Master's thesis

Master's Programme in Computer Science

Enabling Self-healing Locators for Robot Framework with Large Language Models

Paavo Rohamo

June 2, 2024

FACULTY OF SCIENCE

UNIVERSITY OF HELSINKI

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Paavo Rohamo

Työn nimi — Arbetets titel — Title

Enabling Self-healing Locators for Robot Framework Tests with Large Language Models

Ohjaajat — Handledare — Supervisors

Prof. M. V. Mäntylä

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	June 2, 2024	64 pages, 8 appendix pages

Tiivistelmä — Referat — Abstract

The fast development cycles of Web User Interfaces (UI) create challenges for test automation to keep up with the changes in web elements. Test automation may suffer from test breakages after developers update the UIs of the System Under Test (SUT). Test breakages are not defects or bugs in the SUT, but a failure in test automation code. Failing to correctly locate web element from the UI, is one of the key reasons for test breakages to occur. Prior work to gain self-healing of element locators, has been traditionally done with different algorithms and recently with the help of Large Language Models (LLMs).

This thesis aims to discover how to enable self-healing locators for Robot Framework Web UI tests, are there some web element locator types that are more easily repaired than others, and which LLMs should be used for this task. An experimental study was conducted for enabling self-healing locators for Robot Framework. Custom Robot Framework library was created with Python, which was tested for eight different locator strategies raised from locator breakage taxonomy.

Results show that the best performance in self-healing locators is gained by using the bigger LLMs. GPT-4 Turbo and Mistral Large showed the best performance accuracy by repairing 87,5% of the locators in the Robot Framework test cases. The worst performer was Mistral 7B Instruct which was not able to correct any locators. Using LLMs for self-healing locators in Robot Framework tests is possible. To get the best results for self-healing locators, my results suggest that practitioners should focus on LLM prompt designing, in the usage of candidate algorithm with locator version history and use the biggest LLMs available if possible.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation → Software prototyping

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

Test automation, Large Language Models, Self-healing, Artificial Intelligence, Robot Framework

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

One of the primary goals of test automation is to verify and validate that the System Under Test (SUT) works as desired [1]. However, the rapid development cycles of Web User Interfaces (UI) create challenges for test automation in keeping up with changes done to web elements. Test automation may suffer from test breakages after developers update the UIs of the SUT. These test breakages are not defects or bugs in the SUT but failures in the test automation code [2]. Due to the fragility of test automation, even the smallest updates to the SUT can break the test automation scripts, rendering them temporarily useless [3]. The most common reason for test breakages is the changes made to the web elements [2]. When test breakages occur, the test automation code requires additional maintenance [4]. The time required for this maintenance can range from quick and simple fixes to slow and complex ones, but in all cases, it takes time away from developing new features [5]. Web elements and element locators are different things. Web elements are the text, headings, links, etc. that you can see on a Web page [6]. Element locators on the other hand are a way to identify and find specific elements in Web pages [7]. Locators enable test automation to handle Web pages as if a real user would be operating it [8].

To address this issue with test breakages happening and the reason being in the web element locator, this thesis presents the possibility of Large Language Model (LLM) assisted Self-healing locators for Robot Framework test cases. This approach could have a major impact on how Robot Framework test cases are maintained in the future and has the potential for cost reductions for businesses with test automation practices. Traditionally frequent maintenance of test automation will lead to best results in financial perspective [3]. If test automation maintenance is neglected for long time periods during software development, then the cost of repairing may be significant. If the enablement of Self-healing locators for Robot Framework test cases could be done easily, results would be accurate, and the process be automatic, then there might be a spot for LLMs to maintain test automation in the future.

Self-healing and robustness of element locators have been studied by many [9] [10] [11] [12], but without the assistance of LLMs. Self-healing locators with the help of LLMs have only a few very recent studies which to rely on, the VON Similo LLM method by Nass et al. [13] and Guiding ChatGPT to Fix Web UI Tests via Explanation-Consistency Checking by Xu et al. [14]. Both studies use very similar method to gain the Self-healing of element locators and my thesis also uses some methods they introduced, like extraction of element attributes and information from the SUT webpage. Both studies used Selenium and Java combination as their test framework. In this thesis, I will build a Self-healing mechanism for Robot Framework and SeleniumLibrary. Apart from earlier studies, I will also conduct testing with many different LLMs, and test those capabilities to repair different types of element locators.

With this thesis, I'm trying to create proof-of-concept on how the Self-healing mechanism could be implemented into the Robot Framework. I am trying to broaden the knowledge from the LLM capabilities

and is there some LLM that could be suggested in the future for Self-healing practices with Robot Framework? I dive into the reasons for test breakages, and how these match with Robot Framework with SeleniumLibrary locator strategy. Finally, I present my results and analyze what were the capabilities of LLMs at self-healing element locators and how the performance could be improved.

* Note to the reader: As the University of Helsinki guidelines allow the usage of LLMs in theses, ChatGPT-4 has been used for grammar and vocabulary checking of this thesis.

2 Background

2.1 Software Testing

Whittaker, J.A. stated in his article “What is software testing? And why it is so hard?” that software testing is arguably the least understood part of the software development process [5]. This statement has been done over 20 years ago, but it may still have some grounding behind it. Even though, the testing processes have developed all the way from Waterfall to V-model and into the newest Agile methodologies [15]. Still, there are challenges that remain [4]. Software testing is a process where developed software is verified and validated against business and technical requirements [16]. Software testing is an important part of software development as it identifies defects, flaws, and errors in the software [16].

Software testing can be divided into two different approaches: manual and automated testing [17] [18]. In manual testing, the test personnel acts as an end user and executes planned test cases [18]. Test automation executes the same kind of planned test activities which are automated using specific automation tools or frameworks [18]. Both, manual and automated testing are crucial for overall software testing, because even though there are benefits in automated testing, fully counting at test automation is not seen as possible approach between practitioners [19].

2.1.1 Main Types of Functional Software Testing

The main types of functional software testing can be presented by tying it with the Software Development Life Cycle model (SDLC) [15]. For every step of development process there exists a corresponding testing level which can be observed from Figure 2.1. The V-model presents four different functional types of testing: Unit testing, Integration testing, System testing, and (User) Acceptance testing [16]. Follow the V-model, from left to right and first down then up [16]. It is suggested that testing should be done by different individuals at each level of testing, for example, developers performing testing at unit testing level and business users at acceptance testing level [1].

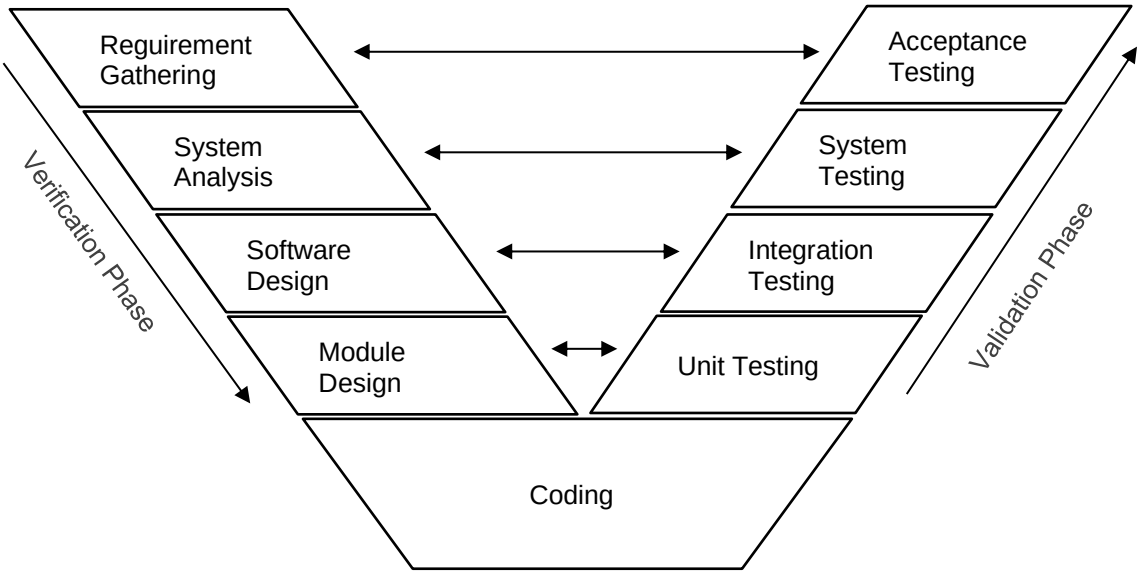


Figure 2.1: V-Model of Software testing, illustration based on Balaji et al. [15]

Unit Testing tests individual components and modules [5] [16] or a collection of those [5]. Unit testing is the lowest level of testing and in most cases created and executed by the developers [20].

Integration Testing covers the part of connections and communication between developed components and modules [20]. Integration testing ensures that the data flowing between different components was it new, changed or affected by change is correct [16]. Performance testing should be done during the integration testing phase, but because sometimes it takes too much effort from the environment and stakeholders at this stage, it can also be postponed being performed at the acceptance testing level [16].

System Testing covers the whole software. At this level, all the new components and modules which are created, changed, affected by change, or needed for the application are tested [16]. System testing validates and verifies that the functional design specifications are met [16]. Regression testing should be conducted after defects are discovered and bugs fixed to ensure that previously tested components and modules are still functional after repairs [20].

Acceptance Testing or User Acceptance Testing can be called Beta testing, application testing or end-user testing [16]. Regardless of the name used, at this phase, the developed software is tested by a mix of business users to evaluate that the software matches the business requirements [16]. The software is tested in against real-world scenarios and in this phase business users commonly find that software doesn't do things it was supposed to do or is not quite optimal doing so [16]. If testing is conducted like in V-Model, there should not be any defects or bug findings at this level. Most of the defects and bugs are found at earlier phases of testing [16].

As the V-model covers testing levels during the development phase, some organizations might incorporate an additional level called release testing after moving to production, which verifies the production environment [16]. V-model illustrates how each level should verify and validate the work done in earlier level, and how development is used to guide testing with different phases of software development cycle.

This connection between different levels and phases enables us to identify important defects, errors, and other problems early. [16]

2.1.2 Other Testing Types

Apart from the testing types presented earlier there exist multiple other types of testing such as black box, white box, regression, release, reliability, usability, performance, security, smoke and sanity testing, etc. [20]. All, but performance and security testing are part of functional testing types [20].

Performance and security testing are both non-functional testing types. The most popular types of performance testing are Load testing and Stress testing. Stress testing is done to get an understanding of the upper limits of the capacity in the system. Load tests on the other hand will determine the robustness of the system. Security testing is important so organizations can protect their information and services. Security testing is conducted either with a risk-based attacker mindset or based on the functional mechanisms existing within the software. [20]

All the tests from earlier phases can be used as regression testing for each affected functionality after subsequent software versions are developed and implemented into the software [5]. With regression testing, it is ensured that after changes, updates, or modifications, the existing features of the software system are not broken [21]. Regression testing can be carried at all levels of software testing either by doing it manually or by automating it [22]. Some testing practitioners suggest that it is best to test as early as possible, but some practitioners like to postpone regression testing as late as possible, and some might even practice regression testing throughout the software development process. So, there exist different opinions on when should we conduct the regression testing of a software system [22].

2.1.3 Costs of Testing

Testing a software is an expensive task, but the benefits of testing overcome the price of faults in the software products [18]. Annual costs of erroneous software in the United States only were about 59.5 billion dollars in the year 2000 [18]. According to studies, 50% of software maintenance costs come from testing practices, and 80% of total testing costs are from regression testing [22]. It is estimated that most regression test cases are run at least five times per project and one-fourth over twenty times [18].

It is important to test as early as possible. According to industry data, the costs of correcting and finding defects are lower if detected early [23]. The average price of defect fixed rises exponentially after proceeding from the coding phase to the testing phase and the price will double again if fixed in production. These can be observed from the Costs of Fixing Defects at Different Stages of SDLC chart in Figure 2.2 [23]. Average price of fixing defects at coding phase in year 2008 was approximately $977 US dollars, and fixing same defect at production stage would have cost $14102, which is about 14 times more [23].

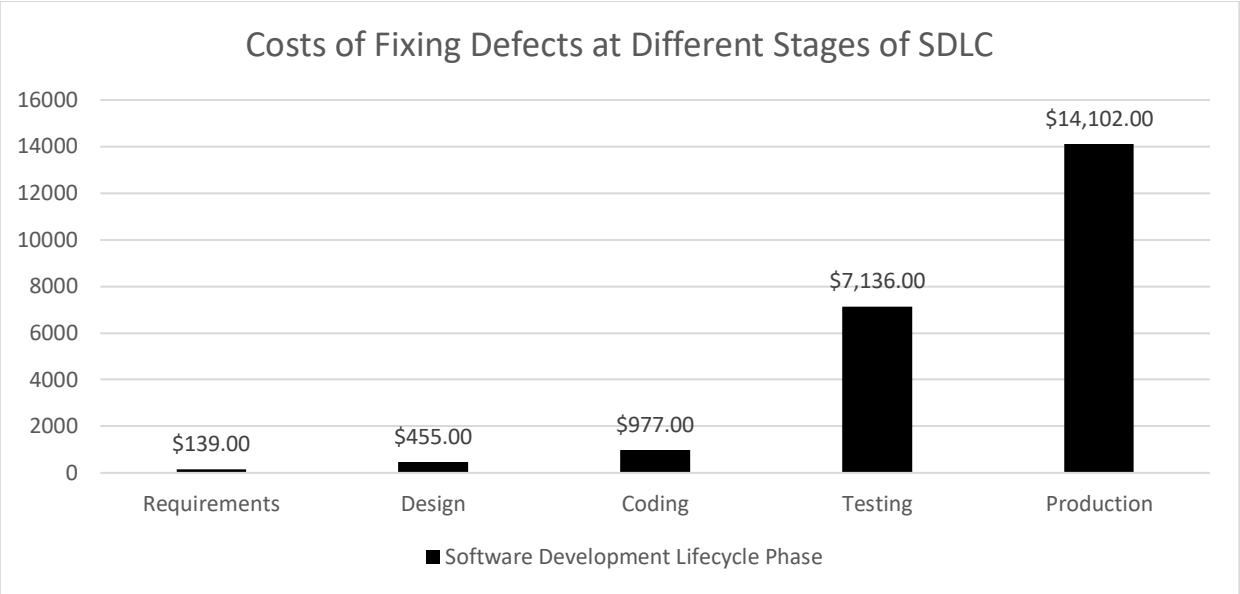


Figure 2.2: Costs of Fixing Defects at Different Stages of SDLC, based on charts by Lazic et. al. [23]

Doing more testing does not directly mean more costs if organizations automate the right parts of testing at the right time [17], also automating all testing is not seen as possible or cost-efficient [22, 17]. By automating the right parts of testing processes, organizations can improve the overall quality of software products, reach earlier defect identification, and simultaneously lower the overall cost of testing [18]. Especially the smoke, component, and integration tests are running constantly over the development of software and are good candidates for test automation [18].

2.2 Test Automation

Test automation means the automation of software testing activities which are usually done manually by human testers [17]. In test automation, the manual testing processes are scripted or recorded using enterprise or open-source test automation tools and frameworks [18]. The test automation tool or framework will run the test case according to the steps that the test engineer has scripted or recorded. It will execute every step and activity that was created. If scripting is used, then the test automation will likely require some level of knowledge in programming languages. [24]

Easy to understand model for test automation components is presented by Garousi & Mäntylä in the following way. According to them, test automation can be divided into four basic components: test engineers, test automation tools, test cases or suites, and System Under Test [17]. These four basic components interact in a very linear model to end up in the test automation process. Test engineers script or record test cases or suites with test automation tools. After test scripts or recordings are created, those are then executed using the test automation tool against SUT. Afterward, the test automation tool produces a test report for test engineers to interpret. [17]

Engström & Runeson reported best practices for test automation from practitioners in their qualitative survey

of regression testing practices [22]. Practitioners' responses included running automated tests daily at the module level and focusing automation testing below the user interface [22]. Wang et al. state that there are only six best practices with a positive effect on test automation, validated by academic studies [25]. According to them validated best practices that help to improve test automation maturity are: define a clear test automation strategy, provide sufficient resources, employ skilled professionals, select the appropriate testing tools, set up effective test environments, and design the system under test with test automation in mind [25].

Garousi & Mäntylä list test types that are good candidates for test automation: unit, smoke, load, performance tests, and tests that are similar to each other and contain large amounts of data, etc. [17]. When automating tests, organizations should also consider that tests created can be reused, which tests are critical, is the focus on high-quality test data and test scripts, are test results analyzed efficiently, adopt new technologies, is there enough budget, and is the schedule too tight [17, 25]. Also, it's important to have organizational and top management support for test automation because there might exist resistance against software test automation in organizations [17].

However, test automation is not total automation. It requires intervention from test engineers to diagnose test results and fix broken tests. [18] In the long run, test automation can bring many different benefits to the software testing processes, but it is not seen as a replacement for manual testing, because it's seen as impossible and cost-inefficient to automate all of testing [18, 19].

2.2.1 Test Automation Benefits and Maturity

Test automation will increase the efficiency of regression testing and help to reduce the time spent on manual testing [18]. According to Garousi & Mäntylä, the best pay-off with test automation will be achieved when automating regression testing and other repetitive testing tasks [17]. Direct benefits of using test automation reported by Taipale et al. are the quality improvement gained in the software, better test coverage in overall and more testing done in less time via the ability to reuse the already scripted test automation [18].

Rafi et al. list nine different test automation benefits, which are well-supported by evidence from experiments and case studies on implementing test automation, as detailed in their systematic literature review [26]. Benefits they list for test automation benefits include improved product quality, higher test coverage, reduced testing times, reliability, increased confidence in SUT, reusability, less human effort, cost reductions and increased fault detection [26].

To gain these benefits in System Under Test, also the test automation itself must be mature enough [17, 25]. It would be best if SUT would be a generic system with a long life cycle, and its' interface would be stable without major changes [17]. Regarding test automation maturity Wang, Y. et al. have collected thirteen key process area (KPA) best practices for assessing organizations test automation maturity [27]. Those maturity-related best practices include test automation strategy, resources, test organization, knowledge transfer, testing tools, environment and requirements, test design and execution, verdicts, test automation process,

measurements, and system under test [27].

Achieving a high level of test automation requires that mature practices are performed in each key process area [27]. There exists a multitude of evidence in the literature that if immature test automation practices are conducted it will lead to negative results. Selecting the wrong tools may lead to problems in organizational performance, automating test cases which have better fit for manual testing are waste of time and money, and inappropriate metrics may guide test automation in the wrong direction [27].

2.2.2 Known Challenges of Test Automation

Wiklund et al. (2017) made a systematic literature review of the challenges in test automation. They found out that there are many different "socio-technical" challenges present in test automation processes. Those could be divided into the following types of challenges: behavioral effects, business challenges, lack of skills, test automation tool challenges, and challenges in SUT. Behavioral effects could include excessively high expectations or negativity toward test automation. Business challenges often encompass a lack of time, people, funding, and steering. Additionally, deficiencies in development, testing, management, and automation skills are perceived as challenges. Test automation tools may also pose challenges if they are coupled with inadequate development practices, quality issues, or technical limitations. Furthermore, the quality, testability, complexity, and speed of the system under test are all challenges that impact on automation effectiveness. [28]

Taipale et al. conducted a case study where interview results revealed that one major challenge associated with test automation is the costs [18]. The challenge lies not only in the cost of implementing test automation but also in the costs associated with maintaining it and manually analyzing the results [18]. Wiklund et al. on the other hand presented this socio-technical system for the test automation development process where it can be observed from Figure 2.3 that time and resources, automation quality, and the perceived value will greatly affect the willingness to invest (costs) [28] As the test automation development is initially expensive and people involved may have unrealistically high expectations towards the value. If the expected results are not achieved it may result in the abandonment of test automation completely [28]

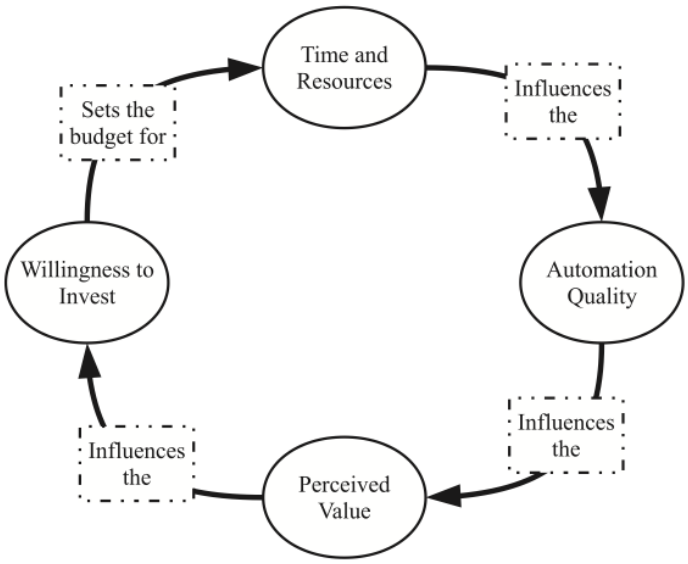


Figure 2.3: Feedback between connected phenomena as presented by Wiklund et al. [28]

Regarding automated GUI testing the challenges are similar according to Nass et al. [4]. They list three main groups where challenges occur: test execution fragility, tools and automation skills, and model based testing challenges [4]. Some of the challenges will remain regardless of what practitioners do such as, challenges with changing dynamic application or challenges with test automation tools. On the other hand, some challenges can be solved like automation and programming skills and robust locating of GUI elements [4]. According to Nass et al. at least nine academic papers have attempted to solve problems with robust locating of GUI elements and still the challenge exists [4].

Challenges may remain as Wang Y. et al. have discovered when there is a lack of guidelines on designing and executing automated tests which would help the test automation practitioners to make development and maintenance more efficient [27]. Also, they found there is a lack of the right metrics to measure and improve test automation processes in general [27]. Similar challenges have been found in automating tests for GUI applications where a large number of paths can be used to traverse the application and a lack of high-level guidance for typical patterns on how to test GUI applications efficiently [29].

2.3 Robot Framework

Robot Framework (RF) is a generic open-source automation framework [30, 31]. Originally developed by Nokia Networks from 2008 to 2015, and from 2016- onwards it has been supported by Robot Framework Foundation [32]. Robot Framework is licensed under Apache License 2.0, and it is free to use without any license fees [31] and has a rich ecosystem and a big open-source community around it, including multiple Finnish companies [31, 32]. Robot Framework is used in this thesis as the testing framework.

Robot Framework has a highly modular architecture which can be observed from the Robot Framework Architecture diagram in Figure 2.4. When Robot Framework is started, it processes the data, executes test

cases, and automatically generates logs and test reports [32]. Robot Framework has Built-in libraries, separately developed external libraries, and supporting tools to help with automating: building, editing, and running test automation [31].

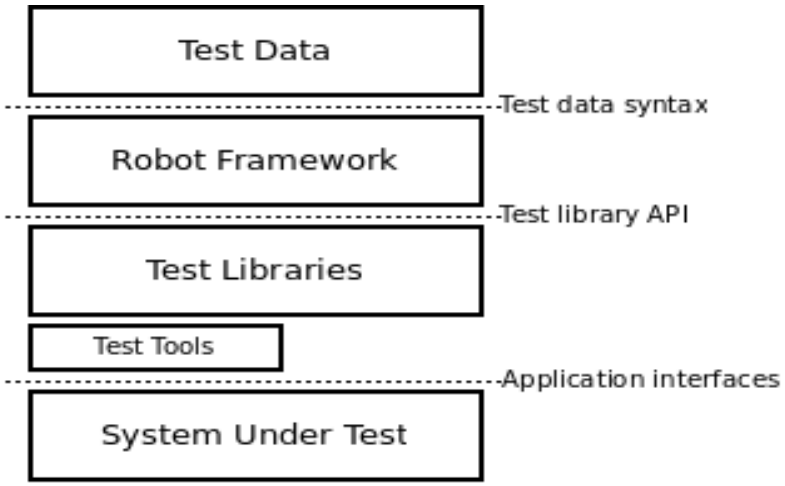


Figure 2.4: Robot Framework Architecture, from Robot Framework User Guide Version 7.0 [32]

Robot Framework has an easy syntax, and it utilizes human-readable keywords [32]. Keyword syntax can be observed from the example of the Robot Framework test case file in Figure 2.5. With Robot Framework test engineers can create test scripts without any demanding language and the design makes it easy to maintain and reuse the created test scripts [24]. In Robot Framework test cases are collected under Test Suites which may consist of one or many test cases [32]. Test cases are constructed from different available keywords, which can be created under keywords title or to be imported from test libraries or from resource files [32]. The test case is named based on the text in the first column, like “Valid login” in the example test case file in Figure 2.5. The second column normally has all the keywords listed which are executed in a test case [32]. Robot Framework test cases are executed from the command line, and the results of execution will by default be an XML format output file and HTML report and log [32].

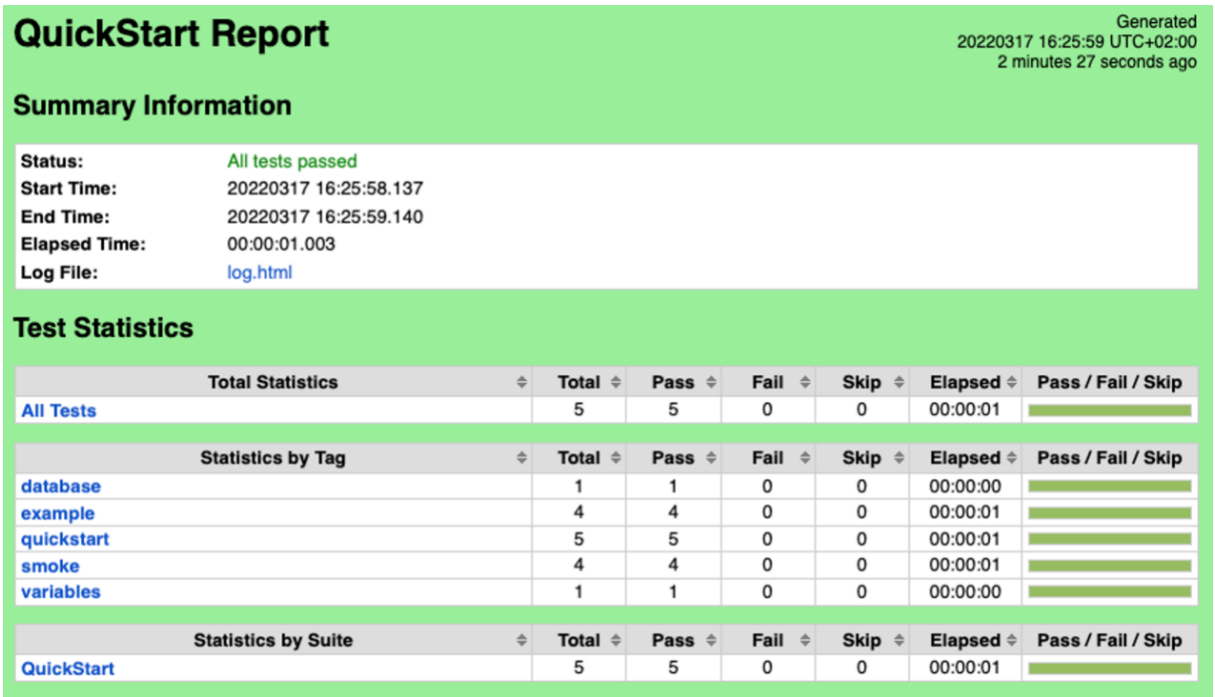
```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          login.resource

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

Figure 2.5: Robot Framework Test case file, from Robot Framework User Guide Version 7.0 [32]

Robot Framework report files in Figure 2.6 contain the results of all tests executed at specific suites in HTML format. The report file will be bright green if all tests have passed and red if there are any failed test cases. Report files have direct links to Log files which contain more detailed information about the failed test cases.

Robot Framework log files in Figure 2.7 have a hierarchical structure for showing the test suite, test case, and test keyword details. Figure 2.7 contains one test case, and keywords for it, and also keywords for suite setup and test teardown. Log files are essential for test engineers to gain knowledge about test results and especially to understand what went wrong in a test case. Reports are seen as a better way to get a high-level overview of test cases run, even though log files also include statistics. [32]



QuickStart Report

Generated
20220317 16:25:59 UTC+02:00
2 minutes 27 seconds ago

Summary Information

Status:	All tests passed
Start Time:	20220317 16:25:58.137
End Time:	20220317 16:25:59.140
Elapsed Time:	00:00:01.003
Log File:	log.html

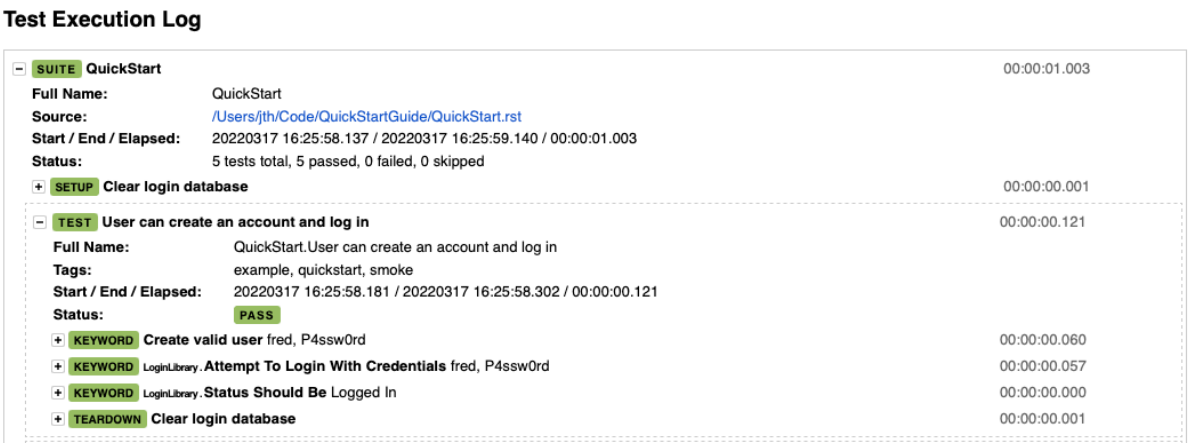
Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	5	5	0	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
database	1	1	0	0	00:00:00	
example	4	4	0	0	00:00:01	
quickstart	5	5	0	0	00:00:01	
smoke	4	4	0	0	00:00:01	
variables	1	1	0	0	00:00:00	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
QuickStart	5	5	0	0	00:00:01	

Figure 2.6: Robot Framework test report file from Robot Framework User Guide Version 7.0 [32]



Test Execution Log

SUITE **QuickStart** 00:00:01.003

Full Name:	QuickStart
Source:	/Users/jth/Code/QuickStartGuide/QuickStart.rst
Start / End / Elapsed:	20220317 16:25:58.137 / 20220317 16:25:59.140 / 00:00:01.003
Status:	5 tests total, 5 passed, 0 failed, 0 skipped

SETUP **Clear login database** 00:00:00.001

TEST **User can create an account and log in** 00:00:00.121

Full Name:	QuickStart.User can create an account and log in
Tags:	example, quickstart, smoke
Start / End / Elapsed:	20220317 16:25:58.181 / 20220317 16:25:58.302 / 00:00:00.121
Status:	PASS

KEYWORD **Create valid user** fred, P4ssw0rd 00:00:00.060

KEYWORD LoginLibrary.**Attempt To Login With Credentials** fred, P4ssw0rd 00:00:00.057

KEYWORD LoginLibrary.**Status Should Be** Logged In 00:00:00.000

TEARDOWN **Clear login database** 00:00:00.001

Figure 2.7: Part of Robot Framework test execution log file from Robot Framework User Guide Version 7.0 [32]

2.3.1 Robot Framework Libraries

The Robot Framework software is built to be expanded to various needs and it can be done in numerous different ways with libraries [31]. The Robot Framework core does not know anything about the SUT, and the interaction is handled by test libraries [32]. Robot Framework installation is included with Built-In tools and libraries such as Collections, DateTime, OperatingSystem, and String [31]. As Robot Framework is written with Python [32], test engineers can also install external test libraries based on their needs or even create their own internal libraries for Robot Framework by coding them with Python language [31]. Robot Framework also allows libraries to be written in C if using Python C API [32].

The four most popular externally developed libraries based on the Robot Framework Foundation site are SeleniumLibrary, Browser Library, HTTP RequestsLibrary, and AppiumLibrary. SeleniumLibrary is used for web testing [31, 33]. Browser Library uses Playwright and has a more modern point of view for web testing [31]. RequestsLibrary is used to interact with different Application Process Interfaces (API) and testing the functionalities of APIs [34, 33]. AppiumLibrary can be used for testing different mobile devices either with iOS or Android operating system [31]. In this thesis, I'm using SeleniumLibrary which enables Robot Framework usage in web testing.

User can implement their own internal libraries as Python modules or classes. The library name will be the same as the module or class implementing it. For example, the Python module "MyLibrary.py", will implement a library with the name "MyLibrary". Keywords can be implemented inside libraries as Python functions. Naming differs a bit and the function "def valid_login()" will be a keyword called "Valid Login" inside Robot Framework. User-created libraries are imported to Robot Framework from the "Settings" section by stating "Library MyLibrary". [32]

2.3.2 Selenium Library

SeleniumLibrary for Robot Framework is a well-known web browser testing library that utilizes the Selenium tool internally [35, 36, 37]. Selenium WebDriver modules are used internally to control a web browser in the Selenium Library [35]. To enable the usage of SeleniumLibrary in Robot Framework tests, the library must be first installed and then imported via the settings of Robot Framework, the importing mechanism can be seen in line 3 in Figure 2.8 [37]. Keywords that SeleniumLibrary provides are generally low level, so the best practice to implement these into test cases is to use Robot Frameworks higher-level keywords to utilize SeleniumLibrary keywords internally [37]. Implementation of SeleniumLibrary keywords can be seen in Figure 2.8, where the earlier presented Test case demo at Figure 2.5 has been expanded with SeleniumLibrary keywords.

```
*** Settings ***
Documentation     Simple example using SeleniumLibrary.
Library           SeleniumLibrary

*** Variables ***
${LOGIN URL}      http://localhost:7272
${BROWSER}        Chrome

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Title Should Be    Login Page

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button

Welcome Page Should Be Open
    Title Should Be    Welcome Page
```

Figure 2.8: SeleniumLibrary keywords extending Figure 2.5: Test case file. Image from SeleniumLibrary GitHub page [37]

2.3.3 Locating Web Elements in Robot Framework SeleniumLibrary

Locators which are used by Robot Framework and other languages can be divided into two sub-classes [2]. Attribute-based locators use element attributes like id, class, or name to identify elements. Structure-based locators are based on the structure of the web page, and use strategies like Xpaths, relative Xpaths, or CSS selectors to identify elements on the web page. [2]

Locating elements in SeleniumLibrary is based on locators that specify how to find an element on a web page [35]. SeleniumLibrary supports multiple different strategies for finding elements such as id, class, name, link, XPath expression, CSS selector, etc. Most often the locator is given as a string to the SeleniumLibrary. The strategy can be explicitly specified with a prefix, or the strategy can be an implicit XPath strategy. SeleniumLibrary also supports chaining of locators, usage of WebElements, and custom

locators. [35]

Explicit locator strategy is used with a prefix using either syntax *strategy=value* or *strategy:value*. Spaces around the separator are ignored, as *strategy:value and strategy : value* are equivalent. Explicit locator strategies which are supported by default at SeleniumLibrary can be observed from Table 2.1. [35]

Table 2.1: SeleniumLibrary Locator Strategies [35]

Strategy	Match based on	Example
id	Element id	id:example
name	Name attribute	name:example
identifier	Either id or name	identifier:example
class	Element class	class:example
tag	Tag name	tag:div
xpath	Xpath expression	xpath://div[@id=”example”]
css	CSS selector	css:div#example
dom	DOM expression	dom:document.images[5]
link	Exact text a link has	link:The example
partial link	Partial link text	partial link:he ex
sizzle	Sizzle selector deprecated	sizzle:div.example
data	Element data-* attribute	data:id:my_id
jquery	jQuery expression	jquery:div.example
default	Keyword specific default behavior	default:example

SeleniumLibrary documentation suggests that using ID as a locator should be done whenever it is possible. If IDs are not visible or are unstable then either tag, class, or CSS selectors are often an easy solution. In more complex cases, XPath expressions are typically the best or only viable approach, downside is that XPath expressions themself may have complex syntax. [35]

Implicit XPath strategy locators start with “//“, in other words using “//div” is equivalent to using explicit xpath://div. In addition to explicit and implicit strategies, user may also use Selenium’s WebElement objects, which requires first getting the WebElement from web page with using “Get WebElement” keyword for example. In more complex lookup cases custom locators can also be created. [35]

2.3.4 Robot Framework Listeners

The Robot Framework listener interface enables modification and inspection of test data and results during execution. The listener interface is called during the start or end of the suite, test, or keyword. This approach enables modification of test scripts during the execution. Listener interfaces are implemented as classes or modules that can be taken into use with “--listener MyListener” from the command line when starting the robot or as a registered library. There are currently two different listener interface versions. Listener version 2 and 3, both having similar methods, but 3 being more powerful and generally recommended currently.

Listener version 2 allows inspection and sending information about execution, but with it user cannot modify test execution data or results directly. Listener version 3 uses the same model objects as Robot Framework itself and allows modification of test data and results during execution. [32]

Robot Framework listeners may be used as test libraries. That allows the library to get notifications from listeners about the test execution. This allows listeners to make activities automatically to test cases at the start or end of a keyword, test case, or test suite. The main benefit of using the listeners is that modifications can be done dynamically based on execution data or results. This allows the modification of library instances, keywords, and others while tests are still running. [32]

2.4 Artificial Intelligence

In October 1950 Alan Turing asked, "Can machines think"? As he presented his concept of the imitation game which is now known as the "Turing test" [38]. Turing test is seen as one of the first concepts of machine intelligence, and some extremists might even state it as the birth of Artificial Intelligence (AI) [39]. Factually, the term artificial intelligence was coined for the first time at the Dartmouth meeting in 1956 by Professor McCarthy [40]. The philosophy behind the Turing test is how humans use language and can machines imitate it [38]. Some AI researchers state that the Turing Test itself is giving a bad name for the whole AI community [39], because they were being blamed for not being able to create AI which would pass the Turing Test. Turing test may have been impossible before [39], but recent advancements may have changed that [41].

The current level and capabilities of conventional AI as mainstream definition AI are seen as weak AI among researchers [40]. That can also be observed from definitions like "Strong AI", "Artificial General Intelligence (AGI)" or "Human-level AI" [40]. All of these indicate that today's AI is less than those mentioned, but all of these are still only concepts and theories [40]. The vision of current AGI research is to work as a guideline for future AI research because no real breakthrough has been made. However, some believe that now is the time to pursue for true artificial general intelligence [42]. If AGI is achieved, it could lead to superintelligence "which would greatly exceed the cognitive performance of humans in all domains of interest" [42].

AI research and development has seen its summers and winters as shown in Figure 2.9 [43]. The development of AI took big hypes in the 1960s and 1980s but had its downs in the 1970s and 1990s [43]. The latest "third tide" hype started around 1997 after Deep Blue won against Gary Kasparov [43], first by advancement in Machine Learning, and after that hype continued with advancement in Deep Learning [43]. After the release of OpenAI's large language model ChatGPT (Generative Pre-trained Transformer) in 2022 [43] [41], AI has attracted worldwide attention [41].

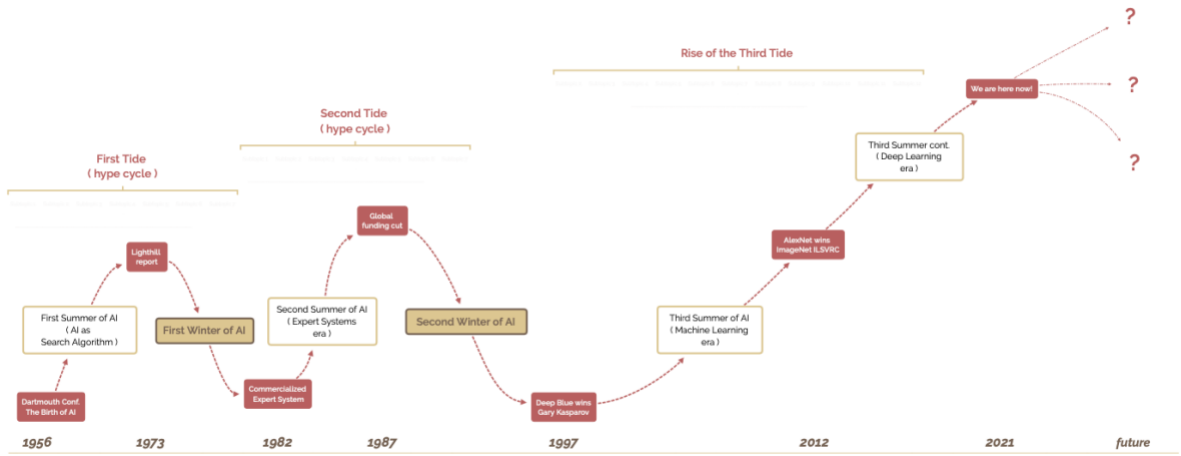


Figure 2.9: Modern history of AI presented by Toosi et al. [43]

The Capabilities of ChatGPT are enabled by Generative AI, which uses deep learning techniques with tremendous amounts of data to train the AI [41]. Generative AI can produce human-like text and generate creative content like pictures, videos, or music [41]. It is even claimed that the old Turing Test has now been passed by OpenAI's latest version of ChatGPT-4 [44, 41]. The generative AI model is an example of highly promising AI capabilities, and it is once again expected to revolutionize society and how we live [41]. Industry reports suggest that Generative AI could raise gross domestic product by as much as 7% [45].

Defining Artificial Intelligence explicitly is hard because, to this date, there is no widely accepted definition for AI [40]. The current field of AI is scattered, and each researcher can make their own decisions on what do they mean when defining their research under AI [40]. Famous AI practitioner Marvin Minsky defined it as: "the science of making machines do things that would require intelligence if done by men" [43].

2.4.1 Generative AI

The generative AI model is seen as highly promising unsupervised form of machine learning, which surpasses all previously presented generative models which lack generalization power [41]. The term Generative AI refers to a computational technique that is capable of generating seemingly new and meaningful content such as text, code, images, and music based on the training data [45]. Examples of Generative AI implementations are tools like OpenAI's chatbot ChatGPT and image generator DALL-E [46]. ChatGPT is based on the GPT (Generative Pretrained Transformer) large language model [46]. These kinds of generative AI tools are the result of different breakthroughs in the field of machine learning [46] and neural networks [41]. Most notably the paper "Attention Is All You Need" which proposed a neural network architecture called Transformers [41]. Paper presented a concept of self-attention which allows the model to attend different parts of the input and while simultaneously generating the output [41].

This kind of breakthrough has allowed generative AI to step into the new era of AI development. Until very recently machine learning was seen as limited predictive model which could observe and classify patterns in given content [46]. Machine learning models used to rely on supervised learning models where humans

oversaw teaching models what they should do [46]. Newer models can learn self-supervised when massive amounts of data are fed to the models. This type of self-supervised learning allows models to start generating predictions based on training data sets [46]. Accuracy comes from the massive amount of data [46]. Generative AI does not only perceive and classify given content but can also create text, images, or music based on it [46]. Although generative AIs can be seen creating new content there are risks in using them. Generative AIs are currently prone to occasional hallucinations which refer to phenomena where generated content is nonsensical to the given source input [41]. Responses might be seemingly correct but make no sense and are incorrect [41, 45].

2.4.2 Challenges and Limitations of Generative AI

Different ways to present challenges and limitations of Generative AI have been presented. Khankhoje presents five different categories for challenges: data quality, algorithmic biases, complexity of tools, challenges in integration, and explainability [47]. Nah et al. go a bit further and divide challenges into four main categories: Ethical, Technology, Regulations and Policies, and Economic challenges [41].

Seen ethical challenges are, harmful content, biases, over-reliance, misuse, privacy and security, and the digital divide. The effectiveness of using Generative AI heavily depends on the quality of the training data. If the data is inconsistent or contains inadequacies or biases, it may lead to false positives or negatives, thus compromising the reliability [47]. Privacy and security are other important factors to consider as using Generative AI may disclose sensitive or private information [41].

From the technological perspective, Nah et al. have listed challenges like hallucination, quality of training data, explainability, authenticity, and prompt engineering. Hallucinations of Generative AI may lead to answers that are incorrect or nonsensical. Generative AI might even generate fictitious outputs with false information in them. Hallucinated answers may sound extremely convincing [46]. Prompt engineering is seen as other important factor as with it we can mitigate these kinds of hallucinations. In other words, with good and specific prompts users can improve the quality and reliability of the content generated by Generative AI. [41] With prompting users can also manipulate Generative AI to give unethical answers [46].

Challenges seen in regulations and policies contain copyright controversies and governance as there exists the lack of controllability over the Generative AI behaviour [41]. Economic challenges are tied with unemployment caused by Generative AI and that some industries might be replaced totally with Generative AI. [41]

Differing from Nah et al. listed challenges Khankhoje states that the integration and complexity of AI-driven tools can also pose challenges as specialized skills and resources are required. There might be compatibility issues when integrating tools to existing Continuous Integration/Continuous Deployment pipelines [47].

2.4.3 Large Language Models

Large Language Models (LLMs) are subcategory of generative AI and currently in the forefront making generative AI known for the masses [48]. LLMs are computational models which have capability to understand and generate human readable language [49]. LLMs are trained with extremely big amount of data which allow them to generate fluent and precise text, phrases, and responses and align those with given inputs [49]. LLMs also have robust arithmetic reasoning capabilities and excels in logical reasoning and can understand the current context well [49] and can even generate code. Examples of easily accessible LLMs are Open AI's Chat GPT-3 and GPT-4, Meta's Llama models, and Google's Bert [48].

Even the smallest LLMs can write syntactically correct code, for example, Python code around 80% of the time [50]. Austin et al. have tested the performance of different LLM models to solve problems written in Python code, and the solving capabilities of the LLM get better related to the training set size of the LLM. In other words, the bigger the LLM the better the results [50]. Bigger LLMs can also solve easy problems more reliably than small LLMs, although there also exist problems that are unsolvable regardless of LLM size [50]. The performance of LLM size related to problem-solving capabilities can be investigated from Table 2.2 presented originally by Austin et al. They had a dataset of 100 Python code problems, and performance could be improved by editing the dataset to be more clear and more specific. Percentage of problems solved can be improved by using bigger models and with fine-tuning of datasets to be more precise. [50]

Table 2.2: LLM Problem-solving capabilities related to model size by Austin et al. [50]

Model size	**Dataset edited**	**% of Problems solved correctly (k=100)**
8B	no	35 %
8B	yes	45 %
68B	no	48 %
68B	yes	61 %
137B	no	63 %
137B	yes	79 %

2.4.4 Prompt Engineering for LLM

Inputs given to Large Language Models are called prompts and the practice of writing them prompt engineering [51]. LLM operates on tokens which differs on how humans perceive language. So basically, LLMs try to predict the next token that would fit to given input prompt and try to do it with the highest possible continuation [52].

The performance of LLM is very sensitive to the prompts. Austin et al. suggest that prompt-tuning could yield major improvements in the LLMs performance to solve problems related to coding languages. When creating prompts for LLM, instructions should be clear, and specific and contain examples [51]. Austin et

al. have found out that including test cases and natural language descriptions in the LLM prompt, will lead to the highest overall performance when solving problems with code languages [50]. Another way to improve prompts is to use short prompts with compact examples, as it will help LLM to answer with best results [50].

Prompt engineering can be done based on official OpenAI documentation [14]. OpenAI lists in their documentation ways to improve results and guides how to execute prompt engineering. OpenAI documentation for prompt engineering presents six strategies for getting better results from LLM: write clear instructions, provide reference text, split complex tasks into simpler subtasks, give the model time to "think", use external tools, and test changes systematically. OpenAI also provides different tactics on how to use these specific strategies when giving instructions to your specific LLM [53].

Xu et al. tied OpenAI's documentation to their prompt engineering in their study "Guiding ChatGPT to Fix Web UI Tests via Explanation-Consistency Checking" in a way that for each prompt content line, there is a corresponding "rule" in the OpenAIs prompt engineering documentation like in Table 2.3 [14].

Table 2.3: The patterns of prompt and generation rules as presented by Xu et al. [14]

Matching Prompt: Context[p1, p2, p3] + Input[p8], Repair Prompt: Context[p1, p4, p5] + Input[p9], Self-Correction Prompt: Context[p6, p7]

PID	Rule in OpenAIs official documentation	Prompt Content
p1	Use system instruction to give high level instructions	You are a web UI test script repair tool.
p2	Split complex tasks into simpler subtasks	To repair the broken statement, you need to choose the element most similar to the target element from the given candidate element list firstly.
		Give me your selected element's numericId and a brief explanation containing the attributes that are most similar to the target element.
p3	Provide examples	Your answer should follow the format of this example: "The most similar element's numericId: 1. Because they share the most similar attributes: id, xpath, text."
p4	Summarize or filter previous dialogue	To repair the broken statement, you chose the element <selected element>as the most similar to the target element from the given candidate element list.
p5	Specify the steps required to complete a task	Now based on your selected element, update the locator and outdated assertion of the broken statement. Give the result of repaired statement.
p6	Use delimeters to clearly indicate distinct parts of the input	This is a previous prompt: <Matching Prompt> This is your previous answer: <Corresponding Answer>
p7		But your explanation for attributes <attributes>are inconsistent with your selection and this will influence the correctness of your answer. Please answer again.
p8	Use delimeters to clearly indicate distinct parts of the input	Target element: <target element> Candidate elements: <candidate element list>
p9		Broken statement: <broken statement>

2.5 The Concept of Self-healing

Self-healing as a concept can be described as an approach to detecting improper software operation and subsequently initiating corrective actions without disrupting users [54]. In other words, it is the system's ability to automatically recover from faults [54]. As software execution environments become increasingly complex, and as software must continue to perform robustly, the adoption of self-healing mechanisms becomes essential. This is because resolving problems can require considerable time and effort. [54] The incorporation of AI technologies such as Large Language Models to enable self-healing into test automation is motivated by the potential to enhance the efficiency and speed of test processes [47].

Park et al. categorize self-healing into three main methods: architecture-based, component-based, and log-based. According to them architecture-based is more like external monitoring, which can't make changes to the target system, but uses the external environment to monitor, analyze, and reconfigure the target system.

Component-based focuses on the internal status of software and can make changes to target systems itself, but sometimes lack knowledge of the problem due to lack of resources. Log-based suggests self-healing based on logs generated by the target system. It is useful in searching for errors in log events but is unable to make corrections if an event has not been logged. [54]

2.5.1 Test Breakage

When test automation code fails, the problem might be a test breakage [55]. Test breakage is defined as an event where the test case raises an error or exception without the presence of a malfunction, defect, or bug in the SUT [55]. This significantly differs from cases where a test fails and exposes failures in the SUT. In such scenarios, developers must correct the application itself. However, in instances of test breakage, it is the responsibility of the test automation developer to fix the test automation code [55].

Reasons for test breakages have been identified by earlier studies, and taxonomies of test breakages have been created [2] [56]. Hammoudi has identified categories that cause test breakages in test automation, these test breakages occur when we are testing Graphical User Interfaces (GUI) or Web applications [2]. A total of 1,065 test breakage cases were analyzed, revealing five distinct causes: locators used, values and actions, page reloads, popup boxes, and changes in user session times [2]. According to the Taxonomy table presented by Hammoudi in Figure 2.10, we can see that 73.62% of breakages are caused by locators. A second major reason for test breakages is the values and actions with 15.21%. About two-thirds (~68,5%) of the locator-based test breakages are caused by Attribute-Based Locators and one-third (~31,5%) are caused by Structure-Based Locators.

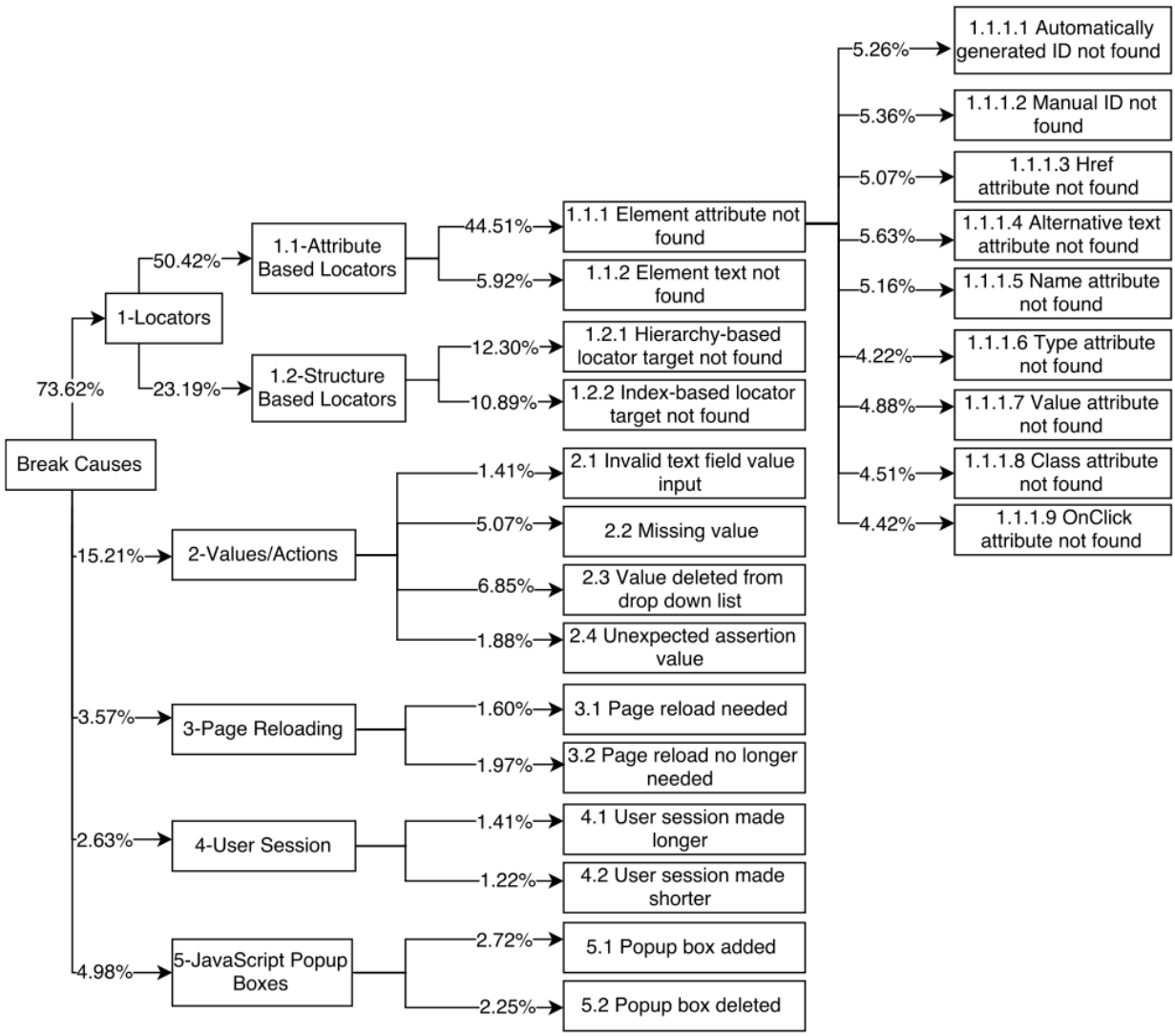


Figure 2.10: Taxonomy for test automation script breakage as presented by Hammoudi [2], the total number of cases was 1065.

Attribute-based locators are divided into two subgroups: the majority, which are element attribute not found-based locators, and the minority, which are element text not found. The 'element attribute not found' category includes nine different types: automatic ID, manual ID, link href attribute, alternative text, name, type, value, class, and OnClick attribute. [2]

Structure-based locators are divided into two subtypes: hierarchy-based and index-based locator target not found. Hierarchy-based locators fail because the DOM tree of the webpage changes after updates to the SUT. Updates affect the Xpath locators, which may be different if the location of elements has been changed. Index-based locators fail due to additions or deletions on the webpage, resulting in the locator matching a different element than originally intended. [2]

Test breakages are closely connected and may occur multiple times with different causes in a single test case. Tests should be run multiple times to allow other breakages to surface after the initial test breakage is fixed. Hammoudi suggests that finding repairs solely for locators will have the largest overall impact on

reducing test breakages. [2]

2.5.2 Prior Work of Conventional Self-Healing Locator Methods

Multiple conventional methods for self-healing of Element Locators have been proposed. These conventional methods don't utilize LLMs in any form. Earliest of presented methods are dated to 2011 and this far two different strategies to enable self-healing with conventional methods has been emerged. One of these methods is to repair locators after failures occur and the other method focuses in generating resilient and robust locators, so failures would not occur in the first place. [13]

Notably, approaches that focus on repairing element locators after failures occur are WATER by Choudhary et al. [57] and COLOR by Kirinuki et al. [10]. The WATER method compares passing and failing element locators from two different versions of the system under test. It uses equality or Levenhstein distance algorithm to match locator attributes like (XPath, coordinates, visibility, index, and hash) [57]. A bit better performing COLOR uses a very similar method than WATER that includes several locator attributes, but also includes element position, images, and other properties, and based on those it suggests repair for element locator [10].

Resilience and robustness of element locators have been studied by many, and several approaches have attempted to create robust Xpath element locators. Studies include iteratively changing Xpath locators by Montoto et al. [12], ROBULA+ by Leotta et al. [11], ATA-QV by Yandrapally et al. [9], and Similo and VON Similo by Nass et. al. [13], to name a few interesting approaches. The solution by Montoto et al. for robust locators is to iteratively change Xpath locators, starting from simple Xpath and adding sub-expressions until Xpath can be declared unique to make them change-resilient [12]. ROBULA+ approach by Leotta et al. is a novel algorithm that tries to generate robust Xpath locators. The idea is to start from the Xpath root locator and then select all webpage DOM elements and start to modify Xpath using transformations that affect (name, class, title, alt, value) attributes until only one webpage DOM element is selected which will be the eventual unique robust Xpath locator [11]. Montoto et al. solution and ROBULA+ doesn't fix anything, both just create unique robust locators that should work accordingly after updates happen to SUT [11].

ATA-QV by Yandrapally et al. uses information from neighboring web elements to triangulate the correct element locator. This allows ways to find the correct element locator even when attributes or other properties of the element are deleted [9]. Similo and VON Similo are both multi-locator approaches which uses locator attributes for identifying correct elements (id, xpath, label, tag) [13]. All of these ways have the prior knowledge of earlier versions of the locator and SUT, so these use the history information to be able to fix locators [9, 13]. The Similo attempts to identify element locators from set of candidates that have the most similar attributes to the last working locator. VON Similo uses same kind of method as Similo but adds visually overlapping nodes (VON) which increases the likelihood for locating correct elements. The VON concept in VON Similo simplifies the number of potential locator elements in a webpage by combining them into visual web elements [13].

2.5.3 Prior Work of Self-healing Locators with the Help of LLMs

Self-healing with the help of Large Language Models is a new concept and only a couple of studies exist. Transformers by Khaliq et al. [58] is the first from the year 2022, VON Similo LLM method by Nass et al. [13] from the year 2023 and Xu et al. [14] presented an approach of fixing UI locators with ChatGPT from year 2024. According to Xu et al. the existing techniques in Web UI test repair focus on finding target elements on the new web page that match the old ones so that the corresponding broken element locators can be repaired [14] and this achieved with different candidate algorithms between studies [14, 13].

Transformers for GUI Testing: A Plausible Solution to Automated Test Case Generation and Flaky Tests by Khaliq et al. [58] is one of the earliest methods to incorporate LLMs (GPT2) to automated test case generation and fixing of flaky tests. In the method implemented by Khaliq et al. they have an Appium server using the Pytest framework which was connected to GPT2 through element classification. The focus was more on generating test flows, but it was also capable of repairing flaky tests when the GUI is modified, by automatically generating new test cases based on the updated GUI. [58] So, it's fixing the test case around a specific locator, and not the locator in a specific test case.

The VON Similo LLM method by Nass et al. combines their advancements with conventional approaches with Similo and VON Similo and adds GPT-4 to that functionality to make it display some form of context awareness [13]. According to their results adding LLMs like GPT-4 to the conventional approach, it enables more accurate element locator fixes. LLM can understand the purpose of elements, analyze neighboring text, and evaluate web page structures which helps in locating and fixing correct element and thus further reducing the manual intervention and script maintenance [13].

VON Similo LLM method uses history information of broken locator to match it to candidate web elements extracted from the SUT. This approach will select top ten candidates that will match desired properties like id, class, name, Xpath of the broken target element. This list of top ten is then sent as part of a prompt in JSON format to the LLM. With prompt engineering, LLM is instructed to answer with the most similar candidate out of those 10 provided. The VON Similo LLM process can be observed from Figure 2.11.

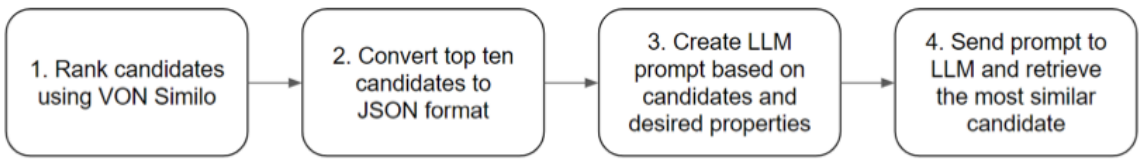


Figure 2.11: The VON Similo LLM process as presented by Nass et al. [13]

"Guiding ChatGPT to Fix Web UI Tests via Explanation-Consistency Checking" presented by Xu et al. [14] uses a similar approach to VON Similo LLM to repair broken element locators. They have used three different conventional methods to extract candidates from SUT. Conventional methods include usage of WATER, VISTA and Edit Distance (Levenshtein) algorithms for extracting top ten candidate element locator that are best match to the broken target element locator [14]. An example of candidate elements

returned when using the WATER algorithm be seen in Figure 2.12, other approaches (VISTA, Edit Distance) return similar candidates, but attributes differ. This approach is similar to the candidate algorithm used in the VON Similo LLM approach.

Target Element	{numericId=70, id='', name='new_category', class='', xpath='/html[1]/body[1]/div[4]/form[1]/table[1]/tbody[1]/tr[2]/td[2]/input[1]', text='Category1', tagName='input', linkText='', x=363, y=278, width=261, height=21, isLeaf=true}
Candidate 1	{numericId=20, id='', name='', class='button-small', xpath='/html[1]/body[1]/table[1]/tbody[1]/tr[1]/td[3]/form[1]/input[1]', text='Switch', tagName='input', linkText='', x=951, y=121, width=51, height=20, isLeaf=true}
...	...
Candidate 6	{numericId=70, id='', name='name', class='', **xpath='/html[1]/body[1]/div[3]/form[1]/table[1]/tbody[1]/tr[2]/td[2]/input[1]'**, **text='Category1'**, tagName='input', linkText='', x=403, y=295, width=261, height=21, isLeaf=true}

Figure 2.12: An example that shows the target element to be matched, and a list of candidate elements returned by WATER, presented as is in paper by Xu et al. [14]

Out of different conventional methods used by Xu et al. VISTA uses the Fast Normalized Cross Correlation algorithm which return a list of elements at particular screen position that match the target element. WATER uses different attributes (id, Xpath, class, linkText, name) to match the locator to the target element. Edit Distance uses only Xpath similarity to produce the candidates for the broken target element locator [14]. The exact workflow of test repair using ChatGPT by Xu et al. can be investigated in Figure 2.13.

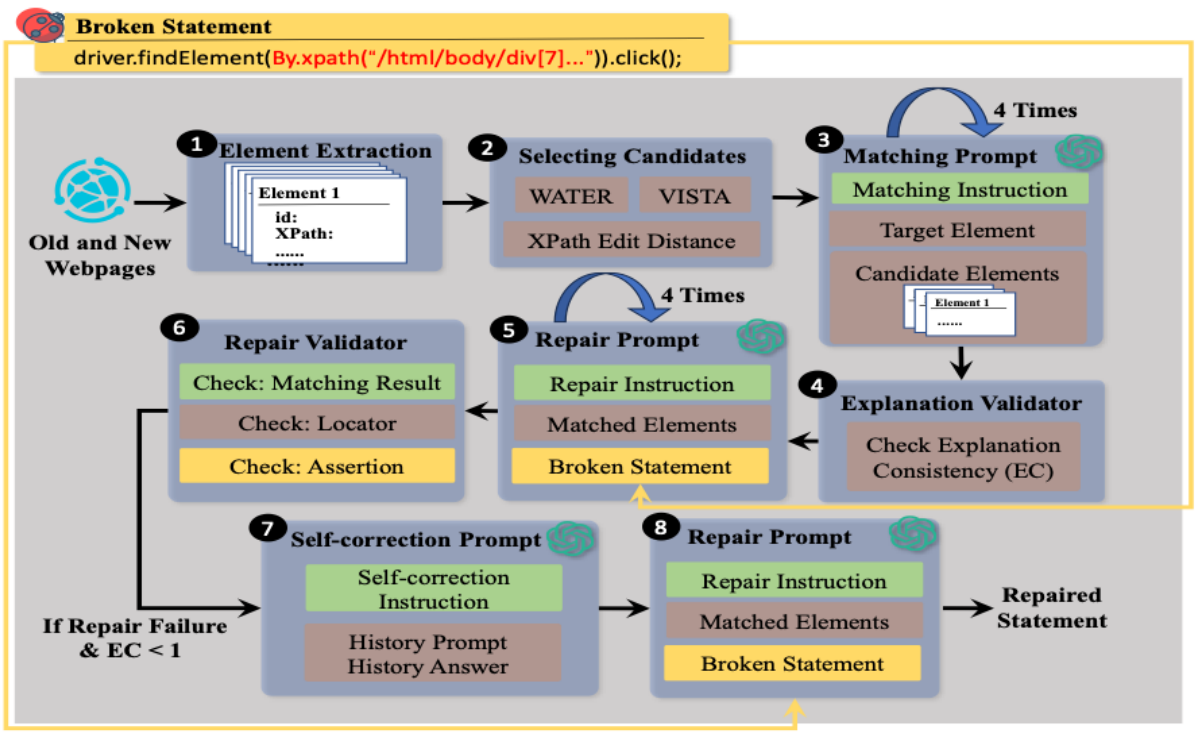


Figure 2.13: The workflow of Web UI test repair using ChatGPT as presented by Xu et al. [14]

Performance results when using VON Similo LLM [13], or VISTA [14], WATER [14], Edit Distance + LLM [14] methods can be observed in Table 2.4. VON Similo LLM has achieved 95% accuracy when repairing broken element locators, Nass et al. used ChatGPT-4 in their study [13]. Xu et al. have achieved accuracy varying from 61.9% to 87.1% between their different methods. Edit Distance which used only Xpath to locate elements with ChatGPT-3.5Turbo being the highest performer [14].

Table 2.4: Results achieved when using LLMs to repair broken locators, excluding Khalid et al. work as it repairs test cases around specific locators.

Technique	**LLM used**	**Accuracy to repair broken element locator**
VON Similo LLM	ChatGPT-4	95.0 %
VISTA + LLM	ChatGPT-3.5Turbo	69.8 %
WATER + LLM	ChatGPT-3.5Turbo	61.9 %
Edit Distance + LLM	ChatGPT-3.5Turbo	87.1 %

3 Methods

3.1 Research Objectives and Questions

The goal of this study is to address the following research questions:

RQ1: How can self-healing locators be enabled in Robot Framework test cases with the assistance of Large Language Models?

RQ2: What kind of locators in Robot Framework test cases have the most potential to be repaired with the assistance of LLM?

RQ3: What are the requirements for LLMs to be able to assist in repairing locators?

RQ4: How accurate are LLMs when suggesting repair for Robot Framework locators?

Based on prior work in the field of repairing and self-healing element locators, my hypothesis was that the concept of self-healing locators could be successfully applied to Robot Framework test cases. The main objective of this study was to determine how self-healing locators could be enabled for these test cases. A crucial part of this self-healing process would be the assistance of LLMs. I assumed that some locators in test cases might be impossible to repair without understanding the context. Based on that, I also aimed to discover what kind of contextual information LLMs require from the test cases and what is necessary for suggesting repairs.

I sought to determine whether LLMs can suggest repairs without using candidate algorithms, as employed in prior work, and whether information from the old version of the SUT is necessary for enabling self-healing locators. I hypothesized that self-healing locators could be achieved without prior knowledge of earlier versions of the SUT, despite previous studies by Nass et al. and Xu et al. utilizing information from both old and new versions of the SUT. Given the versatility of the Robot Framework, I envisioned that providing sufficient information about the test case and the locator would enable LLMs to repair the locator based on the information provided.

3.2 Research Context

In terms of testing, this research will focus solely on Robot Framework UI test cases. Nass et al. have pointed out that many studies suggest challenges are particularly prevalent in GUI test cases [4], which are the ones that use element locators.

Test breakage-wise, the focus would be on the self-healing of locators with the assistance of LLMs. According to Hammoudi [2] the majority of test breakages (~73.5%) were caused by broken locators. Hammoudi suggested that the greatest benefits in fixing test breakages could be achieved by concentrating on fixing locators. Based on the taxonomy presented by Hammoudi, I created Robot Framework test cases

that could be used to simulate locator-based test breakages against different versions of the System Under Test (SUT).

To sufficiently answer research question 4: How reliable are LLMs when suggesting repairs for Robot Framework locators? there was a need to include different LLMs in the execution phase where the self-healing mechanism would be tested. By testing the self-healing ability with different LLMs, I could provide information on whether there are significant differences when using different models.

3.2.1 Selecting Large Language Models

The selection of Large Language Models (LLMs) for this research was based on the latest and most popular LLMs that are easily accessible. Four of the models were instruction-finetuned, as shown in Table 3.1. Instruction fine-tuning means that these models better follow the instructions given in the prompt and are more helpful in general [59]. The selection included OpenAI's latest models, GPT-3.5 Turbo and GPT-4 Turbo, Meta's models, Llama 3 70B Instruct and Llama 3 8B Instruct, from Mistral: Mixtral 8x22B Instruct, Mistral 7B Instruct, and Mistral Large, and Anthropic's Claude 3 Sonnet. I wanted to include some of the most popular LLMs available, such as GPT-4, as well as smaller models like Mistral 7B Instruct. This allows me to assess the capability differences between models and determine if there is a model that could be recommended in the future. Additionally, the cost of each model is a major consideration. For instance, the cost for GPT-4 Turbo to process 1 million tokens is $30, whereas for Mistral 7B Instruct and Llama 3 8B Instruct, it is only $0.07. There are also differences in context length between models, which is not within the scope of this study, as my prompt length will be a maximum of about 1000 tokens.

Table 3.1. Selected LLMs: Input cost, output cost, and context token count extracted from Openrouter.ai [60]

Model name	**Prompt cost ($ per 1M tokens)**	**Completion cost ($ per 1M tokens)**	**Context (tokens)**
Claude 3 Sonnet	$3	$15	200,000
GPT-3.5 Turbo	$0,5	$1,5	16,385
GPT-4 Turbo	$10	$30	128,000
Llama 3 8B Instruct	$0,07	$0,07	8,192
Llama 3 70B Instruct	$0,59	$0,79	8,192
Mistral 7B Instruct	$0,07	$0,07	32,768
Mistral Large	$8	$24	32,000
Mixtral 8x22B Instruct	$0.65	$0.65	65,536

3.3 Experimental Study Design

3.3.1 Setting Up the System Under Test

One of the first tasks in this experimental study was to decide to either create the system under test or obtain some ready-made boilerplate for its setup. Since the primary focus of this study was to enable self-healing locators with LLMs for the Robot Framework rather than webpage creation, I decided to proceed with the boilerplate option. After spending some time searching for potential boilerplates that were easy to use and manage, I eventually chose the Rocket Django dashboard, which was well-suited for this experiment. The free version of Rocket Django includes APIs, data tables, charts, and login functionality, and is both simple and complex enough to facilitate the creation of real-world-like Robot Framework test cases. As Rocket Django is Docker-ready, it can be deployed to run on localhost with a simple command: "docker-compose up --build". Docker Compose builds and loads all the necessary requirements, and by default, the dashboard is hosted at localhost:5085. The landing page of the Rocket Django dashboard can be seen in Figure 3.1

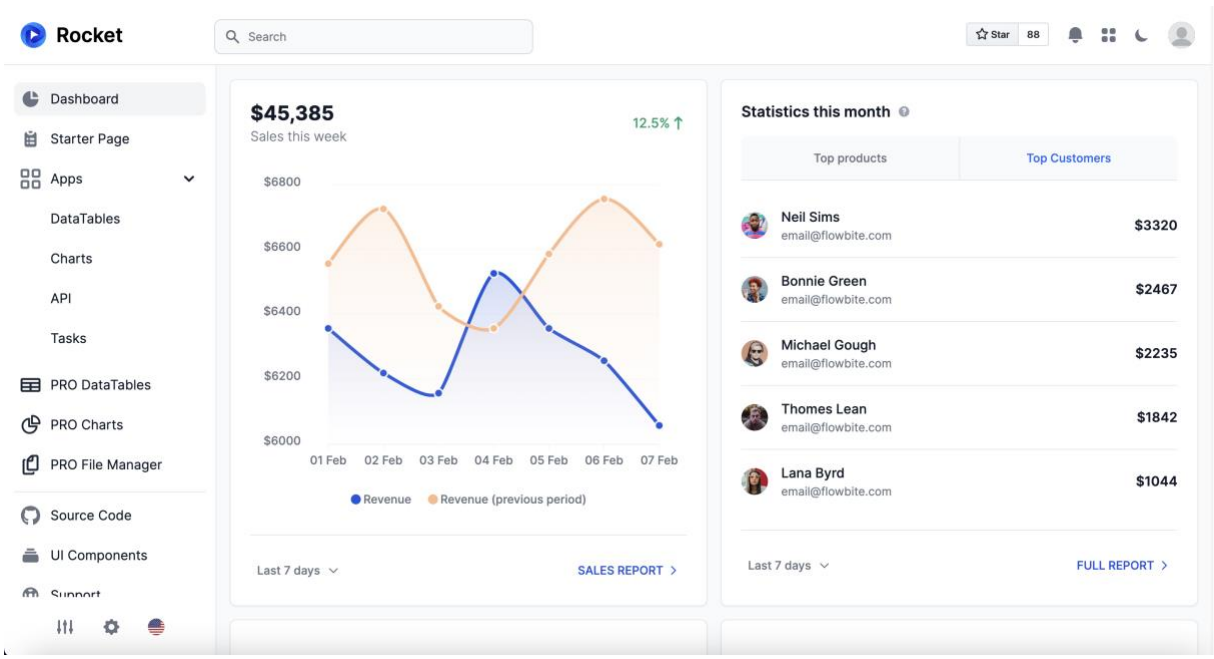


Figure 3.1: Rocket Django dashboards graphical user interface (GUI) [Screenshot taken, 25.4.2024]

This approach made it easy to create another version of the Rocket Django dashboard that could be hosted locally but on a different port (localhost:5095). Simultaneously it allowed me to develop Robot Framework test cases against the original SUT. With this approach, I was able to first create passing Robot Framework test cases against the original SUT and, after a simulated version update (new version of SUT running in different port), run those same tests against a modified version of SUT, by just changing localhost port at Robot Framework test scripts. The modified version of the SUT was achieved by making changes to the Rocket Django dashboard HTML code. Changes made to the code can be observed from Figure 3.2.

All the Robot Framework test cases would pass against the original SUT (localhost:5085) and fail when run

against a modified version of SUT (localhost:5095).

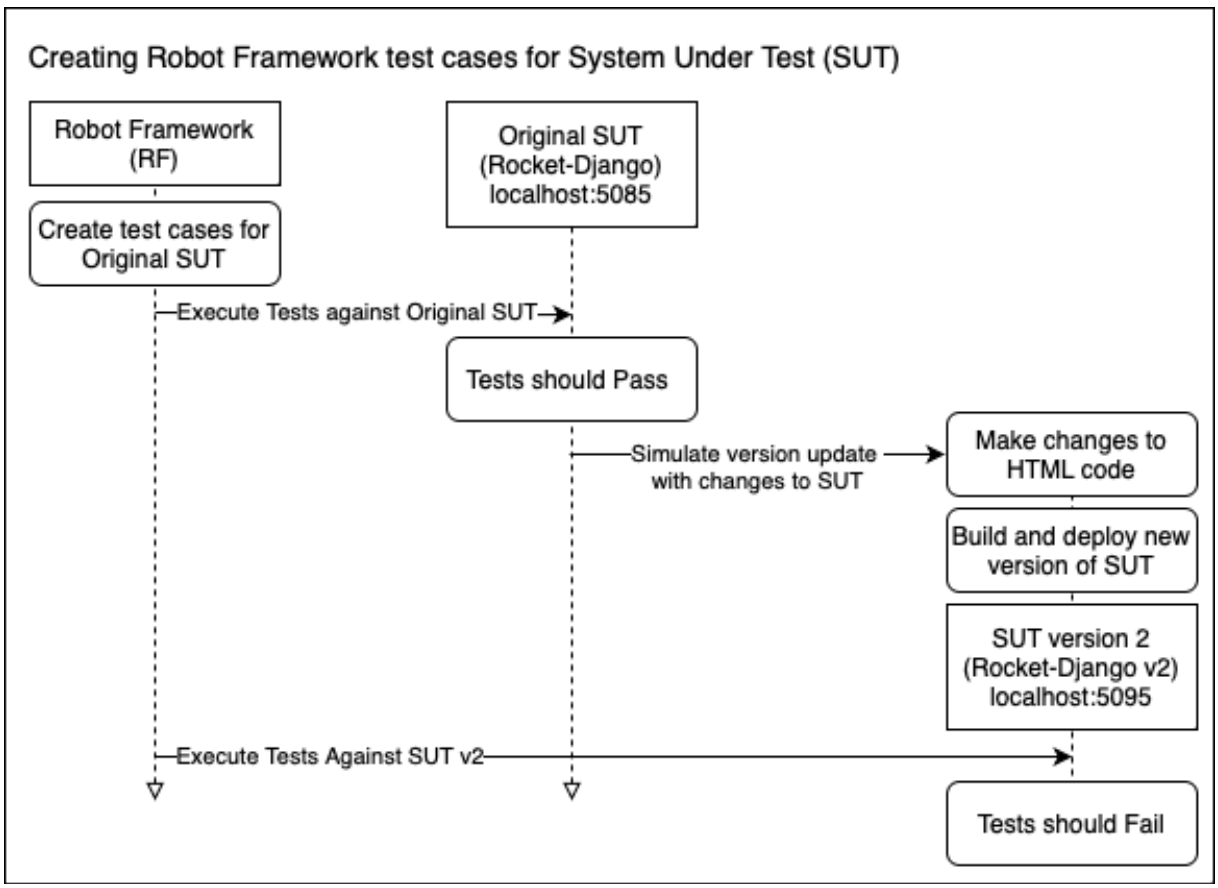


Figure 3.2: Creating Robot Framework test cases for System Under Test (SUT)

3.3.2 Creating Robot Framework Test Cases

I started planning the test case creation by looking into the best practices of Robot Framework. Best practices can be found in the Robot Framework Style Guide [61]. Style Guide suggested that existing standards from Robot Framework User Guide should be used [32, 61].

As it was found out from test breakage taxonomy studies the greatest benefits would be gained by focusing solely on fixing locator issues. I created the following plan for creating Robot Framework test cases which would account for different locator breakage causes presented by Hammoudi [2]. The robot Framework test case plan can be inspected in Table 3.2. While planning I found out that some of the locator types presented in the test breakage taxonomy were not supported by Robot Framework SeleniumLibrary. There were four attribute-based locators that were missing: alternative text, value, type, and OnClick. These were not supported on the SeleniumLibrary locator strategy by default. For the “type attribute” I decided to make support with Xpath expression, but for others, I decided not to include those into the plan as either those would be very similar to the “type attribute” or I would have to do custom locators which are not in the scope of this study.

Automatically and manually generated IDs not found were originally separated in the taxonomy, but as SeleniumLibrary only sees unique IDs there was no reason to do those separately.

Table 3.2: Robot Framework Test Cases

Test Id	Locator Base	Taxonomy Locator Type	Test Case Name	Selenium Locator Strategy Used
A1	Attribute	Element text not found	All Apps Are Visible At Navigation	Xpath expression, xpath://div[@id="example"]
A2	Attribute	Automatically and manually generated id not found	Theme Can Be Toggled To Dark Mode	id:example
A3	Attribute	Href attribute not found	Navigation To Starter Page From Sidebar Is Functional	link:example href:example
A4	Attribute	Name attribute not found	Task List Scripts Can Be Changed	name:example
A5	Attribute	Type attribute not found	Navigation To Starter From Sidebar Is Functional	Type not supported by Selenium, using xpath expression for selecting type: //*[@type="submit"]
A6	Attribute	Class attribute not found	Dashboard Search Has Label Attached	class:example
S1	Structure	Hierarchy-based locator target not found	Statistics Table Visible At Dashboard	Xpath expression, xpath://div[@id="example"]
S2	Structure	Index-based locator target not found	All Apps Are Visible At Sidebar	Xpath expression, xpath://div[@id="example"]
N/A	Attribute	Alternative text	-	Alternative text not supported by default in Robot Framework SeleniumLibrary
N/A	Attribute	Value	-	Value not supported by default in Robot Framework SeleniumLibrary
N/A	Attribute	OnClick	-	OnClick not supported by default in Robot Framework SeleniumLibrary

Robot Framework test cases for this study were developed using the plan created. Overall, 8 different test cases were created that uses SeleniumLibrary to handle the website functionality. For each test case, I included ‘Test Id’ as a test case [Tags], so they could be run easily separated if needed.

3.3.3 Running Tests Against Original SUT

To run test cases against the original SUT, navigate to the root of the test case folder using the command line tool and type the command “robot --variable env Tests.robot”. In this command, I set the variable "env" with the value "v1" as an argument to the test run. I configured it so that when the "env" value is "v1", it points to the original SUT at localhost:5085, and when the "env" value is "v2", it points to the modified SUT at localhost:5095. Initially, all eight test cases passed against the original SUT. A screenshot from the command line shows my Test Suite run, which contains eight different tests with test IDs visible. The

screenshot of the test run can be seen in Figure 3.3.

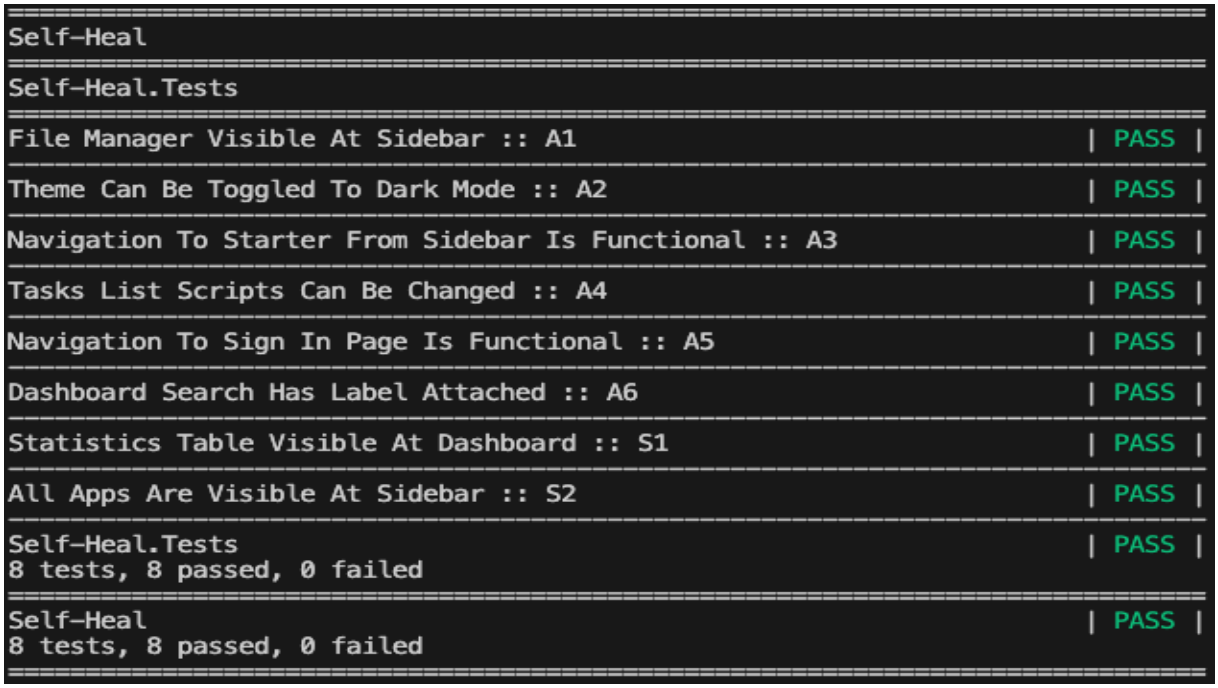


Figure 3.3: Screenshot from the command line. Test case run against original SUT

3.3.4 Changes Made to the System Under Test to Achieve Test Breakage

After creating the initial Robot Framework test set, it was time to modify the System Under Test (SUT). Various modifications were made to the attributes such as id, class, label, and text, and to the DOM structure. Following these changes, SUT v2 was built and deployed using the docker compose-up command. The new version of the modified SUT was up and running at localhost:5095. The changes made are detailed in Table 3.3.

In hindsight, I realized that simulating the version update or SUT modifications could have been approached differently. An alternative approach could have been to create test cases against the SUT with failing locators from the beginning. However, this method seemed more authentic than simply modifying failing locators in the Robot Framework code, so I decided to proceed with this approach.

Table 3.3: Changes made to the SUT

ID	Change Type	Originally	Modification
A1	Element text	”PRO File Manager”	”File Manager”
A2	Element id	id:theme-toggle	id:theme-toggler
A3	Link text	Starter Page	Start Page
A4	Element name	name:script	name:scripts
A5	Button type	type:submit	type:button
A6	Element class	class:sr-only	class:search-only
S1	Remove DOM element	-	Removed one DIV from Dashboard
S2	DOM Element position switch	Order of navigation elements [DataTables, Charts, API, ..]	Order of navigation elements [API, DataTables, Charts, ..]

3.3.5 Running Tests Against Modified SUT

The created Robot Framework tests were run against the updated SUT, now running at localhost:5095. This time, the command used was “robot --variable env Tests.robot,” which changed the URL to the correct port. As shown in Figure 3.4, all 8 out of 8 test cases failed. Robot Framework also displays the reasons for each failure under the respective test cases.

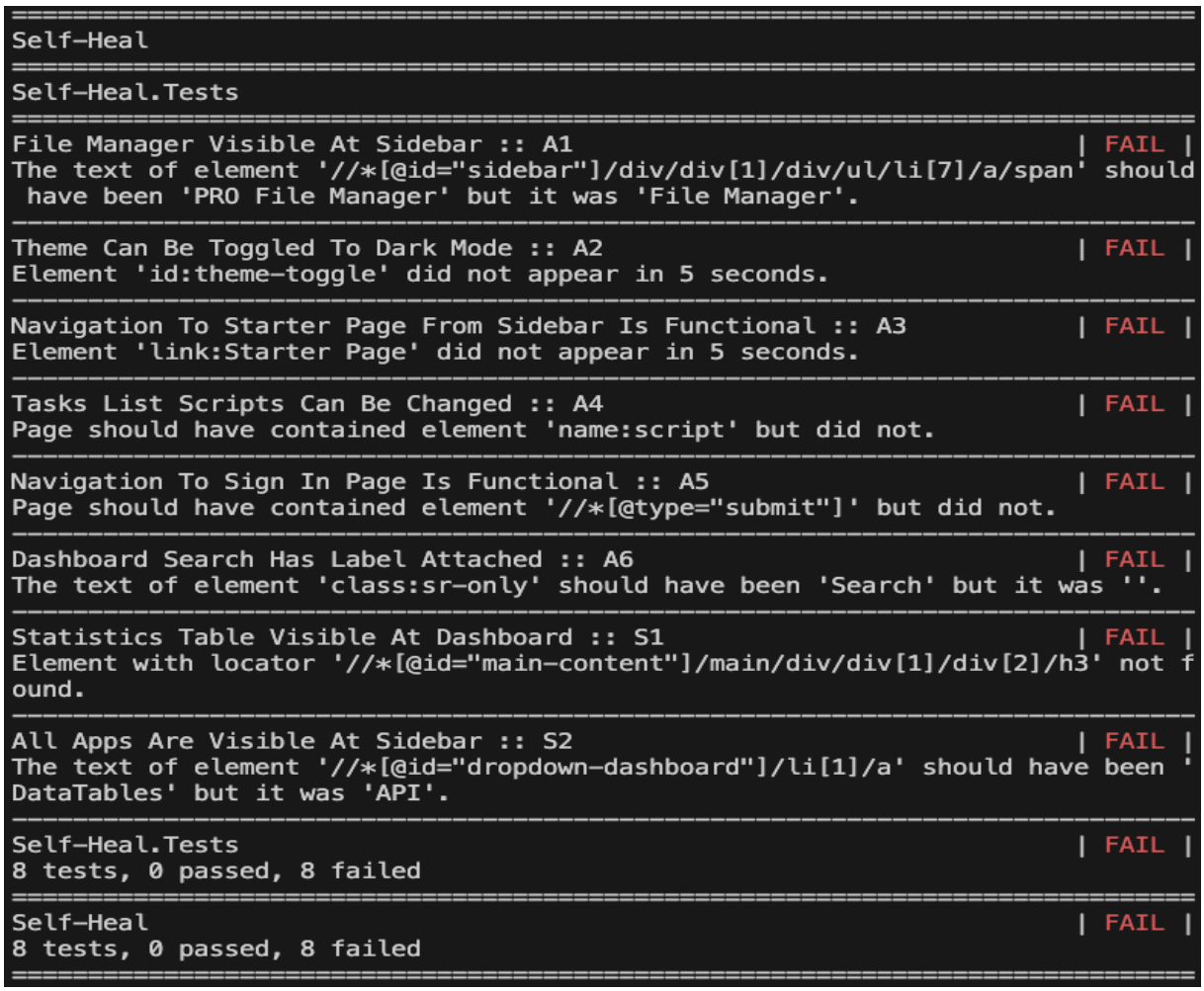


Figure 3.4: Running tests against updated SUT version 2

3.3.6 Planning on how to integrate Self-healing Mechanism into Robot Framework

Initially I had a concept in mind how this integration of self-healing locator mechanism to Robot Framework test cases would be done. I based my concept on relevant recent studies from Nass et al. [13] and Xu et al. [14], and based on my initial assumptions I first decided not to include a candidate algorithm in my self-healing mechanism. As I believed that I could provide enough resources for LLM from the test case and SUT, so it could make suggestion on how to repair the broken locator.

The Self-healing mechanism itself was planned to be initialized in the “Test Teardown” part if a test failed. In Robot Framework, both Suites and Tests have their own separate teardown mechanisms, which are executed after each Test Suite or Test Case, regardless of the test outcome (pass or fail) [32]. So even if a test fails, the keywords placed in teardown will be executed. The plan was that the self-healing mechanism would be called from Test Teardown, when a test fails. It would first collect the required resources from the test case and from the SUT, then proceed to create the prompt message for LLM API and send the request. The LLM would respond with suggestions on how to repair the broken locator of the test case. After receiving the response from LLM, the self-healing mechanism would automatically update the Robot Framework code by changing the locator to the new one suggested. The recently repaired test case could

then run again, but only manually at this point.

After a few initial tests with first version of the self-healing mechanism using GPT-4, I noticed that there was not enough context for the LLM to make repair suggestions for failed locators. GPT-4 was able to make the correct locator suggestion on one of the test cases (test id=S2), as shown in Tables 3.2 and 3.3. However, smaller LLMs like GPT-3.5 were unable to make any locator corrections and were only guessing or hallucinating. At this point, I realized that without some kind of locator candidate algorithm, the results would be inadequate.

3.3.7 Candidates

Recent studies using LLMs to repair locators, such as those by Nass et al. [13] and Xu et al. [14], employed similar methods to derive candidates for the LLM. Both studies used the top 10 candidates and included version history for the locators when they were still passing. Essentially, these studies compared locators from two different SUT versions to find a match for the failing locator.

One might ask why we don't simply extract the entire HTML page source from the SUT and send it to the LLM instead of these candidates. While this could be an option if the SUT's HTML page size is small enough, for a self-healing mechanism to be suitable for more general use, we must understand that larger LLM prompt inputs can reduce the accuracy of locator repairs, according to earlier studies. Additionally, the HTML page source might exceed the context size that an LLM can handle as input, and sending large prompts also increases costs. This might change in the future as LLMs grow in size and context length capabilities expand.

I was still hesitant about using locator version histories or databases, as this would require modifications to existing continuous integration and continuous development systems when implementing this self-healing mechanism. As a result, I first implemented a candidate method that would simply read the HTML page source and the current failing locator. The best candidates could be found by matching the failing locator to words in the source code using the Levenshtein distance algorithm with a sum weight of 2, provided by the RapidFuzz Python package [62]. This resulted in a list of potential candidates with probability ratios, which was then narrowed down to the 10 most probable candidates. With this method, I could already see improvements, but there were still a lot of hallucinations from smaller-sized LLMs and some guessing from the larger ones.

At this point, I realized I had no other option but to include the locator version history into the mechanism. Initially, this was problematic because Robot Framework's SeleniumLibrary does not currently have built-in functionality to fetch all attributes and information for a specific locator or do this for every visible locator. A workaround for this problem was found in a study by Nass et al. [13] , which included a replication package. From that replication package, I found JavaScript functions that could perform these specific fetches. Since Robot Framework can execute JavaScript using built-in keywords, retrieving all necessary information became straightforward. All passing locators are stored in TinyDB, a Python package that uses

JSON files as a database [63]. An example of a locator stored in the Locator DB can be seen in Table 3.4.

I named these candidate methods the Weak Candidate Method and the Strong Candidate Method, naming was based on the quality of the candidates. The Weak Candidate Method does not have information about the locator's older versions. The Strong Candidate Method uses locator attributes and information stored in TinyDB, which I refer to as the Locator Database in this thesis.

3.3.8 Retrieving and Storing Locators

This method was employed when using the Strong Candidate Method. I used JavaScript commands to retrieve all the attributes and information of passing locators during execution. This method is similar to what Nass et al. [13] used in their study, except I removed the text element from the JavaScript command as it was causing problems for LLM requests by retrieving many line changes and empty lines. These text elements would require slicing and stripping to be usable. Although there is significant potential for improving the stored values, my main goal was to enable self-healing for Robot Framework locators, so I did not explore this matter further.

An example of the attributes and other information stored in the Locator Database for "id" used in test case ID A2 can be seen in Table 3.4. The final solution involved storing element attributes such as tag, class, type, name, id, value, and title. Other information stored includes the element's position on the screen, width, and height. DOM-related information retrieved and stored includes the number of children, Xpath, and idXpath. Lastly, the locator used by the Robot Framework test case is saved as the last locator value.

Table 3.4: Example of attributes and information of passing locator stored to Locator DB

Locator	Locator data stored in JSON format

id:theme-toggle	"2": {
	"tag": "BUTTON",
	"class": "text-gray-500 dark:text-gray-400 hover:bg-gray-100
	"type": "button",
	"name": "",
	"id": "theme-toggle",
	"value": "",
	"title": "",
	"x": 1161,
	"y": 15,
	"width": 40,
	"height": 40,
	"children": 2,
	"xpath": "/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[4]",
	"idxpath": "//*[@id='theme-toggle']",
	"locator": "id:theme-toggle"
	}

3.3.9 Weak Candidate Method

I started testing candidate methods by implementing first the Weak Candidate Method. I aimed to avoid additional database integrations. Initially, I attempted to read the HTML page source, split it into words, and use the RapidFuzz InDel distance algorithm to get a similarity ratio for different words. This approach allowed me to produce a list of potential candidate elements for self-healing. For example, in the case of "id:theme-toggle" this approach worked like a charm, see Table 3.5, but on many other occasions not so well.

Table 3.5: Example of Failing locator and candidates with similarity ratios

Decsription	Locator + (Similarity ratio in candidates)
Failing locator	id:theme-toggle
Candidate 1	id='theme-toggler':84.85
Candidate 2	id='theme-toggle-dark-icon':66.67
…	…
Candidate 10	sidebar-toggle-item=">PRO:48.78

How the Weak Candidate Method was included in the self-healing mechanism process flow can be observed in Figure 3.5. The Weak Candidate Method had only two main functionalities: to collect candidates by

retrieving the HTML page source from the SUT and to create the top 10 candidates using the InDel distance algorithm by comparing the failing locator to those retrieved from the SUT. The InDel distance algorithm functionality is achieved through a Python package called RapidFuzz, which is a variation of the Levenshtein distance algorithm [62].

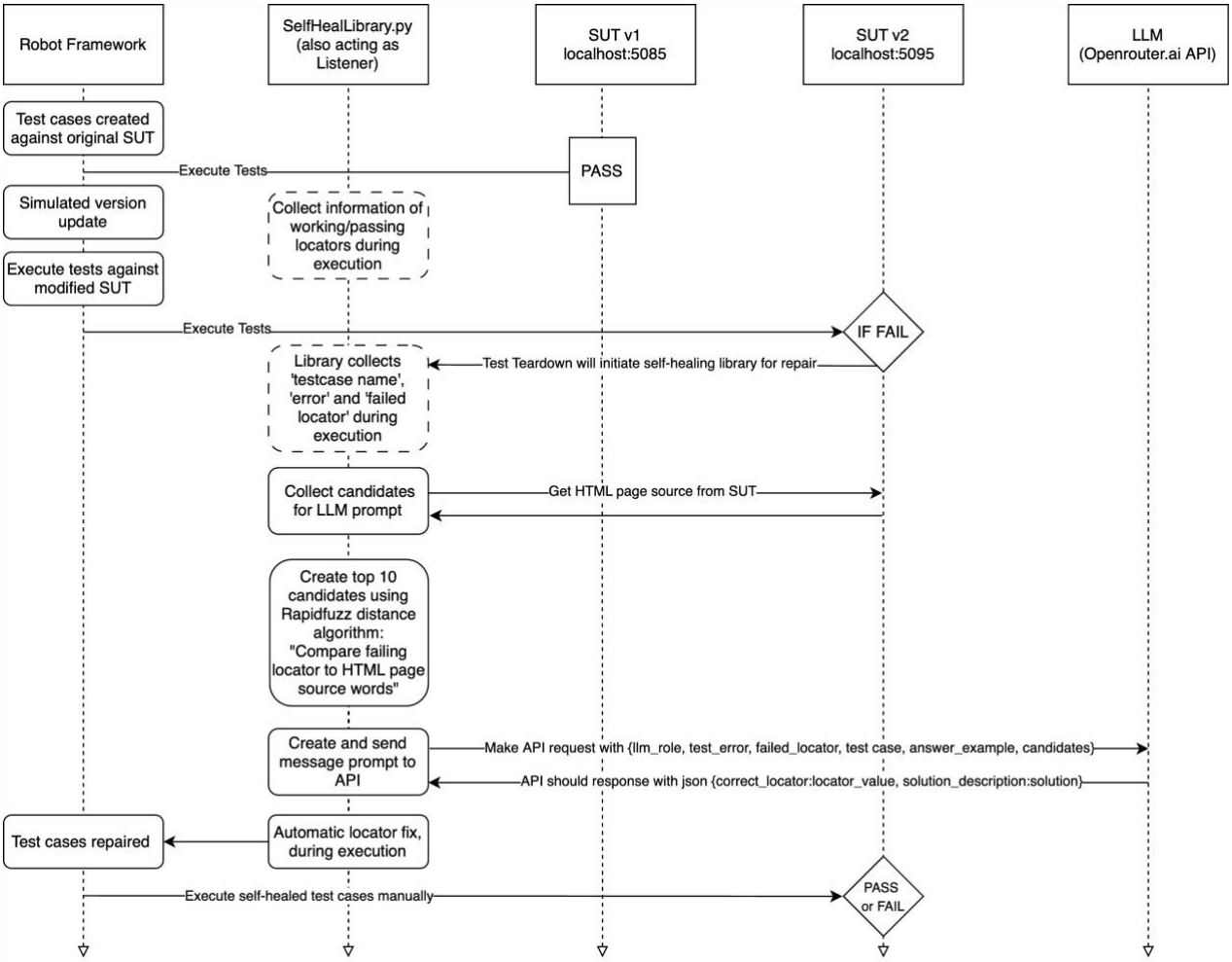


Figure 3.5: Self-healing mechanism process flow when using the "Weak Candidate Method"

3.3.10 Strong Candidate Method

The Strong Candidate Method retrieves locator information stored in the Locator Database for the failed locator. I used the locator assigned to the test case as the ID that would match in the Locator Database. After obtaining all relevant information for the failing locator, the self-healing mechanism executes a JavaScript command in the SUT to retrieve all visible element locators along with their relevant information and attributes. It then compares the failed locator information to all visible locators using the Levenshtein distance algorithm with a sum weight of 2, provided by the RapidFuzz Python library [62]. The failing locator and candidates derived from earlier steps in JSON format can be observed in Table 3.6

Table 3.6: Example of failing locator and candidates with similarity ratios

Type	**Locator stored in JSON format, (candidates include similarity ratio as last value)**
Failing locator	{'tag': 'BUTTON', 'class': 'text-gray-500 dark:text-gray-400 hover:bg-gray-100 dark:hover:bg-gray-700 focus:outline-none focus:ring-4 focus:ring-gray-200 dark:focus:ring-gray-700 rounded-lg text-sm p-2.5', 'type': 'button', 'name': '', 'id': 'theme-toggle', 'value': '', 'title': '', 'x': 1161, 'y': 15, 'width': 40, 'height': 40, 'children': 2, 'xpath': '/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[4]', 'idxpath': "//*[@id='theme-toggle']", 'locator': 'id:theme-toggle'}
Candidate 1	{'tag': 'BUTTON', 'class': 'text-gray-500 dark:text-gray-400 hover:bg-gray-100 dark:hover:bg-gray-700 focus:outline-none focus:ring-4 focus:ring-gray-200 dark:focus:ring-gray-700 rounded-lg text-sm p-2.5', 'type': 'button', 'name': '', 'id': 'theme-toggler', 'value': '', 'title': '', 'x': 1161, 'y': 27, 'width': 40, 'height': 40, 'children': 2, 'xpath': '/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[4]', 'idxpath': '//*[@id='theme-toggler']'}:95.93
Candidate 2	{'tag': 'BUTTON', 'class': 'flex text-sm bg-gray-800 rounded-full focus:ring-4 focus:ring-gray-300 dark:focus:ring-gray-600', 'type': 'button', 'name': '', 'id': 'user-menu-button-2', 'value': '', 'title': '', 'x': 1213, 'y': 31, 'width': 32, 'height': 32, 'children': 2, 'xpath': '/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/div[5]/div[1]/button[1]', 'idxpath': '//*[@id='user-menu-button-2']'}:76.99{'tag': 'BUTTON', 'class': 'p-2 text-gray-500 rounded-lg hover:text-gray-900 hover:bg-gray-100 dark:text-gray-400 dark:hover:text-white dark:hover:bg-gray-700', 'type': 'button', 'name': '', 'id': '', 'value': '', 'title': '', 'x': 1081, 'y': 27, 'width': 40, 'height': 40, 'children': 2, 'xpath': '/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[2]', 'idxpath': '/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[2]'}:73.51
…	...
Candidate 10	{'tag': 'A', 'class': 'inline-flex justify-center p-2 text-gray-500 rounded cursor-pointer hover:text-gray-900 hover:bg-gray-100 dark:hover:bg-gray-700 dark:hover:text-white', 'type': '', 'name': '', 'id': '', 'href': 'http://localhost:5095/#', 'title': '', 'x': 53.5, 'y': 686.5, 'width': 40, 'height': 40, 'children': 1, 'xpath': '/html[1]/body[1]/div[1]/aside[1]/div[1]/div[2]/a[1]', 'idxpath': '//*[@id='sidebar']/div[1]/div[2]/a[1]'}:66.09

How the Strong Candidate Method was included into the self-healing mechanism process flow can be observed in Figure 3.6. The Strong Candidate Method had two main functionalities. First, it collected information on working/passing locators during test case executions. If keywords/locators passed, it would retrieve other relevant information and attributes from the SUT using a JavaScript command, which extracts all the necessary data from the element locator. To avoid duplicates, it would only store new information if the element locator information was not already stored.

The self-healing process involved first fetching attributes and information of the failed locator from the Locator Database and then retrieving information and attributes of all visible locators in the SUT. It would then create the top ten candidates for the LLM using the InDel distance algorithm, the same as used in the Weak Candidate Method.

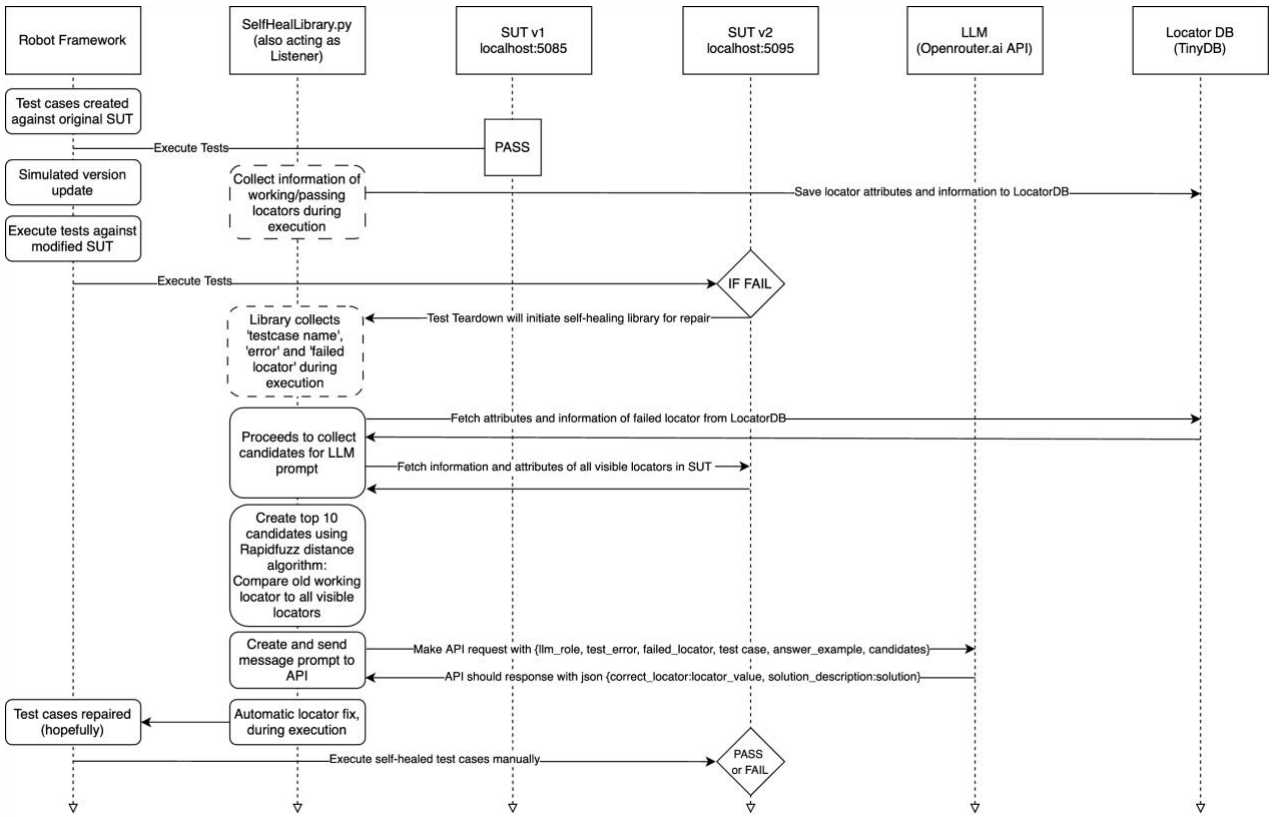


Figure 3.6: Self-healing mechanism process flow when using the "Strong Candidate Method"

3.3.11 Self-healing Library

For the self-healing mechanism, I created an internal Python library called "SelfHealLibrary.py," which was placed in the libraries folder of the Robot Framework project structure. It contained the following basic functionalities: collecting candidate locators for the LLM, creating the LLM prompt, logging, sending requests to the Openrouter.ai API, and repairing code based on the suggestions.

The library also functions as a listener, enabling the collection of crucial information about the test case during execution. It implements the listener keywords "start test" and "end library keyword." The "start test" listener keyword is used solely to record the test case name and save it as a Python variable. The "end library keyword" listener keyword has either one or two functions. If the Weak Candidate Method is selected, it will collect the test error message and set the failed locator into a Python variable if a failing keyword is detected, as seen in Figure 3.5. If the Strong Candidate Method is selected, it has two different purposes depending on the keyword results, as shown in Figure 3.6. If a keyword fails, it performs the same functions as in the Weak Candidate Method, collecting the test error message and setting the failed locator into a Python variable. However, when a keyword passes, it checks if the keyword was called from SeleniumLibrary and whether the keyword had a locator in use. If it did, it fetches information and attributes from the SUT for that specific locator. These two keywords are the listener part of the library.

The main keyword of this self-healing library is "Make Request For LLM," which is called from Robot Framework test cases in the Test Teardown part, based on the test status. If a test fails, a request is made to the LLM, if the test passes, nothing happens. It is crucial to set this "Make Request For LLM" keyword

before closing the browser, as it will collect potential candidate locators based on the selected Weak or Strong Candidate Method from the SUT. After that, it proceeds to create the message for the LLM, appending our LLM prompt message with the test error message, test case name, failed locator attributes and information, and the top 10 candidates. Next, it makes a request to the Openrouter.ai API with the specific LLM selected as the model and the LLM prompt created as the message payload. The API should respond with the LLM's response containing the correct locator and solution description in JSON format. The final step is to find and replace the locator in the Robot Framework test scripts with the correct locator provided by the LLM. This process is illustrated in the diagram in Figure 3.7.

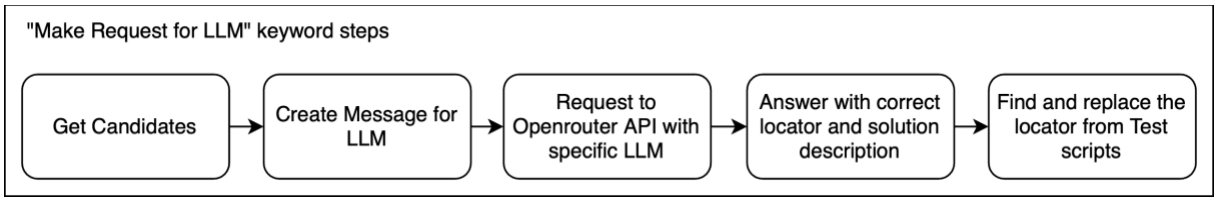


Figure 3.7: Make Request. for LLM keyword steps

3.3.12 Openrouter.ai API for LLM

I used the Openrouter.ai API to make requests to different LLMs. This allowed me to easily test a multitude of different models by simply changing the model name in the test automation scripts. Openrouter.ai provides a standardized API that requires no changes in the code when switching between models or providers. Openrouter allows usage through OpenAI's client API, making the process simple. I only needed to generate my own API key, include the model and messages in the LLM prompt, and I was ready to send requests. Pricing is automatically based on your API key usage.

The cost of using Openrouter.ai was very affordable. It allowed me to monitor credit usage from their site. I made a total of 311 requests to the Openrouter.ai API for different LLMs during this thesis, and the total cost was $2.14.

3.3.13 Prompt Engineering for LLM

Initial prompt messages were made based on prior work on using LLM to repair test cases by Xu et al [14]. First version of prompt messages can be observed from following Table 3.7. I thought that with this one I could get sufficient answers from LLMs, but the results were not what I initially expected. I noted that you have to be very precise when making prompts for LLMs, and in this case when the "answer" format was too broad and the answers varied.

Table 3.7. First version of LLM prompt

Description	Role	Content
LLM Role	system	You are a Robot Framework GUI test script repair tool.
Test Error message	user	Robot Framework Test using SeleniumLibrary failed with following error message: + TEST_ERROR_MSG
Broken locator in Robot Framework scripts	user	Locator used: + BROKEN_LOCATOR
Test Case Name	user	Robot Framework Test Name: + TEST_CASE
Candidates	user	Here is a list of 10 potential locator candidates with similarity ratio values: + CANDIDATES
Answer	user	"Please, give your answer as the correct element locator value. "
		"Example: correct_locator: locator_value"
		"After that, make line change and tell how you ended up in the solution."

After carefully fine-tuning my prompt, the final version looked like the one presented in Table 3.8. The role remained the same, instructing the LLM to act as a Robot Framework GUI test script repair tool. Following this, I included the test context-related messages: the test error message, the failing locator, and the test case name. This was followed by a list of 10 potential candidates provided by the candidate methods. The last message in the prompt was the answer format and an example. Providing a clear and precise example and instructing the LLM to respond in JSON format made a huge difference. Almost all the answers after this change were in the same format, regardless of which LLM was used.

Table 3.8: Final version of LLM prompt with Weak Candidate method

Description	Part of message
Giving a role for LLM	{'role': 'system', 'content': 'You are a Robot Framework GUI test script repair tool.'}
Test Error message	{'role': 'user', 'content': "Robot Framework Test using SeleniumLibrary failed with following error message: Element 'id:theme-toggle' did not appear in 5 seconds."}
Broken locator in Robot Framework scripts	{'role': 'user', 'content': 'Broken element locator used: id:theme-toggle'}
Test case name	{'role': 'user', 'content': 'Robot Framework Test Name: Theme Can Be Toggled To Dark Mode'}
Candidates	{'role': 'user', 'content': "List of 10 potential locator candidates which was retrieved from tested web page with similarity ratio values: id='theme-toggler':84.85id='theme-toggle-dark-icon':66.67id='theme-toggle-light-icon':65.12id='tooltip-toggle':58.82sidebar-toggle-item=":54.05md:items-center:53.33items-center:51.85sidebar-toggle-item=">UI:50.00sidebar-toggle-item>Server:48.78sidebar-toggle-item=">PRO:48.78"}
Giving example on how LLM should answer	{'role': 'user', 'content': 'Please, give your answer in json format, without json suffix. In example replace locator_value with your suggestion.Example: {correct_locator: locator_value, solution_description: solution}'}

4 Results

The self-healing mechanism, utilizing both the Strong and Weak Candidate Methods, was executed once with each selected Large Language Model (LLM) for the entire test set. Initially, tests were run with the self-healing mechanism enabled. After automatic code changes were made based on the LLM responses, the test cases were rerun to obtain the final results. Due to this test setup, the locator suggestion made by LLM had to be precise to be able to repair the test case. The results included many near-correct answers, most of which contained issues with the presence of extra quotes in the locator's syntax. An example of these extra quotes could be provided by presenting the correct locator change *id:theme-toggler* and the near correct locator change *id:'theme-toggler'*.

Exact results, including examples and repaired locators, can be found in Appendix 7. The locators used, particularly XPath locators, are not always explicit and may vary in some answers while still being correctly repaired. This is because element locators can use multiple strategies to identify specific elements. All responses from the LLMs have been manually analyzed, and the answers have been recorded in the replication set [64].

Table 4.1: Presenting test case ID, locator strategy and locators originally and altered versions

ID	Locator Strategy	Locator originally	Locator modification
A1	Element text	”PRO File Manager”	”File Manager”
A2	Element id	id:theme-toggle	id:theme-toggler
A3	Link text	Starter Page	Start Page
A4	Element name	name:script	name:scripts
A5	Button type	type:submit	type:button
A6	Element class	class:sr-only	class:search-only
S1	Remove DOM element	-	Removed one DIV from Dashboard
S2	DOM Element position switch	Order of navigation elements [DataTables, Charts, API, ..]	Order of navigation elements [API, DataTables, Charts, ..]

Both Llama models provided answers in a different JSON style than the others, which caused problems on the Python code side. Suggested locators for both Llama LLMs were manually changed in the code, and the test cases were run manually afterward to obtain results for those models.

4.1 Self-healing Results with Weak Candidate Method

When using the Weak Candidate Method, the best element locator repair accuracy was achieved by using GPT-4 Turbo, which successfully repaired test cases A2, A5, and S1, surpassing all other LLMs in comparison. Mistral 7B Instruct and Llama 3 8B Instruct were more prone to hallucinations than others in

the comparison between different test cases. In particular, Mistral 7B Instruct struggled to suggest anything close to correct. A list of correct locator repairs can be seen in Table 4.2.

LLM-wise, ChatGPT-4 Turbo correctly repaired the locator for 37.5% of the test cases. Claude 3 Sonnet and Mixtral 8x22B Instruct followed, each repairing 25% of the test cases. Llama 3 8B Instruct, Llama 3 70B Instruct, and Mistral Large each managed to repair 12.5% of the test cases correctly. GPT-3.5 Turbo and Mistral 7B Instruct did not repair any test cases using the Weak Candidate Method.

Test case-wise, the best results were achieved with test case A2, which used ID as the locator strategy. Four out of eight LLMs correctly repaired test case A2. The other four LLMs were close to repairing test case A2 but made minor syntax mistakes, resulting in failure. Similar cases, where repairs were almost successful but had syntax errors, were seen in test cases A4 and A5. Test case A5 was correctly repaired by GPT-4 Turbo, Mixtral 8x22B Instruct, and Mistral Large. Test case S1 was correctly repaired by GPT-4 Turbo and Llama 3 70B Instruct. Test case S2 was correctly repaired only by Claude 3 Sonnet. Test cases A1, A3, A4, and A6 were not correctly repaired by any of the LLMs.

Table 4.2: Self-healing results when using the Weak Candidate Method

	Claude 3 Sonnet	**GPT-3.5 Turbo**	**GPT-4 Turbo**	**Llama 3 8B Instruct**	**Llama 3 70B Instruct**	**Mistral 7B Instruct**	**Mixtral 8x22B Instruct**	**Mistral Large**
A1	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
A2	**Pass**	Fail	**Pass**	**Pass**	Fail	Fail	**Pass**	Fail
A3	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
A4	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
A5	Fail	Fail	**Pass**	Fail	Fail	Fail	**Pass**	**Pass**
A6	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
S1	Fail	Fail	**Pass**	Fail	**Pass**	Fail	Fail	Fail
S2	**Pass**	Fail	Fail	Fail	Fail	Fail	Fail	Fail
Correct Repairs	2/8 (25 %)	0/8 (0 %)	3/8 (37,5%)	1/8 (12,5%)	1/8 (12,5 %)	0/8 (0 %)	2/8 (25 %)	1/8 (12,5%)

4.2 Self-healing Results with Strong Candidate Method

When using the Strong Candidate Method, the best element locator repair accuracies were achieved by using GPT-4 Turbo and Mistral Large. These models were able to repair test cases from A2 to S2, achieving locator repair rates of 87.5%. Llama 3 70B Instruct was close, being able to correct six locators in total. Mistral 7B Instruct on the other hand struggled and could not repair any of the locators in the test case set. A list of correct locator repairs can be seen in Table 4.3.

Overall, results when using the Strong Candidate Method were much more accurate than when using the Weak Candidate Method. A correct locator repair rate of over 50% was achieved by five different Large Language Models, with GPT-4 Turbo and Mistral Large leading the pack at 87.5% accuracy, followed by Llama 3 70B Instruct with 75% accuracy, and GPT-3.5 Turbo and Llama 3 8B Instruct each achieving 50% accuracy.

Test case-wise, the best results were achieved when repairing test cases A2 and A6, which used ID and class as locator strategies. Additionally, test cases A4 and S1 were correctly repaired by five different LLMs. Test case A4 used the element's name as a locator, and in test case S1, had a DOM element was removed from the SUT. Test case A1 was not successfully repaired by any of the LLMs in comparison.

Table 4.3: Self-healing results when using the Strong Candidate Method

	Claude 3 Sonnet	**GPT-3.5 Turbo**	**GPT-4 Turbo**	**Llama 3 8B Instruct**	**Llama 3 70B Instruct**	**Mistral 7B Instruct**	**Mixtral 8x22B Instruct**	**Mistral Large**
A1	Fail	Fail	Fail	Fail	Fail	Fail	Fail	Fail
A2	**Pass**	**Pass**	**Pass**	**Pass**	**Pass**	Fail	**Pass**	**Pass**
A3	Fail	Fail	**Pass**	Fail	Fail	Fail	**Pass**	**Pass**
A4	Fail	**Pass**	**Pass**	**Pass**	**Pass**	Fail	Fail	**Pass**
A5	Fail	Fail	**Pass**	Fail	**Pass**	Fail	Fail	**Pass**
A6	**Pass**	**Pass**	**Pass**	**Pass**	**Pass**	Fail	**Pass**	**Pass**
S1	Fail	**Pass**	**Pass**	**Pass**	**Pass**	Fail	Fail	**Pass**
S2	Fail	Fail	**Pass**	Fail	**Pass**	Fail	Fail	**Pass**
Correct repairs	2/8 (25%)	4/8 (50%)	7/8 (87,5%)	4/8 (50%)	6/8 (75%)	0/8 (0 %)	3/8 (37,5%)	7/8 (87,5%)

4.3 Cost Per Test Set Run Using Openrouter.ai

By using Openrouter.ai, I was able to track the pricing of different Large Language Models and provide approximate self-healing costs per test set (tests from A1 to S2) for each LLM used. The prices for running the entire test set using the Weak Candidate Method can be found in Table 4.4, while prices for the Strong Candidate Method are in Table 4.5. The price difference is due to the length of candidates provided to the LLM, which increases the cost of the input prompt, as the LLM outputs were similar in length most of the time. The cost difference is about three times lower for the Weak Candidate Method than for the Strong Candidate Method.

When using the Weak Candidate Method, GPT-4 Turbo and Mistral Large were the most expensive to use. However, when switching to the Strong Candidate Method, Mistral Large became the costliest of the compared LLMs. Overall, the cheapest LLM for self-healing locators was Llama 3 8B instruct.

Table 4.4: Pricing for running the whole test set (8/8) with selected LLM when using the Weak Candidate Method

Number of runs	Claude 3 Sonnet	GPT-3.5 Turbo	GPT-4 Turbo	Llama 3 8B Instruct	Llama 3 70B Instruct	Mistral 7B Instruct	Mixtral 8x22B Instruct	Mistral Large
1x	$0.02	$0.002	$0.04	$0.0002	$0.002	$0.0004	$0.003	$0.04
100x	$2	$0.2	$4	$0.02	$0.2	$0.04	$0.3	$4

Table 4.5: Pricing for running the whole test set (8/8) with selected LLM when using the Strong Candidate Method

Number of runs	Claude 3 Sonnet	GPT-3.5 Turbo	GPT-4 Turbo	Llama 3 8B Instruct	Llama 3 70B Instruct	Mistral 7B Instruct	Mixtral 8x22B Instruct	Mistral Large
1x	$0.07	$0.007	$0.15	$0.001	$0.02	$0.002	$0.015	$0.17
100x	$7	$0.7	$15	$0.1	$2	$0.2	$1.5	$17

4.4 Potential Design Flaw

Test case A1 was not correctly repaired by any of the LLMs using either the Weak or the Strong Candidate Method. This might be due to a flaw in the combination of test case and LLM prompt syntaxes. If we look at the syntax for test case A1 in Figure 4.1, we can observe that the test case uses the variable "${FILE_MANAGER_LOCATOR}" as the locator, which should contain the text "PRO File Manager." The locator itself can be seen in Figure 4.2.

The change made to the SUT was renaming "PRO File Manager" to "File Manager." So, I am not changing the locator but the text that should be found by the locator. The element locator is correct and should not be changed. The problem is that in my LLM prompt, I specifically ask for the locator to be repaired by stating: "Please, give your answer in JSON format, without JSON suffix. For example, replace locator_value with your suggestion. Example: {correct_locator: locator_value, solution_description: solution}." The changes should be made to the text, and since I am not directly asking for that, the LLMs are struggling in this case.

```
File Manager Visible At Sidebar
    [Documentation]    A1
    ...                SUT Change: Naming "PRO File Manager" to "File Manager"
    [Tags]    A1
    Wait Until Page Contains Element    ${FILE_MANAGER_LOCATOR}
    Element Text Should Be    ${FILE_MANAGER_LOCATOR}    PRO File Manager
```

Figure 4.1: Test Case A1 syntax

Figure 4.2: File Manager Locator Variable

This leaves little to no room for a correct repair to be possible for this test case. However, the solution descriptions from three LLMs suggest that the locator was correct and other approaches should be taken to repair this test case. GPT-4 Turbo's solution description included the following: "*No change is required in the locator. It points accurately to the sidebar element where the 'File Manager' text is seen. Instead, check for updates or changes in the text content within the application. The expected text should be updated to 'File Manager' if the application now reflects this new text instead of 'PRO File Manager'*."

Similar solution descriptions were provided for test case A1 by Mixtral 8x22B Instruct and Claude 3 Sonnet. Mixtral's solution description was very direct: "*The locator seems to be correct based on the provided information. However, the expected text in the test case is 'PRO File Manager' while the actual text is 'File Manager'. You should update the expected text in your test case to 'File Manager' to resolve the issue*."

In the end, as I see there is still a small possibility to get this case corrected with the help of an LLM, I left it as part of this accuracy evaluation. The rest of the solution descriptions are in the thesis replication set [64].

5 Discussion

5.1 The Concept of Self-healing Mechanism

I envisioned this thesis as a proof-of-concept, which would present the enablement of self-healing locators for Robot Framework test cases. There exist prior studies where practitioners have used LLMs to repair failing locators, but this might be one of the first studies where this self-healing mechanism for locators is enabled in the Robot Framework context. My self-healing mechanism relies on the Robot Framework architecture which allows the creation of new custom testing libraries. These libraries can be integrated easily and can extend existing testing environments created with Robot Framework.

The Self-Healing Library created heavily depends on the Robot Framework's Listener API architecture. This allowed me to perform the self-healing process while the test case itself was still executing. All the steps for self-healing occurred in real time and had no significant effect on the length of the test case. The default timeout for Robot Framework SeleniumLibrary keywords when not finding the correct locator is 5 seconds. This self-healing mechanism took a total of 1-3 seconds to create the request for the LLM and repair the locator.

This concept is definitely not fully developed. I see a need for additional enhancements to the self-healing library code, including the addition of a self-correction mechanism for LLM responses, automatic reruns for test cases, and automatic commits and pushes to the version management system. Furthermore, the current level of LLM prompting does not adequately serve all types of test keywords. I would suggest that keywords which directly use locators, for example, checking if elements exist, should have a different prompt than those which check for specific text in an element with a given locator. This issue was highlighted by the inability of LLMs to repair test case A1.

5.2 Accuracy of LLMs in Repairing Locators

Overall, I think that the accuracy of correct repairs from LLMs are on par with prior studies related to this subject. Although, achieving that was not the real goal of this study. The Accuracy of both GPT-4 Turbo and Mistral Large was 87,5%, but it is actually a full 100%, because test case A1, is not repairable with the given LLM prompt instructions.

I had hypothesis that LLM could be able to repair locators without history knowledge of the locator attributes and information. Based on my results, this is possible but not practical in a real-world context. Without using candidate methods, I was able to get GPT-4 Turbo to correctly repair one test case, S2, where elements had switched positions. Implementing the Weak Candidate Method improved the results slightly. With the Weak Candidate Method, GPT-4 Turbo was able to repair three locators in total, but the overall results were underwhelming. GPT-3.5 Turbo, Llama 3 70B Instruct, and Mistral 7B Instruct were not able to make any

correct repair suggestions with the Weak Candidate Method.

The best accuracy was achieved with the Strong Candidate Method, which had access to a locator database containing locator attributes and information that could be compared against the current SUT. Using this method, both versions of GPT, both versions of Llama, and Mistral Large were able to achieve over a 50% repair rate.

The most surprising result was the accuracy of Llama 3 8B Instruct with the Strong Candidate Method. Llama 3 8B Instruct was the cheapest model used in this study, as shown in Tables 4.3 and 4.4, yet it achieved higher repair rates than Claude 3 Sonnet and Mixtral 8x22. This could indicate that Llama 3 8B Instruct may have some additional knowledge about Robot Framework or locator syntax that the other models lacked.

5.3 Locator Types

From the results collected when running the self-healing mechanism with the Strong candidate method, we noticed that some locators and test cases were repaired more frequently than others. Test cases and locators that were more likely to be repaired by the self-healing mechanism with different LLMs were A2, A4, A6, and S1. Of these, three were attribute-based, and one was structure-based.

When using Strong candidate method, test case A2, which used “id” as a locator, was repaired a total of 7 times out of 8 by different LLMs. A4, which used “name” as a locator, was repaired 6 times out of 8. A6, which used “class” as a locator, was repaired 5 times out of 8. Test case with id S1, using XPath where one div element was removed from the SUT, was repaired 5 times out of 8. This could indicate that other locator types, like “link” and “type,” and changing text element positions are too difficult for smaller LLMs to repair correctly. The fact that S2, where text elements changed positions, was repaired only by the high-level models (GPT-4 Turbo, Mistral Large, and Llama 3 70B Instruct) suggests that these models may have more contextual capabilities to understand what has happened and not just directly try to repair the locator.

5.4 LLM Prompting

During the development of this self-healing mechanism, I found that LLM prompting is a major factor for success. Based on prior studies by Xu et al. and Nass et al., I created my own LLM prompt. I'm not entirely sure if all the lines in the prompt I currently send to the LLM are necessary. However, when evaluating the current set of different instructions, I don’t know what to leave out. I see that it is important to give the LLM a role, and provide the test case context by including the test case name, test error message, and the failing locator. Locator candidates are a must, and giving an example of how the LLM should respond is essential.

Initially, I had a version where LLM answers were not instructed to be in JSON format. That led to a situation where responses from the LLM varied, making it impossible to create functional code that could parse the correct locator from those answers. After providing direct instructions and an example of how to answer in JSON format, the quality of responses improved. Only a couple of LLMs struggled with consistently

providing JSON responses, both struggling models were Llamas, 8B, and 70B Instruct.

5.5 Candidate Methods

Based on my results the context that LLMs requires are either the whole HTML page source or some kind of candidate locator which to match the failing locator. Sending the whole HTML page for the LLM was not tried in this thesis, so I can't directly suggest that using candidate methods for a self-healing locator is a must. But, it definitely cuts costs from the LLM prompts sent by reducing the token count from both input and LLM output.

Although a candidate method without locator history knowledge was possible, I would use historical information when trying to self-heal locators automatically with LLMs.

5.6 LLM Assisted Test Automation Maintenance

To achieve optimal results in maintaining test automation, it is essential to perform regular updates, as one of the main causes of test failures is broken element locators. We should consider utilizing Large Language Models (LLMs) for maintaining test automation. Based on the results, leading LLMs like GPT-4 Turbo and Mistral Large are highly capable of repairing element locators. This approach necessitates enhancements to the current self-healing mechanism, such as automatic reruns and version management functionality for repaired locators.

Referencing Tables 4.3 and 4.4, using the GPT-4 Turbo model for self-healing an entire test set (8 locators/test cases) 100 times with the Strong candidate method would cost approximately $15. Thus, repairing 800 locators would cost $15. For other accurate models like Llama 3 70B Instruct and Llama 3 8B Instruct, the cost for self-healing 800 locators would be $2 and $0.10, respectively. If self-healing is performed daily for 800 locators, the annual cost would be: GPT-4 Turbo: $15 x 365 = $5475, Llama 3 70B Instruct: $2 x 365 = $730 and Llama 3 8B Instruct: $0.10 x 365 = $36.50. According to a survey made by Koodiklinikka, 2022 [65], the average hourly rate for software development consultancy is about 93.50€, approximately $101.50.

Without using a self-healing mechanism, a test engineer must perform the following tasks to maintain a single locator: read test logs, find the failing test case, analyze the issue, open the system under test, navigate to the failing locator page, analyze what is wrong with the locator, change the locator in the test scripts, rerun test cases, and push changes to version management. This process is the basic flow if everything goes smoothly. Even if only a single locator is changed weekly, it could lead to costs like 52 x $101.50 = $5,278. This is comparable to the cost of using GPT-4 Turbo with a self-healing mechanism for (800 locator repair tries per day) x 365 days = 292,000 times a year for a single locator.

If we flip these numbers and consider what we would get with one hour of consultancy work, which costs approximately $101.50, we would get approximately 5410 locator repair attempts with GPT-4, 40600

attempts with Llama 3 70B Instruct, and a whopping 812000 locator repair attempts with Llama 3 8B Instruct. Of course, some of these repairs would not work, and the test cases and locators could not be repaired, but even with 50% accuracy, this could be highly beneficial to implement.

5.7 The Flaw in Test Case Design

My experimental design had a flaw in the test cases I created, specifically in Test Case A1, which is almost impossible to repair with the given LLM prompt and test setup. I discovered this issue after attempting self-healing using the Strong Candidate Method. Despite this, I decided to keep the test runs and results as they were, primarily because I found it interesting to see if any of the LLMs could produce a correct solution for that particular test case. However, it turned out that no correct repairs were made this time for Test Case A1.

What was more delightful was the fact that three of the LLMs tested, Claude 3 Sonnet, GPT-4 Turbo, and Mixtral 8x22B Instruct were able to identify and explain that the locator was actually correct, but the text was not, and saying that a different approach was needed to repair the test case. This demonstrates the impressive contextual awareness of these models, which was amazing to discover.

6 Conclusions

6.1 Answering Research Questions

This study aimed to identify ways to enable self-healing locators for Robot Framework test cases. I wanted to determine how self-healing locators could be implemented in the Robot Framework context, what the requirements are for doing so, whether some types of locators are easier for LLMs to repair, and how accurate different LLMs are at making these repairs. The study was conducted in an experimental fashion, where a self-healing library was built and the self-healing of Robot Framework test case locators was tested with a set of different LLMs.

My main research question was to find out (RQ1): How can self-healing locators be enabled in Robot Framework test cases with the assistance of Large Language Models? I presented a solution for self-healing locators in Robot Framework, which utilized a custom-made Robot Framework library. This library used Robot Framework's internal listener APIs to collect the necessary information for locator repairs and to create candidate locators in two different ways. It employed the Openrouter.ai API to communicate with multiple LLMs and autonomously repair locators during test execution.

The second research question was (RQ2): What kind of locators in Robot Framework test cases have the most potential to be repaired with the assistance of LLMs? According to my results, larger LLMs like GPT-4, Mistral Large, and Llama 3 70B Instruct are capable of repairing a variety of locator types used in Robot Framework with SeleniumLibrary. Almost all of the LLMs that were compared could provide repair for test case id A2, where the locator was changed from “id:theme-toggle” to “id:theme-toggler”. Although this repair was simple, it suggests that using ID as a locator could be beneficial in the long run when using self-healing locators.

The tested locator types included six different element attributes: element text, id, link text, name, type, and class. Additionally, there were two different structural changes in the locators: one where an element was removed from the page and another where element positions were switched. GPT-4 and Mistral Large were able to repair 7 out of 8 test cases, the only one that couldn’t be repaired was test case A1, where the element text was changed. This test could not be repaired by any other LLMs either, indicating that the problem was in the test setup and LLM prompt, not in the LLM capabilities. Solution descriptions provided by Mixtral 8x22B Instruct, GPT-4, and Claude 3 Sonnet suggested that the locator itself in test case A1 was correct and that the repair should be done differently in the test case. This shows great contextual awareness from the mentioned LLMs.

The third research question was (RQ3): What are the requirements for LLMs to assist in repairing locators? Based on the results and findings of this study, I highly suggest that LLM self-healing requires instructed LLM prompting and the use of a candidate algorithm method, which provides clear choices for the LLM. By giving LLMs instructed commands in the input message prompt, we can get answers in a manner that

enables direct changes to Robot Framework scripts, thus enabling the self-healing of locators. The results suggest that using some kind of locator candidate algorithm is essential to achieve self-healing of locators with the help of LLMs. Based on my findings, without provided candidates, self-healing was not possible in a way that could be implemented for further usage.

The fourth research question was (RQ4): How accurate are LLMs when suggesting repairs for Robot Framework locators? Based on my results, accuracy is, in most cases, correlated with the size and cost of the LLM. The accuracy of GPT-4, Mistral Large, and Llama 3 70B Instruct was great, with results suggesting that GPT-4 is the best candidate for further research in this matter. A similar conclusion was reached by Nass et al. [49]. Mistral 7B Instruct was not able to repair any of the locators using either the Weak or the Strong Candidate Method and hallucinated more than any other LLM. Llama 3 8B and 70B Instruct had problems answering with the correct JSON format, which affected the way results were collected from those LLMs answers.

Prior studies by Nass et al. [13] and Xu et al. [14] used LLMs with version history information for the locators. This study suggests that it is possible to achieve some level of self-healing locators without knowledge of prior locator versions, although the capability is not on par with those using locator history information. Overall, the results with the best performing LLMs exceeded my expectations in the self-healing of locators. These results provide valuable insights into how this self-healing mechanism could be implemented in Robot Framework and within a business environment.

6.2 Limitations

Although the test cases created and tested in this study simulated real-life test scenarios for multiple locator types, this also posed a limitation in terms of the results. The number of test cases used doesn't demonstrate how accurate some of the LLMs would be if larger test sets were introduced.

Additionally, the method we used to extract attributes and information from SUT webpage elements should be revised in the future. The solution used in this thesis employed the same JavaScript command as Nass et al. [13]. There is potential for improvement by adjusting which attributes and information of the commands are necessary and determining which should be used to achieve the best results from the LLMs.

A potential threat to the validity in this thesis is the small sample size of different test cases created. The test cases were created according to the Robot Framework SeleniumLibrary locator strategy and were implemented based on the test breakage taxonomy presented. Eight different test cases were made to match each strategy to the corresponding test breakage type. The number of test cases used does not provide the best accuracy results regarding the abilities of the LLMs compared in this study, but it does offer a rough estimate of what is possible.

6.3 Future Research

Based on these conclusions, practitioners should consider whether fine-tuning LLM prompting could further improve results. By inspecting the results of the test case with ID A1, it was evident that the test setup and the LLM prompt made it impossible to repair in the first place. I suggest that different Robot Framework test keyword types could benefit from different prompt styles.

Further research is needed to determine if we could achieve improvements in self-healing locators in Robot Framework test cases using different LLMs with self-correcting prompts, as used by Xu et al. [14]. Self-correction was not within the scope of this study. I believe it requires substantial new logic on the Robot Framework library side and for the LLM prompts. However, it could enhance the accuracy of some LLMs, as the results indicated that some repairs were nearly correct or very close to being correct. This suggests that with minor adjustments and more instructions, those repairs could eventually be correct.

Lastly, the current solution requires manual intervention after self-healing is done and the code is changed according to LLM locator suggestions. Currently, the user has to run tests manually after self-healing to get the results for the repaired test cases. However, in Robot Framework, there are ways to run failed test cases directly after a test case has failed, which would automate this process. Further development could also include automatic version management, where a subprocess could command, for example, Git to commit changes and push them to the version management system.

References

[1] S. M. K. Quadri and S. U. Farooq, "Software Testing - Goals, Principles, and Limitations," *International Journal of Computer Applications (0975 - 8887),* vol. 6, no. 9, 2010.

[2] M. Hammoudi, "Why Do Record/Replay Tests of Web Applications Break?," University of Nebraska, Computer Science and Engineering: Theses, Dissertations, and Student Research, Lincoln, 2016.

[3] E. Alégroth, R. Feldt and P. Kolström, "Maintenance of Automated Test Suites in Industry: An Empirical study on Visual GUI Testing," arXiv:1602.01226v1, 2016.

[4] M. Nass, E. Alégroth and R. Feldt, "Why many challenges with GUI test automation (will) remain," *Information and Software Technology,* vol. 138, no. 106625, 2021.

[5] J. A. Whittaker, "What is software testing? And why is it so hard?," in *IEEE Software: Volume: 17: Issue: 1*, IEEE, 2000, pp. 70-79.

[6] Selenium, "Information about web elements," 30 January 2024. [Online]. Available: https://www.selenium.dev/documentation/webdriver/elements/information/. [Accessed 27 May 2024].

[7] Selenium, "Locator strategies," 20 October 2023. [Online]. Available: https://www.selenium.dev/documentation/webdriver/elements/locators/. [Accessed 27 May 2024].

[8] Selenium, "Selenium Overview," 6 February 2024. [Online]. Available: https://www.selenium.dev/documentation/overview/. [Accessed 27 May 2024].

[9] R. Yandrapally, S. Thummalapenta, S. Sinha and S. Chandra, "Robust Test Automation Using Contextual Clues," in *International Symposium on Software Testing and Analysis*, 2014.

[10] H. Kirinuki, H. Tanno and K. Natsukawa, "Color: Correct locator recommender for broken test scripts using various clues in web application," *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER),* vol. 36, no. 4, pp. 310-320, 2019.

[11] M. Leotta, A. Stocco, F. Ricca and P. Tonella, "Robula+: An Algorithm for Generating Robust Xpath Locators for Web Testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 177-204, 2016.

[12] P. Montoto, A. Pan, J. Raposo, F. Bellas and J. López, "Automated Browsing in Ajax Websites," *Data & Knowledge Engineering,* vol. 70, no. 3, pp. 269-283, 2011.

[13] M. Nass, E. Alégroth and R. Feldt, "Improving web element localization by using a large language model," 2023.

[14] Z. Xu, Q. Li and S. H. Tan, "Guiding ChatGPT to Fix Web UI Tests via Explanation-Consistency Checking," in *ACM Conference (Conference´17)*, New York, 2024.

[15] S. Balaji and M. Sundararajan Murugaiyan, "WATEERFALL Vs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC," *International Journal of Information Technology and Business Management,* vol. 2, no. 1, 2012.

[16] J. E. Bentley, W. Bank and N. Charlotte, "Software testing fundamentals - concepts, roles and terminology," in *Proceeding of SAS Conference , pp. 1-12*, 2005.

[17] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review," *Information and Software Technology,* vol. 76, pp. 92-117, 2016.

[18] O. Taipale, J. Kasurinen, J. Karhu and K. Smolander, "Trade-off between automated and manual software testing," *International Journal of Systems Assurance Engineering and Management,* vol. 2, pp. 114-125, 2011.

[19] F. Dobslaw, R. Feldt, D. Michaelsson, P. Haar, F. G. de Oliveira Neto and R. Torkar, "Estimating Return on Investment for GUI Test Automation Frameworks," in *IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019.

[20] I. Hooda and R. S. Chhillar, "Software Test Process, Testing Types and Techniques," *International Journal of Computer Applications (0975 - 8887),* vol. 111, no. 13, 2015.

[21] Z. Gao, C. Fang and A. M. Memon, "Pushing the Limits on Automation in GUI Regression Testing," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[22] E. Engström and P. Runeson, "A qualitative survey of regression testing practices," *Product-Focused Software Process Improvement/Lecture Notes in Computer Science,* vol. 6156, pp. 3-16, 2010.

[23] L. Lazic and N. Mastorakis, "Cost Effective Software Test Metrics," *WSEAS Transactions on Computers, Vol: 7, Issue: 6,* pp. 599-619, June 2008.

[24] K. Sangwatthanarat and T. Suwannasart, "Generating Test Scripts for a Single Page Web Application Based On Database Schema," in *4th International Conference on Applied Research in Engineering, Science and Technology*, Dublin, Ireland, 2021.

[25] Y. Wang, M. V. Mäntylä, Z. Liu, J. Markkula and P. Raulamo-Jurvanen, "Improving Test Automation Maturity: a Multivocal Literature Review," *Software: Testing, Verification and Reliability,* vol. 32, no. 3, 2022.

[26] D. M. Rafi, K. R. K. Moses, K. Petersen and M. V. Mäntylä, "Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey," in *7th International Workshop on Automation of Software Test (AST)*, Zurich, 2012.

[27] Y. Wang, M. V. Mäntylä, S. Demeyer, K. Wiklund, S. Eldh and T. Kairi, "Software Test Automation Maturity - A Survery of the State of the Practice," in *15th International Conference on Software Technologies ICSOFT*, 2020.

[28] K. Wiklund, S. Eldh, D. Sundmark and K. Lundqvist, "Impediments for software test automation: A systematic literature review," *Software Testing Verification and Reliability,* vol. 27, no. 8, 2017.

[29] L. Mariani, M. Pezzè and D. Zuddas, "Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles," in *ACM/IEEE 40th International Conference on Software Engineering*, 2018.

[30] S. Stresnjak and Z. Hocenski, "Usage of Robot Framework in Automation of Functional Test Regression," in *ICSEA 2011: The Sixth International Conference on Software Engineering Advances*, 2011.

[31] R. F. Foundation, "Robot Framework," Robot Framework Ry, [Online]. Available: www.robotframework.org. [Accessed 3 April 2024].

[32] R. F. Foundation, "Robot Framework User Guide Version 7.0," 2024. [Online]. Available: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html. [Accessed 3 April 2024].

[33] P. Tiwari, S. Rajendran and S. Kumaravel, "Automatic Performance Verification of Industrial Gateway using Python Framework," in *14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 2023.

[34] MarketSquare, "Github - MarketSquare/robotframework-requests," 13 April 2024. [Online]. Available: https://github.com/MarketSquare/robotframework-requests#readme. [Accessed 13 April 2024].

[35] R. F. Foundation, "SeleniumLibrary, version 6.2.0," 2024. [Online]. Available: https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html. [Accessed 13 April 2024].

[36] S. D. Fabiyi and O. Ajibuwa, "Automatic Code Commenting in Integrated Development Environments Based on Indirect Interaction with Chatbots," in *International Scientific Conference on Computer Science (COMSCI)*, 2023.

[37] Robotframework, "Github - robotframework/SeleniumLibrary - Web testing library for Robot Framework," 2024. [Online]. Available: https://github.com/robotframework/SeleniumLibrary. [Accessed 14 April 2024].

[38] A. M. Turing, "I. - COMPUTING MACHINERY AND INTELLIGENCE," *Mind,* vol. LIX, no. 236, pp. 433-460, 1950.

[39] A. P. Saygin, I. Cicekli and V. Akman, "Turing Test: 50 Years Later," *Minds and Machines,* vol. 10, pp. 463-518, 2000.

[40] P. Wang, "On Defining Artificial Intelligence," *Journal of Artificial General Intelligence,* vol. 10, no. 2, pp. 1-37, 2019.

[41] F. F.-H. Nah, R. Zheng, J. Cai, K. Siau and L. Chen, "Generative AI and ChatGPT: Applications, challenges, and AI-human collaboration," *Journal of Information Technology Case and Application Research,* vol. 25, no. 3, pp. 277-304, 2023.

[42] V. C. Müller and N. Bostrom, "Future Progress in Artificial Intelligence: A Survey of Expert Opinion," *Fundamental Issues of Artificial Intelligence,* pp. 553-571, 2016.

[43] A. Toosi, A. Bottino, B. Saboury, E. Siegel and A. Rahmim, "A Brief History of AI: How to Prevent Another Winter (A Critical Review)," *PET Clinics,* vol. 16, no. 4, pp. 449-469, 2021.

[44] Q. Mei, Y. Xie, W. Yuan and M. O. Jackson, "A Turing test of wheter AI chatbots are behaviorally similar to humans," arXiv:2312.00798 [cs.AI], 2024.

[45] S. Feuerriegel, J. Hartmann, C. Janiesch and P. Zschech, "Generative AI," *Business & Information Systems Engineering,* vol. 66, pp. 111-126, 2023.

[46] McKinsey & Company, "What is generative AI?," McKinsey, 2 April 2024. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-generative-ai. [Accessed 18 May 2024].

[47] R. Khankhoje, "Ai in Test Automation: Overcoming Challenges, Embracing Imperatives," *International Journal on Soft Computing Artificial Intelligence and Applications,* 2024.

[48] IBM, "What are large language models (LLMs)?," [Online]. Available: https://www.ibm.com/topics/large-language-models. [Accessed 18 May 2024].

[49] Y. Chang, X. Wang, Y. Wu, L. Yang, K. Zhu and J. Wang, "A Survey on Evaluation of Large Language Models," eprint arXiv, 2023.

[50] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le and C. Sutton, "Program Synthesis with Large Language Models," 2021.

[51] McKinsey & Company, "What is prompt engineering?," 22 March 2024. [Online]. Available: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-prompt-engineering. [Accessed 18 May 2024].

[52] L. Beurer-Kellner, M. Fischer and M. Vechev, "Prompting Is Programming: A Query Language for Large Language Models," *Proc. ACM Program. Lang. 7, PLDI, Article 186,* p. 24, 2023.

[53] OpenAI, "OpenAI Documentation: Prompt engineering," OpenAI, [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering. [Accessed 20 May 2024].

[54] J. Park, H. Youn and E. Lee, "An Automatic Code Generation for Self-Healing*," *Journal of Information Science and Engineering,* vol. 25, pp. 1753-1781, 2009.

[55] A. Stocco, R. Yandrapally and A. Mesbah, "Visual Web Test Repair," in *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*, Lake Buena Vista, 2018.

[56] V. Lelli, A. Blouin and B. Baudry, "Classifying and Qualifying GUI Defects," in *8th IEEE International Confence on Software Testing, Verification and Validation*, Graz, 2015.

[57] S. R. Choudhary, D. Zhao, H. Versee and A. Orso, "Water: Web application test repair," in *First International Workshop on End-to-End Test Script Engineers*, 2011.

[58] Z. Khaliq, S. U. Farooq and D. A. Khan, "Transformers for gui testing: A plausible solution to automated test case generation and flaky tests," *Computer,* vol. 55, no. 3, pp. 64-73, 2022.

[59] IBM, "What is instruction tuning?," IBM, 2024. [Online]. Available: https://www.ibm.com/topics/instruction-tuning. [Accessed 22 May 2024].

[60] Openrouter.ai, "Openrouter Supported Models," [Online]. Available: https://openrouter.ai/docs#models. [Accessed 21 May 2024].

[61] R. F. Foundation, "Style Guide Robot Framework," 2024. [Online]. Available: https://docs.robotframework.org/docs/style_guide. [Accessed 4 May 2024].

[62] RapidFuzz, "RapidFuzz 3.9.1 documentation - Indel," 2024. [Online]. Available: https://rapidfuzz.github.io/RapidFuzz/Usage/distance/Indel.html. [Accessed 20 May 2024].

[63] TinyDB, "Welcome to TinyDB!," 2024. [Online]. Available: https://tinydb.readthedocs.io/en/latest/. [Accessed 20 May 2024].

[64] "Thesis replication set," 27 May 2024. [Online]. Available: https://github.com/paapu-commits/thesis-replication-set.

[65] Itewiki, "Paljonko on ohjelmistokehityksen tuntihinta 2022?," Koodiklinikka, 2022. [Online]. Available: https://www.itewiki.fi/opas/paljonko-on-ohjelmistokehityksen-tuntihinta/#Yhteenveto-ohjelmistokehityksen-tuntihinnasta-suomessa-2022. [Accessed 25 May 2024].

[66] P. Kumar and K. Syed, ""Software testing - goals, principles and limitations," *International Journal of Computer Applications (0975 - 8887),* vol. 6, no. 9, 2010.

7 Appendix

7.1 Original Locators and Repair Examples

Table 7.1. Original locators used in test cases

Id	Locator
A1	Pro File Manager
A2	id:theme-toggle
A3	link:Starter Page
A4	name:script
A5	//*[@type="submit"]
A6	class:sr-only
S1	//*[@id="main-content"]/main/div/div[1]/div[2]
S2	//*[@id="dropdown-dashboard"]/li[1]/a

Table 7.2. Examples of locator repairs

Id	Locator
A1	File Manager
A2	id:theme-toggler
A3	link:Start Page
A4	name:scripts
A5	//*[@type="button"]
A6	class:search-only
S1	//*[@id="main-content"]/main/div[1]/div[2]
S2	//*[@id="dropdown-dashboard"]/li[2]/a

7.2 Weak Candidate Method

Table 7.3: Self-healing results with Weak Candidate Method for Test Case A1

LLM	Repair	Test case A1 locator repair suggestion
Example		File Manager
Claude 3 Sonnet	Fail	//*[normalize-space(.)='File Manager']
GPT-3.5 Turbo	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span
GPT-4 Turbo	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span
Llama 3 8B Instruct	Fail	#sidebar > div > div > div > ul > li:nth-child(7) > a > span
Llama 3 70B Instruct	Fail	id:'sidebar'
Mistral 7B Instruct	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a
Mixtral 8x22B Instruct	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span
Mistral Large	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span[contains(text(), 'File Manager')]

Table 7.4: Self-healing results with Weak Candidate Method for Test Case A2

LLM	Repair	Test case A2 locator repair suggestion
	Example	id:theme-toggler
Claude 3 Sonnet	Pass	id=theme-toggler
GPT-3.5 Turbo	Fail	id='theme-toggler'
GPT-4 Turbo	Pass	id=theme-toggler
Llama 3 8B Instruct	Pass	id:theme-toggler
Llama 3 70B Instruct	Fail	id='theme-toggler'
Mistral 7B Instruct	Fail	id='theme-toggle">
Mixtral 8x22B Instruct	Pass	id:theme-toggler
Mistral Large	Fail	id='theme-toggler'

Table 7.5: Self-healing results with Weak Candidate Method for for Test Case A3

LLM	Repair	Test case A3 locator repair suggestion
	Example	link:Start Page
Claude 3 Sonnet	Fail	link:href='#a'
GPT-3.5 Turbo	Fail	link:Starter Page
GPT-4 Turbo	Fail	link:href='#a'
Llama 3 8B Instruct	Fail	//a[@href='#starter']
Llama 3 70B Instruct	Fail	"link:Starter"
Mistral 7B Instruct	Fail	link:#a
Mixtral 8x22B Instruct	Fail	xpath://a[text()='Starter Page']
Mistral Large	Fail	ink:href='#a'

Table 7.6: Self-healing results with Weak Candidate Method for Test Case A4

LLM	Repair	Test case A4 locator repair suggestion
	Example	name:scripts
Claude 3 Sonnet	Fail	name='scripts'
GPT-3.5 Turbo	Fail	name='description'
GPT-4 Turbo	Fail	name='scripts'
Llama 3 8B Instruct	Fail	xpath://script
Llama 3 70B Instruct	Fail	name="scripts"
Mistral 7B Instruct	Fail	{"candidates": [{"locator_value": "name='scripts'"},{"locator_value": "name='description'"},{ "locator_value": "script"},{"locator_value": "<script>" },{"locator_value": "<script>"},{"locator_value": "<script>" …
Mixtral 8x22B Instruct	Fail	name='scripts'
Mistral Large	Fail	<script

Table 7.7: Self-healing results with Weak Candidate Method for Test Case A5

LLM	Repair	Test case A5 locator repair suggestion
	Example	//*[@type="button"]
Claude 3 Sonnet	Fail	//button[@type='submit']
GPT-3.5 Turbo	Fail	type='button'
GPT-4 Turbo	Pass	//*[@type='button']
Llama 3 8B Instruct	Fail	//input[@type='submit']
Llama 3 70B Instruct	Fail	/*[type='button']
Mistral 7B Instruct	Fail	{/'[@type='button']'/}
Mixtral 8x22B Instruct	Pass	//*[@type='button']
Mistral Large	Pass	//*[@type='button']

Table 7.8: Self-healing results with Weak Candidate Method for Test Case A6

LLM	Repair	Test case A6 locator repair suggestion
	Example	class:search-only
Claude 3 Sonnet	Fail	class='sr-only'>Search
GPT-3.5 Turbo	Fail	class='sr-only'>Search</label>
GPT-4 Turbo	Fail	class='sr-only'>Search</label>
Llama 3 8B Instruct	Fail	//span[@class='sr-only'][contains(.,'Search')]
Llama 3 70B Instruct	Fail	xpath://*[@class='sr-only' and text()='Search']
Mistral 7B Instruct	Fail	class='sr-only'>Search
Mixtral 8x22B Instruct	Fail	//span[contains(text(),'Search')]
Mistral Large	Fail	class='sr-only'>Search

Table 7.9: Self-healing results with Weak Candidate Method for for Test Case S1

LLM	Repair	Test case A7 locator repair suggestion
	Example	//*[@id="main-content"]/main/div[1]/div[2]
Claude 3 Sonnet	Fail	//*[@id='main-content']//h3
GPT-3.5 Turbo	Fail	//*[@id='main-content']/main/div/div[1]/div[2]/h3
GPT-4 Turbo	Pass	id=main-content
Llama 3 8B Instruct	Fail	data-date='1717794000000'
Llama 3 70B Instruct	Pass	id:main-content
Mistral 7B Instruct	Fail	.//div[@id='main-content']/main/div/div[1]/div[2]/h3
Mixtral 8x22B Instruct	Fail	//*[@id='main-content']/main/div/div[1]/div[2]/h3
Mistral Large	Fail	//h3[contains(text(), 'Statistics Table')]

Table 7.10: Self-healing results with Weak Candidate Method for Test Case S2

LLM	Repair	Test case A8 locator repair suggestion
	Example	//*[@id="dropdown-dashboard"]/li[2]/a
Claude 3 Sonnet	Pass	//*[@id='dropdown-dashboard']//a[normalize-space(.)='DataTables']
GPT-3.5 Turbo	Fail	//*[@id='dropdown-dashboard']/li[1]/a
GPT-4 Turbo	Fail	//*[@id='apps-dropdown']/li[1]/a
Llama 3 8B Instruct	Fail	#dropdown-dashboard > li:nth-child(1) > a
Llama 3 70B Instruct	Fail	id='dropdown-dashboard'
Mistral 7B Instruct	Fail	//*[@id="main-content"]/main/div/div[1]/div[2]/h3
Mixtral 8x22B Instruct	Fail	//*[@id='dropdown-dashboard']/li[1]/a[text()='DataTables']
Mistral Large	Fail	//*[@aria-controls='dropdown-dashboard']/li[1]/a

7.3 Strong Candidate Method

Table 7.11: Self-healing results with Strong Candidate Method for Test Case A1

LLM	Repair	Test case A1 locator repair suggestion
Example		File Manager
Claude 3 Sonnet	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span
GPT-3.5 Turbo	Fail	//*[@id='sidebar']/div[1]/div[1]/div[1]/ul[1]/li[7]/a[1]/span[1]
GPT-4 Turbo	Fail	//*[@id='sidebar']/div[1]/div[1]/div[1]/ul[1]/li[7]/a[1]/span[1]
Llama 3 8B Instruct	Fail	.//span[@class="ml-3"]
Llama 3 70B Instruct	Fail	//span[@class='ml-3' and contains(text(), 'File Manager')]
Mistral 7B Instruct	Fail	{"xpath": "/html[1]/body[1]/div[1]/aside[1]/div[1]/div[1]/div[1]/ul[1]/li[5] /a[1]/span[1]"}
Mixtral 8x22B Instruct	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span
Mistral Large	Fail	//*[@id='sidebar']/div/div[1]/div/ul/li[7]/a/span[contains(text(), 'File Manager')]

Table 7.12: Self-healing results with Strong Candidate Method for Test Case A2

LLM	Repair	Test case A2 locator repair suggestion
	Example	id:theme-toggler
Claude 3 Sonnet	Pass	id:theme-toggler
GPT-3.5 Turbo	Pass	id:theme-toggler
GPT-4 Turbo	Pass	//*[@id='theme-toggler']
Llama 3 8B Instruct	Pass	//button[@id='theme-toggler']
Llama 3 70B Instruct	Pass	id:theme-toggler
Mistral 7B Instruct	Fail	{”xpath":"/html[1]/body[1]/nav[1]/div[1]/div[1]/div[2]/button[4]"}
Mixtral 8x22B Instruct	Pass	id:theme-toggler
Mistral Large	Pass	id:theme-toggler

Table 7.13: Self-healing results with Strong Candidate Method for for Test Case A3

LLM	Repair	Test case A3 locator repair suggestion
	Example	link:Start Page
Claude 3 Sonnet	Fail	/html[1]/body[1]/div[1]/aside[1]/div[1]/div[1]/div[1]/ ul[1]/li[3]/a[1]
GPT-3.5 Turbo	Fail	link:http://localhost:5095/start/
GPT-4 Turbo	Pass	xpath://*[@id='sidebar']/div[1]/div[1]/div[1]/ul[1]/li[3]/a[1]
Llama 3 8B Instruct	Fail	//li[@class='flex items-center p-2 text-base text-gray-900 transition duration-75 rounded-lg hover:bg-gray-100 group dark:text-gray-200 dark:hover:bg-gray-700']
Llama 3 70B Instruct	Fail	xpath:///html[1]/body[1]/div[1]/aside[1]/div[1]/div[1]/div[1] /ul[1]/li[3]/a[1]
Mistral 7B Instruct	Fail	””
Mixtral 8x22B Instruct	Pass	xpath=//a[contains(@href, '/start/')]
Mistral Large	Pass	xpath:/html[1]/body[1]/div[1]/aside[1]/div[1]/div[1]/div[1]/ ul[1]/li[3]/a[1]

Table 7.14: Self-healing results with Strong Candidate Method for Test Case A4

LLM	Repair	Test case A4 locator repair suggestion
	Example	name:scripts
Claude 3 Sonnet	Fail	select[name='scripts']
GPT-3.5 Turbo	Pass	name=scripts
GPT-4 Turbo	Pass	xpath=//*[@id='main-content'] /main[1]/div[1]/div[1]/div[2]/div[1]/div[1]/div[2]/div[1]/table[1] /tbody[1]/tr[1]/td[4]/span[1]/select[1]
Llama 3 8B Instruct	Pass	//select[@name='scripts']
Llama 3 70B Instruct	Pass	name:scripts
Mistral 7B Instruct	Fail	```vbnet{"tag": "SELECT","class": "w-full py-1 px-2 mb-1 border border-gray-200 rounded dark:border-gray-600 dark:bg-gray-700 dark:text-white","name": "scripts","id": "","value": "check-db-health.py"}```
Mixtral 8x22B Instruct	Fail	id: scripts
Mistral Large	Pass	name:scripts

Table 7.15: Self-healing results with Strong Candidate Method for Test Case A5

LLM	Repair	Test case A5 locator repair suggestion
	Example	//*[@type="button"]
Claude 3 Sonnet	Fail	/html[1]/body[1]/main[1]/div[1]/div[1]/form[1]/button[1]
GPT-3.5 Turbo	Fail	//*[@id='id_username']
GPT-4 Turbo	Pass	xpath=/html[1]/body[1]/main[1]/div[1]/div[1]/form[1]/button[1]
Llama 3 8B Instruct	Fail	//input[@type='submit']
Llama 3 70B Instruct	Pass	xpath://*[@type='button' and @class='w-full px-5 py-3 text-base font-medium text-center text-white bg-primary-700 rounded-lg hover:bg-primary-800 focus:ring-4 focus:ring-primary-300 sm:w-auto dark:bg-primary-600 dark:hover:bg-primary-700 dark:focus:ring-primary-800']
Mistral 7B Instruct	Fail	//button[@type='button']
Mixtral 8x22B Instruct	Fail	//button[@class='w-full px-5 py-3 text-base font-medium text-center text-white bg-primary-700 rounded-lg hover:bg-primary-800 focus:ring-4 focus:ring-primary-300 sm:w-auto dark:bg-primary-600 dark:hover:bg-primary-700 dark:focus:ring-primary-800']
Mistral Large	Pass	xpath=//button[@type='button']

Table 7.16: Self-healing results with Strong Candidate Method for Test Case A6

LLM	Repair	Test case A6 locator repair suggestion
	Example	class:search-only
Claude 3 Sonnet	Pass	//form//label[contains(@class,'search')]
GPT-3.5 Turbo	Pass	xpath:/html[1]/body[1]/nav[1]/div[1]/div[1]/div[1]/form[1]/label[1]
GPT-4 Turbo	Pass	xpath://html[1]/body[1]/nav[1]/div[1]/div[1]/div[1]/form[1]/label[1]
Llama 3 8B Instruct	Pass	//label[@class='search-only']
Llama 3 70B Instruct	Pass	xpath://*[@class='search-only']
Mistral 7B Instruct	Fail	tag:LABEL
Mixtral 8x22B Instruct	Pass	xpath:/html[1]/body[1]/nav[1]/div[1]/div[1]/div[1]/form[1]/label[1]
Mistral Large	Pass	xpath:/html[1]/body[1]/nav[1]/div[1]/div[1]/div[1]/form[1]/label[1]

Table 7.17: Self-healing results with Strong Candidate Method for for Test Case S1

LLM	Repair	Test case A7 locator repair suggestion
	Example	//*[@id="main-content"]/main/div[1]/div[2]
Claude 3 Sonnet	Fail	/html[1]/body[1]/div[1]/div[2]/main[1]/div[1]/div[2]/h3[1]
GPT-3.5 Turbo	Pass	//*[@id='main-content']/main[1]/div[1]/div[2]/h3[1]
GPT-4 Turbo	Pass	//*[@id='main-content']/main[1]/div[1]/div[2]/h3[1]
Llama 3 8B Instruct	Pass	//div[div/h3[@class='flex items-center mb-4 text-lg font-semibold text-gray-900 dark:text-white']]
Llama 3 70B Instruct	Pass	//*[@id='main-content']/main[1]/div[1]/div[2]/h3[1]
Mistral 7B Instruct	Fail	{"broken_locator": "//*[@id='main content']/main/div/div[1]/div[2]/h3","locator_candidates": [{"tag": "H3", "index": 87.86},{"tag": "H3", "inde …
Mixtral 8x22B Instruct	Fail	/html[1]/body[1]/div[1]/div[2]/main[1]/div[1]/div[2]/h3[1]
Mistral Large	Pass	//*[@id='main-content']/main[1]/div[1]/div[2]/h3[1]

Table 7.18: Self-healing results with Strong Candidate Method for Test Case S2

LLM	Repair	Test case A8 locator repair suggestion
	Example	//*[@id="dropdown-dashboard"]/li[2]/a
Claude 3 Sonnet	Pass	//*[@id='dropdown-dashboard']/li[1]/a
GPT-3.5 Turbo	Fail	//*[@id='dropdown-dashboard']/li[1]/a
GPT-4 Turbo	Pass	//*[@id='dropdown-dashboard']/li[2]/a[1]
Llama 3 8B Instruct	Fail	//a[contains(@href, 'api/product')]
Llama 3 70B Instruct	Pass	//*[contains(@href, '/tables/')]
Mistral 7B Instruct	Fail	//*[@id='dropdown-dashboard']/li[1]/a
Mixtral 8x22B Instruct	Fail	//*[@id='dropdown-dashboard']/li[1]/a
Mistral Large	Pass	//*[@id='dropdown-dashboard']/li[2]/a[1]