.btn"`
			Changed locator from:
			id = `"reserveButton"` to
			cssSelector = `"button.fancy-button.btn-primary.btn-lg.btn"`

Search for available taxis and join (V1)	Original Scenario	Test successful	Test successful

Search for available taxis and join (V2)	1. The ID for the search button is no longer available; only type and name are present.
	2. The ID for the invisible reserve button is no longer available; only type and name are present.
		Healing and Test successful
		Changed locator from:
		id = `"searchButton"` to
		xpath = `"//button[@type='submit' and text()='Search']"`
		Changed locator from:
		id = `"reserveButton"` to
		xpath = `"//button[@type='submit' and text()='Reserve']"`
			Healing and Test technical successful
			Changed locator from:
			id = `"searchButton"` to
			cssSelector = `"button.fancy-button.btn-primary.btn-lg.btn"`
			Changed locator from:
			id = `"reserveButton"` to
			cssSelector = `"button.fancy-button.btn-primary.btn-lg.btn"`

Tab. 5: Reservation UI changes and healing results

9 Summary

Selenium is a widely used framework for regression testing of web interfaces. Since many companies have a large inventory of tests, they face the challenge of investing a significant amount of time in the maintenance of their test automation. One of the challenges is broken locators. The self-healing tool Healenium tries to repair these broken locators automatically by using heuristics. To test if it is possible to heal locators using an LLM as well, this work developed a tool named AICurator and compared it with Healenium.

AICurator uses an engineered prompt and is flexible in the selection of the LLM. Based on coding capabilities, Claude3 Sonnet was chosen as the LLM for structured experiments to evaluate healing performance. The experiments were conducted in a newly built test application to ensure full control over all aspects of the environment. The scenarios in the experiments covered changes in the UI that led to locator failures and reflected practical experience. Both solutions, AICurator and Healenium, were able to address these changes, but each has its own advantages and disadvantages.

9.1 Advantages & Disadvantages

The advantages of Healenium are that it is faster and does not incur costs associated with API usage. Apart from a manipulated test, all tests were successfully healed. However, some of the healed locators, while technically functional, do not align with the intended behavior. This would raise problems when writing additional tests.

AICurator has the advantage of generating better locators than Healenium. It was able to heal all test cases, and the locators cover the intended behavior. Additionally, the solution can generate a complete pull request with the necessary updates. One significant drawback is that the results from AICurator can vary. Therefore, code changes made by AICurator must be reviewed by a human.

AICurator Cost

During the entire development period, 874,571 tokens were used as input, resulting in costs of €6.57. After a response from Claude 3.5 Sonnet, the prompt for healing the reservation contains approximately 3,120 tokens. This means each prompt incurs costs of about €0.02. Identical prompts are already cached by AICurator, which was disabled in the experiments to analyze the variance in results.

9.2 Reflection

A limitation in the representativeness of the results is that LLMs can produce different outputs for the same input. Additionally, online services are continuously updated, sometimes even within the same major version.

This study tested only a few cases, whereas real-world applications involve more complex scenarios. The true value of this approach with AICurator can only be assessed in a real-world project. However, a real project might have restricted the possibility of releasing the solution as open-source.

An exploratory approach was chosen to develop and validate the framework. The test cases were designed based on experience. A more structured approach, focusing on the most common known issues in Selenium tests, would likely have improved the framework's transferability to practical applications.

9.3 Outlook

Currently, AICurator is a prototype and has not yet been used in a real project. The next step will be to integrate it into real projects and evaluate whether the benefits outweigh the costs. Further tuning of the solution will likely be necessary in project usage. Additionally, the solution can be extended for project-specific requirements. Depending on the project, the generation of pull requests could be particularly beneficial. It may also be possible to apply the AICurator approach to other UI testing frameworks, such as Playwright.

A Used AI tools

The following AI tools were used for this paper and the accompanying demo project.

Tool	Usage
OpenAI ChatGPT GPT-4 `https://openai.com/index/gpt-4`	Grammatical corrections in the text. Generation of CSS classes and React components in the demo project.
GitHub Copilot `https://github.com/features/copilot`	Assistance with LaTeX code for the Paper. Auto-completion of Java and React code in the demo project.
Anthropic Claude 3.5 Sonnet `https://www.anthropic.com/news/claude-3-5-sonnet`	Enhancement of the initial prompt as described in Chapter Four.

Tab. 6: Used AI tools

References

[AFK16] Alégroth, E.; Feldt, R.; Kolström, P.: Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. Information and Software Technology 73, pp. 66–80, 2016, ISSN: 0950-5849, DOI: 10.1016/j.infsof.2016.01.012, URL: https://www.sciencedirect.com/science/article/pii/S0950584916300118, visited on: 09/29/2024.

[Ana] Anthropic: Glossary, Anthropic, URL: https://docs.anthropic.com/en/docs/resources/glossary, visited on: 09/24/2024.

[Anb] Anthropic: Prompt engineering overview, Anthropic, URL: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview, visited on: 09/24/2024.

[An24] Anonymous: Scalable Extraction of Training Data from Aligned, Production Language Models. In: Submitted to The Thirteenth International Conference on Learning Representations. under review, 2024, URL: https://openreview.net/forum?id=vjel3nWP2a.

[Ch21] Chernyshova, A.: Healenium: Self-Healing Library for Selenium Test Automation, Geek Culture, 2021, URL: https://medium.com/geekculture/healenium-self-healing-library-for-selenium-test-automation-26c2358629c5, visited on: 09/24/2024.

[Ch24] Chen, B.; Zhang, Z.; Langrené, N.; Zhu, S.: Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review, 2024, arXiv: 2310.14735 [cs.CL], URL: https://arxiv.org/abs/2310.14735, visited on: 10/10/2024.

[Coa] Conservanc, S. F.: History, Selenium, URL: https://www.selenium.dev/history/, visited on: 09/09/2024.

[Cob] Consortium, W. W. W.: WebDriver, URL: https://www.w3.org/TR/webdriver/, visited on: 09/09/2024.

[Di23] Dickson, B.: The implications of ChatGPT's new fine-tuning feature - TechTalks, 2023, URL: https://bdtechtalks.com/2023/08/28/openai-chatgpt-fine-tuning/, visited on: 09/24/2024.

[Ga20a] Gao, L.; Biderman, S.; Black, S.; Golding, L.; Hoppe, T.; Foster, C.; Phang, J.; He, H.; Thite, A.; Nabeshima, N.; Presser, S.; Leahy, C.: The Pile: An 800GB Dataset of Diverse Text for Language Modeling, 2020, arXiv: 2101.00027 [cs.CL], URL: https://arxiv.org/abs/2101.00027, visited on: 10/08/2024.

[Ga20b] García, B.; Gallego, M.; Gortázar, F.; Munoz-Organero, M.: A Survey of the Selenium Ecosystem. Electronics 9 (7), Number: 7 Publisher: Multidisciplinary Digital Publishing Institute, p. 1067, 2020, ISSN: 2079-9292, DOI: 10.3390/electronics9071067, URL: https://www.mdpi.com/2079-9292/9/7/1067, visited on: 08/19/2024.

[HC23] Huang, J.; Chang, K. C.-C.: Towards Reasoning in Large Language Models: A Survey. In (Rogers, A.; Boyd-Graber, J.; Okazaki, N., eds.): Findings of the Association for Computational Linguistics: ACL 2023. Association for Computational Linguistics, Toronto, Canada, pp. 1049–1065, 2023, DOI: 10.18653/v1/2023.findings-acl.67, URL: https://aclanthology.org/2023.findings-acl.67.

[Ji24] Jiang, A. Q.; Sablayrolles, A.; Roux, A.; Mensch, A.; Savary, B.; Bamford, C.; Chaplot, D. S.; de las Casas, D.; Hanna, E. B.; Bressand, F.; Lengyel, G.; Bour, G.; Lample, G.; Lavaud, L. R.; Saulnier, L.; Lachaux, M.-A.; Stock, P.; Subramanian, S.; Yang, S.; Antoniak, S.; Scao, T. L.; Gervet, T.; Lavril, T.; Wang, T.; Lacroix, T.; Sayed, W. E.: Mixtral of Experts, 2024, arXiv: 2401.04088 [cs.LG], URL: https://arxiv.org/abs/2401.04088, visited on: 09/25/2024.

[ka23] katerinaptrv: GPT4- All Details Leaked, Medium, 2023, URL: https://medium.com/@daniellefranca96/gpt4-all-details-leaked-48fa20f9a4a, visited on: 09/25/2024.

[Kh23] Khankhoje, R.: Effortless Test Maintenance: A Critical Review of Self-Healing Frameworks. International Journal for Research in Applied Science and Engineering Technology 11 (10), pp. 826–834, 2023, ISSN: 23219653, DOI: 10.22214/ijraset.2023.56048, URL: https://www.ijraset.com/best-journal/effortless-test-maintenance-a-critical-review-of-self-healing-frameworks, visited on: 07/17/2024.

[Ku24] Kumar, P.: Large language models (LLMs): survey, technical frameworks, and future challenges. Artificial Intelligence Review 57 (10), p. 260, 2024, ISSN: 1573-7462, DOI: 10.1007/s10462-024-10888-y, URL: https://doi.org/10.1007/s10462-024-10888-y, visited on: 11/30/2024.

[Le23] Leotta, M.; García, B.; Ricca, F.; Whitehead, J.: Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. In: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). ISSN: 2159-4848, pp. 339–350, 2023, DOI: 10.1109/ICST57152.2023.00039, URL: https://ieeexplore.ieee.org/abstract/document/10132210, visited on: 08/19/2024.

[Mi] Minyang Tian1, e. a.: SciCode: A Research Coding Benchmark Curated by Scientists, URL: https://arxiv.org/html/2407.13168v1, visited on: 09/24/2024.

[Mu] Muldoon, C.; Görgü, L.; O'Sullivan, J. J.; Meijer, W. G.; Masterson, B.; O'Hare, G. M. P.: Engineering testable and maintainable software with Spring Boot and React. URL: https://www.authorea.com/doi/full/10.36227/techrxiv.15147723.v3?commit=c212148d6cddf4cb7909fc2bf8915cc4eaec3d35, visited on: 10/14/2024.

[Op] OpenAI: OpenAI Platform, URL: https://platform.openai.com, visited on: 09/29/2024.

[Sa24] Sankar, S.: Claude 3.5 Sonnet vs GPT-4o — An honest review, Medium, 2024, URL: https://medium.com/@AIBites/claude-3-5-sonnet-vs-gpt-4o-an-honest-review-fa1d41ad81b5, visited on: 11/30/2024.

[SM23] Smart, J. F.; Molak, J.: BDD in Action: Behavior-driven development for the whole software lifecycle. Simon and Schuster, 2023.

[So] Software, S.: Introduction - Cucumber Documentation, URL: https://cucumber.io/docs/guides/overview/, visited on: 10/10/2024.

[Vi16] Vinsguru: Selenium WebDriver - Advanced Page Object Pattern with Page Fragments using Arquillian Graphene | Vinsguru, Section: Arquillian, 2016, URL: https://www.vinsguru.com/arquillian-graphene-page-fragments/, visited on: 09/24/2024.