Article

AutoQALLMs: Automating Web Application Testing Using Large Language Models (LLMs) and Selenium

Sindhupriya Mallipeddi [1], Muhammad Yaqoob [1], Javed Ali Khan [1,*], Tahir Mehmood [2], Alexios Mylonas [1] and Nikolaos Pitropakis [3]

[1] Cybersecurity and Computing Systems Research Group, Department of Computer Science, University of Hertfordshire, Hatfield AL10 9AB, UK; sm23ahb@herts.ac.uk (S.M.); m.yaqoob3@herts.ac.uk (M.Y.)
[2] School of Information Technology, UNITAR International University, Petaling Jaya 47301, Selangor, Malaysia
[3] Department of Information Technology, Cybersecurity and Computer Science, The American College of Greece, 15342 Athens, Greece; npitropakis@acg.edu
[*] Correspondence: j.a.khan@herts.ac.uk

Abstract

Modern web applications change frequently in response to user and market needs, making their testing challenging. Manual testing and automation methods often struggle to keep up with these changes. We propose an automated testing framework, AutoQALLMs, that utilises various LLMs (Large Language Models), including GPT-4, Claude, and Grok, alongside Selenium WebDriver, BeautifulSoup, and regular expressions. This framework enables one-click testing, where users provide a URL as input and receive test results as output, thus eliminating the need for human intervention. It extracts HTML (Hypertext Markup Language) elements from the webpage and utilises the LLMs API to generate Selenium-based test scripts. Regular expressions enhance the clarity and maintainability of these scripts. The scripts are executed automatically, and the results, such as pass/fail status and error details, are displayed to the tester. This streamlined input–output process forms the core foundation of the AutoQALLMs framework. We evaluated the framework on 30 websites. The results show that the system drastically reduces the time needed to create test cases, achieves broad test coverage (96%) with Claude 4.5 LLM, which is competitive with manual scripts (98%), and allows for rapid regeneration of tests in response to changes in webpage structure. Software testing expert feedback confirmed that the proposed AutoQALLMs method for automated web application testing enables faster regression testing, reduces manual effort, and maintains reliable test execution. However, some limitations remain in handling complex page changes and validation. Although Claude 4.5 achieved slightly higher test coverage in the comparative evaluation of the proposed experiment, GPT-4 was selected as the default model for AutoQALLMs due to its cost-efficiency, reproducibility, and stable script generation across diverse websites. Future improvements may focus on increasing accuracy, adding self-healing techniques, and expanding to more complex testing scenarios.

Keywords: testing; web application; LLM; GPT; selenium

check for updates

Academic Editor: Wenbing Zhao

Received: 12 October 2025
Revised: 10 November 2025
Accepted: 14 November 2025
Published: 18 November 2025

Citation: Mallipeddi, S.; Yaqoob, M.; Khan, J.A.; Mehmood, T.; Mylonas, A.; Pitropakis, N. AutoQALLMs: Automating Web Application Testing Using Large Language Models (LLMs) and Selenium. *Computers* **2025**, *14*, 501. https://doi.org/10.3390/computers14110501

1. Introduction

Software testing plays a pivotal role in evaluating the quality and correctness of software systems throughout the development lifecycle [1]. It ensures that applications function as intended and meet user expectations. Software testing can consume over 50%

of total development costs, especially in complex modern applications [1]. Web applications are particularly challenging to test due to their dynamic content, asynchronous interactions, and frequent interface changes, a set of problems well established in the field's literature [2–4]. As these systems evolve rapidly to meet changing user demands, testing methodologies must also adapt to maintain reliability and efficiency [3,5]. Inadequate software testing can lead to critical system failures and widespread vulnerabilities, emphasising the need for reliable, automated solutions [6,7]. These cases highlight the need for more effective testing methods that can be applied in real-world systems with limited time and resources.

Modern web applications are characterised by complex structures, dynamic content, and asynchronous interactions [8]. These features present significant testing challenges that have been reported in the literature [2,9]. This complexity makes it challenging to test effectively using traditional manual methods [8]. When relying on manual testing, developers and QA engineers must spend additional time and effort on test design and execution, thereby increasing the organisation's overall testing costs [10]. This makes traditional manual testing slow and difficult. Automated testing [11], codeless testing [12], and model-based testing using machine learning have been proposed to improve this process [3]. However, these methods have certain limitations, including difficulties in training models on large datasets and concerns regarding the quality of the resulting product.

LLMs have been widely used in the software development lifecycle [6], including code generation [13], requirements engineering [14], vulnerability detection [15], and test case creation [10,16,17]. Recently, the use of LLMs in software testing has become a significant and rapidly expanding field of research, with the literature mapping the current landscape of tools and techniques [18]. Tools such as GPTDroid and PENTESTGPT demonstrate that LLMs can perform automated testing with minimal human assistance [19,20]. This demonstrates that LLMs can be integrated into automated and context-sensitive testing systems for improved quality with reduced resources and lower costs. However, to date, according to our knowledge, there is little work that uses LLMs in complete testing pipelines for web applications. A few studies have utilised LLMs in conjunction with web scraping or regular expressions; however, these approaches do not encompass full automation or script refinement [16]. There exists a research gap concerning automated systems capable of dynamically adapting to changes in page structure while preserving script readability.

This study introduces a test automation framework called AutoQALLMs that utilises GPT-4 alongside Selenium WebDriver, BeautifulSoup, and regular expressions. The proposed approach reads the HTML of any webpage, finds the elements, and uses GPT-4, Claude 4.5 and Grok to generate Selenium scripts. Regular expressions are used to refine and adapt the scripts even when the webpage structure changes. Unlike conventional methods that rely on static scripts and require updates for every UI change, this approach dynamically adapts to the structure of webpages using DOM (Document Object Model) parsing and pattern-based matching, reducing manual effort and improving resilience to UI changes [3,4]. Unlike GPTDroid, which is designed for mobile testing [20], AutoQALLMs are tailored for web application testing. While GPTDroid focuses on extracting UI trees and generating scripts for native mobile apps, AutoQALLMs target dynamic webpages by parsing the DOM using BeautifulSoup and refining scripts with regular expressions. Additionally, AutoQALLMs support zero-shot prompt-based test generation (enabling HTML-to-Selenium translation without prior examples or fine-tuning) using GPT-4, Claud 4.5, and Grok, eliminating the need for intermediate instrumentation or mobile-specific tools. To our knowledge, this is the first study to integrate LLMs with DOM parsing and regex-based script enhancement, creating a fully automated, browser-oriented testing framework.

With the proposed approach, we aim to answer three research questions: (i) How can LLMs be combined with web scraping and Selenium to create test scripts? (ii) Can LLMs turn HTML into working Selenium scripts using zero-shot prompts? (iii) Can AutoQALLMs outperform manual or semi-automated testing in code coverage and fault detection? Our results demonstrate that LLMs can be adapted to convert HTML into valid Selenium scripts without requiring human-written code. The proposed AutoQALLMs are tested across various websites, and expert feedback indicates that they improve coverage, fault detection, and the time required to generate scripts compared to traditional methods.

Our key contribution is the development of a framework that extracts HTML from a webpage and parses the DOM using BeautifulSoup. We compose structured prompts to direct LLMs (GPT-4, Claude 4.5, and Grok) in generating Selenium-based test scripts. Additionally, we have implemented a regex-driven refinement module that enhances the robustness of these scripts by identifying element patterns, thereby improving their adaptability to changes in the user interface (UI). Another contribution is the integration of these components into a dynamic test automation flow that supports zero-shot test case generation to eliminate the need for manually written templates or prior training. We also conducted extensive evaluations across 30 websites to measure AutoQALLMs' performance in terms of coverage, execution time, and fault detection. Furthermore, we gathered and incorporated expert feedback from software testers to validate the system's practicality and to understand its strengths and limitations in real-world testing environments . AutoQALLMs' code is made available for future work (https://github.com/Sindhupriya2797/AutoQALLMs).

We developed AutoQALLMs, a fully automated framework that transforms web content into executable Selenium test scripts using LLM intelligence. We designed the framework to perform zero-shot test generation by extracting HTML from any given webpage, parsing its DOM structure with BeautifulSoup, and constructing structured prompts that guide LLM to produce context-aware Selenium scripts. We further implemented a regex-driven refinement module that identifies and optimises element patterns to improve the syntactic validity and adaptability of scripts when the user interface or DOM changes. Through this design, we integrated parsing, generation, and refinement into a single workflow that supports end-to-end, self-adjusting automation from HTML extraction to test execution and reporting.

We also devised an evaluation framework to validate the scalability and practicality of AutoQALLMs. We tested the framework across 30 diverse web applications and measured performance using quantitative metrics such as coverage, generation time, execution speed, and fault detection. We further collected expert feedback from professional testers to assess qualitative attributes, including readability, adaptability, maintainability, and scalability. Our results show that AutoQALLMs generate accurate Selenium scripts within seconds, achieve coverage levels close to manual testing, and significantly reduce human effort. With these contributions, we present a practical and intelligent approach to AI-assisted web testing that advances the role of LLMs in automated quality assurance.

The remainder of this paper is organised as follows: Section 2 presents related work in software testing, including manual, ML-based, and LLM-based approaches. Section 3 describes the architecture and methodology of the AutoQALLMs framework. Section 4 outlines the experimental setup, evaluation metrics, and testing procedures. Section 5 presents the results, expert feedback, and comparative analysis. Finally, Section 6 concludes the study and discusses limitations as well as potential directions for future research.

2. Related Work

2.1. Manual Testing

Manual testing involves testers acting as end users, manually executing test cases, and using all available program features to identify problems and ensure the software meets standards [6,7]. This process is often the first step for any new application before automation is considered, as it helps determine whether automated testing is worthwhile [7]. Testers manually follow test plans, execute test cases, and capture evidence, such as screenshots, comparing actual results with expected results to detect discrepancies and unexpected behaviour [21,22].

Although manual testing is effective in uncovering detailed issues, such as invalid input values or unexpected user interface behaviours, it is time-consuming, tedious, and can be error-prone, especially for complex applications with many test cases [8,22]. The extensive effort and cost required for manual testing, particularly in large-scale regression scenarios or systems, has led to a growing demand for automated testing solutions [10,23]. However, automated testing does not fully replace manual methods, as human involvement remains necessary for evaluation of certain testing tasks, especially those that involve nuanced or dynamic user interactions [23]. These limits have led to the search for more rapid and automated methods that require less human work.

2.2. ML and DL Based Testing

Manual testing is time-consuming and repetitive, and earlier automated systems could not learn or adapt to new elements [4]. Some approaches utilised support vector machine (SVM) models with Selenium and BeautifulSoup to predict test cases based on HTML element patterns; however, they were limited to static content and required manual setup [4]. A mapping study showed that most ML-based testing research focused on test case generation and bug prediction, with fewer studies on GUI and security testing and limited industrial validation [1]. Other works applied ML for test suite refinement, fault localisation, risk-based testing, and test oracle automation, though many required domain-specific data and expert input [5].

In developing their framework, ref. [4] considered a Multinomial Naive Bayes classifier followed by SVM as the best model for assigning test cases. Clustering techniques have been used to group test cases with similar behaviours, helping refine test suites by identifying redundant tests [1]. For example, rule-based ML systems, such as RUBAR, were applied for fault localisation. At the same time, separate risk-based models used fault prediction data to prioritise testing and reduce execution costs [5]. These studies demonstrate how classical ML models improve accuracy, reduce manual effort, and support intelligent decision-making in the software testing lifecycle.

Although deep learning models have improved visual testing and automated UI test case generation, they often require large labelled datasets and struggle with visually complex or changing interfaces [24]. Other approaches utilised deep architectures, such as LSTM, CNN, and deep belief networks, for test generation and bug prediction. However, these methods faced significant hurdles. A primary issue was the need for massive, high-quality datasets for training, which were often unavailable [25]. These difficulties limited the practical application of many early deep learning techniques in real-world CI/CD pipelines [25]. Earlier deep neural network testing methods, like DeepXplore, focused on neuron coverage and successfully uncovered corner-case bugs. Still, they were limited to gradient-based transformations and lacked flexibility for broader automation scenarios [26]. Due to these limitations, ML and DL models were unable to generate complete Selenium scripts or adapt to new tasks using natural language. The primary advantage of LLMs over earlier models is their ability to perform complex generation tasks, such as generating code from natural-language prompts, without requiring extensive, task-specific training datasets. To address this gap, we began exploring the use of LLMs by providing prompts to generate Selenium code, thereby enabling more flexible and end-to-end test automation.

2.3. LLM-Based Testing

LLMs for software testing are a broad and active research area, with recent literature providing a comprehensive landscape of the various tasks, models, and prompting techniques currently being explored [18]. Into this broader landscape, web application testing has been specifically identified as a promising future direction [9]. Following this trend, LLMs have recently been applied to different areas of software testing to reduce manual effort and support adaptive test generation. GPT-4 was integrated with Selenium WebDriver in a feedback loop to parse the DOM, generate interaction steps, and adjust test flows based on updated page structure, allowing fully dynamic GUI testing without pre-recorded data [27]. Another approach compared manual scripting, GitHub Copilot, and two ChatGPT variants for generating web E2E test scripts using Gherkin inputs. The study found that ChatGPT Max (multi-turn) with GPT-4 version consistently reduced development time and required fewer manual corrections [16].

For JavaScript unit test generation, the paper tool TESTPILOT was evaluated across 25 npm packages. While the LLM-generated tests produced natural-looking code with meaningful assertions, they also achieved significantly higher code coverage than the state-of-the-art traditional tool [17]. System-level test case generation was explored through US2Test, a Python-based tool that combined GPT-4 with user stories from Redmine. The tool employed black-box techniques, including equivalence class and boundary value analysis, and demonstrated a 60% reduction in design effort in a government setting [28].

LLMs were also used for visual testing, where a framework generated assertions and bug reports by analysing UI screenshots and historical bug data. This enabled A/B comparisons and visual regression testing, though it lacked logical DOM interactions [22]. Beyond test generation, LLMs are also being applied to the critical problem of test maintenance and robustness. A key challenge is "locator fragility", where tests break because UI elements change between software versions. The VON Similo LLM framework addresses this by using an LLM to semantically compare and select the most likely correct element from a list of candidates, significantly reducing localisation failures [29].

Another study proposed a modular framework, T5-GPT, which combined Crawljax, a T5 classifier, a data faker, and GPT-4o to automate web form interaction and validation. The model improved form coverage but had difficulty handling complex inputs and long execution times [8]. A multi-agent LLM framework was introduced for generating test cases, executing them, visualising call graphs, and creating PDF reports using tools such as Gemini and Audio WebClient. Although it achieved high test coverage, it was limited to backend logic and unit-level testing [23]. Mobile GUI testing was reframed as a question-answering task using GPTDroid, which applied memory-aware prompts and GUI context extraction to simulate user actions. The model improved coverage and bug detection but struggled with apps requiring gestures or backend validation [20]. In cybersecurity testing, PENTESTGPT utilised LLMs for interactive penetration testing, featuring separate modules for reasoning, command generation, and output parsing. It achieved strong results in OWASP benchmarks and CTF tasks, although it required human-in-the-loop support and prompt tuning [19].

These studies show that LLMs have expanded the scope of automated testing by supporting visual testing, system-level planning, and code generation. However, many tools are limited to specific domains, lack complete test automation pipelines, or still rely on manual oversight. To better handle the challenges of dynamic navigation and complex interactions, other research has focused on creating intermediate representations of web applications to guide LLMs. For instance, ref. [30] introduced a system that builds screen transition graphs to model site navigation and state graphs for conditional forms, using these structures to generate more robust test cases. The choice of model and prompt structure is also crucial, as demonstrated by large-scale empirical studies. Li et al. [31] conducted a study comparing 11 LLMs for web form testing, concluding that GPT-4 significantly outperformed other models and that providing a cleanly parsed HTML structure in the prompt was more effective than using raw HTML. A more recent paradigm leverages Large Vision-Language Models (LVLMs), which analyse not just the HTML code but also visual screenshots of the webpage. For example, ref. [32] proposed VETL, a technique that uses an LVLM's scene understanding capabilities to generate relevant text inputs and identify the correct GUI elements to interact with, even when HTML attributes are unclear. In contrast, we proposed an AutoQALLMs approach that utilises LLMs in a structured manner, experimenting with various LLMs, including GPT-4, Claude, and Grok, with DOM parsing, prompting, and Selenium to generate reliable and reusable test scripts for modern, dynamic web applications. Table 1 compares the existing tools and methods with the approach proposed in this study.

Table 1. Methodological Comparison of Test Automation Approaches.

Reference	Approach	Technique	Key Findings	Methodological Comparison
[4]	ML-driven web UI testing	Beautiful Soup + ML + Selenium	automatically generate and classify test cases using ML models for each web element.	Combines web scraping, ML classification, and test execution; lacks adaptability to live DOM changes.
[3]	Codeless web testing using ML	Selenium + SVM	Codeless architecture to predict and automate functional testing using training data.	Uses SVM for test prediction; limited real-time responsiveness and DOM parsing.
[33]	AI tool survey and taxonomy	Multivocal Literature Review	Reviewed 55 LLM-powered test tools, highlighting gaps in full-pipeline automation.	No implementation; literature synthesis of tools highlights need for integrated AI pipelines.
[34]	Chatbot testing automation	ML/NLP-based adaptation	Adaptive chatbot testing through AI, but with limited scalability and need for refinement.	Uses AI for conversational flow adaptation; lacks full web execution support.
[17]	Unit test generation using LLM	GPT-3.5 (TESTPILOT)	Generated unit tests with high coverage without fine-tuning.	Generates tests via few-shot prompting; limited to unit testing with no web interaction.
[27]	GUI web testing using GPT-4	GPT-4 + Selenium	Simulated GUI testing using LLMs, partial automation, but required validation.	Integrates LLM with Selenium for GUI navigation; lacks self-healing and optimisation.
[16]	E2E Selenium test using LLMs	ChatGPT + Copilot + Selenium	Accelerated E2E script writing, but required manual refinement.	Maps user stories to scripts via NLP; manual refinement is needed post-generation.
[20]	Android GUI testing using LLM	GPT-4 + memory prompting	Outperformed baselines in activity coverage, limited to Android.	Uses LLM and memory prompt cycles; excels in mobile apps but is not generalised for the web.
[32]	Visual UI A/B testing with LLM	LLM + visual diff validation	Enhanced visual validation, but lacked HTML script generation.	Visual validation using screenshots; lacks DOM parsing and test flow generation.
[11]	Framework-based web test automation	Selenium + DBUnit + Razor templates	Template-driven automation framework with limited adaptability to dynamic web content.	Template-based automation requires manual definition and lacks LLM-driven adaptability.

Table 1. *Cont.*

Reference	Approach	Technique	Key Findings	Methodological Comparison
[28]	LLM-based system test generation	GPT-4 + US2Test + Black-box techniques	Real-world public sector testing tool achieved 100% suite coverage and saved 60% manual effort.	Manual test design from user stories using GPT-4; unlike AutoQALLMs, no DOM parsing or script execution. Focuses on generating test cases for manual validation.
[8]	Automated form testing using LLMs	T5 classifier + GPT-4o + Crawljax + Mocker	Proposed T5-GPT model improving coverage and form interaction over RL-based systems.	Uses T5 for field classification and GPT-4 for validation; unlike AutoQALLMs, it doesn't generate Selenium scripts, focusing on form-filling automation.

3. Methodology

The proposed AutoQALLMs automate the web application testing in four stages: (i) HTML Extraction, (ii) LLMs-based test script generation, (iii) Regex-based script cleaning and optimisation, and (iv) test execution and reporting. These stages form a modular and scalable framework that automatically generates Selenium scripts from web content. The architectural choice to use a powerful LLM like GPT-4, Claude, and Grok in conjunction with structured parsing of HTML is supported by recent empirical studies, which have found this combination to be highly effective for test generation compared to other LLMs and less structured inputs [31]. This approach improves the speed and accuracy of web application testing. Each stage operates as an independent unit with defined inputs and outputs, enabling iterative development and practical deployment. Below, the proposed methodology is elaborated with various functions developed for GPT-4. Moreover, the complete implementation of various LLMs, including GPT-4, Claude, and Grok, is available on GitHub (https://github.com/Sindhupriya2797/AutoQALLMs).

3.1. Step 1: HTML Extraction

The proposed AutoQALLMs approach extracts HTML when the user (software tester) enters the URL of the web application to be tested, as shown in Figure 1. This URL provides the raw HTML content needed for LLM-based testing. The system calls the fetch_html(url) function, which uses the requests library to send an HTTP GET request. The command response = requests.get(url) fetches the webpage content and response.raise_for_status() handles HTTP errors such as 404 or 500 to ensure reliability during data collection.

After retrieving the HTML, the content is parsed using BeautifulSoup with BeautifulSoup(response.text, 'html.parser'). This converts the unstructured HTML into a well-organised DOM tree. The DOM enables structured access to webpage elements, such as buttons, links, and form fields. This representation is essential for the next stage, where the LLMs generate test cases based on a contextual understanding of the UI. After parsing, the AutoQALLMs extract content using the parse_html (soup) function. This function isolates a predefined set of HTML tags relevant for web UI testing. It specifically targets:

(a) The `<title>` tag provides the title of the webpage to verify that the correct page has been loaded and is displayed as expected in the browser.
(b) Anchor (`<a>`) tags with valid `href` attributes. These contain hyperlinks for navigation.
(c) Header tags from `<h1>` to `<h6>`, which capture the semantic layout and hierarchy of the content. These are useful for validating content structure and screen reader accessibility.
(d) Image (`<img>`) tags that include `src` attributes, representing embedded media that should be loaded and visible.

(e) Form (`<form>`) for verifying form submission structure (attributes such as action, method, and name)
(f) Input (`<input>`) for capturing interactive input fields with attributes (type, id, name, placeholder).
(g) Button (`<button>`) for clickable elements that trigger actions or submissions.
(h) Select (`<select>`) for validating dropdown options and user selections.

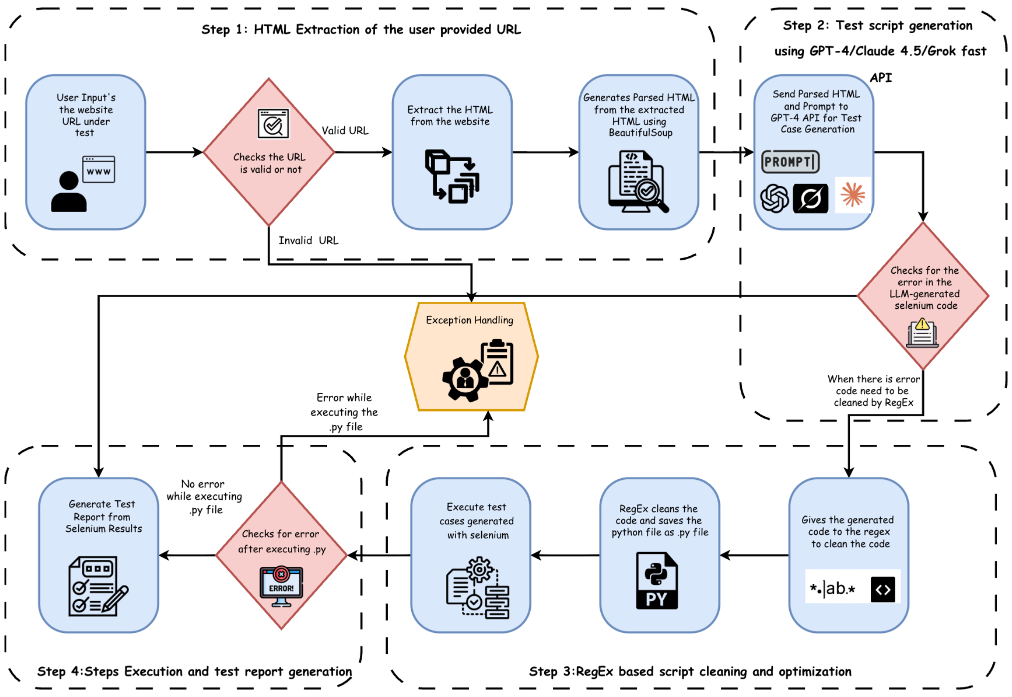
Step 1: HTML Extraction of the user provided URL
User Input's the website URL under test
www
Checks the URL is valid or not
Valid URL
Invalid URL
Extract the HTML from the website
Generates Parsed HTML from the extracted HTML using BeautifulSoup
Step 2: Test script generation using GPT-4/Claude 4.5/Grok fast API
Send Parsed HTML and Prompt to GPT-4 API for Test Case Generation
PROMPT
Checks for the error in the LLM-generated selenium code
When there is error code need to be cleaned by RegEx
Exception Handling
Error while executing the .py file
No error while executing .py file
Generate Test Report from Selenium Results
Checks for error after executing .py
ERROR!
Execute test cases generated with selenium
RegEx cleans the code and saves the python file as .py file
PY
Gives the generated code to the regex to clean the code
.|ab.
<>
Step 4:Steps Execution and test report generation
Step 3:RegEx based script cleaning and optimization

Figure 1. AutoQALLMs Web Testing framework: LLM-Generated Test Case Extraction, Processing, and Execution with BeautifulSoup, Regex, and Selenium.

This subset was selected as it represents both the fundamental and interactive elements required to verify a webpage's structure and behaviour. It includes tags for verifying a page's identity (<title>), navigational integrity (<a>), content hierarchy (<h1>–<h6>), and media rendering (<img>), as well as interactive components such as forms (<form>), input fields (<input>), buttons (<button>), and dropdowns (<select>). Together, these elements capture the core static and dynamic aspects of modern web interfaces, enabling comprehensive validation of user interactions such as data entry, submission, and option selection.

These extracted elements are assembled into a Python dictionary using key-value mappings, where each category is stored as a list of values. A representative sample output would resemble the following:

```
{
  "title": "Example Web Page",
  "links":"[
    "https://example.com/about",
    "https://example.com/contact"
  ],
  "headings":"{
    "h1":"["Welcome"]"
    "h2":"["About Us"]
  },
  "images":"[
    "/images/banner.png"
" ],
  "forms":"[
    {
      "action":"/submit-form",
      "method": "post",
      "id": "contactForm",
      "name": "contact_form"
    }
  ],
  "inputs": [
    {
      "type": "text",
      "name": "username",
      "id": "userField",
      "placeholder": "Enter your name"
    },
    {
      "type": "email",
      "name": "email",
      "id": "emailField",
      "placeholder": "Enter your email"
    }
  ],
  "buttons": [
    {
      "text": "Submit",
      "type": "submit",
      "id": "submitBtn",
      "name": "submit_button"
    }
  ],
  "selects": [
    {
      "name": "country",
      "id": "countrySelect",
      "options": ["India", "United Kingdom", "United States"]
    }
  ]
}
```

The parsed output is lightweight and semantically rich, capturing key aspects of the page's user interface while remaining easy to interpret. It serves as the direct input for the next stage, where LLM prompts are constructed and test scripts are generated using the GPT-4, Claude, and Grok APIs (Application Programming Interface). To ensure the reliable execution of the HTML extraction phase, exception handling is implemented. If the network request fails, the URL is invalid, or the response cannot be parsed, the system catches the error, logs a clear message, and stops further processing. This prevents incomplete or faulty HTML data from affecting later stages, helping to maintain the accuracy and consistency of the overall test automation workflow. Step 1 involves acquiring the webpage content via HTTP requests and then using BeautifulSoup to parse its structure, identifying key elements. This step is fundamental for setting up the automated generation of test cases.

3.2. Step 2: Test Script Generation

After the HTML is parsed and key elements are extracted, the framework enters its second phase, which serves as the intelligence layer, where we utilise various LLMs (GPT-4, Claude, and Grok) to generate automation scripts based on a predefined prompt, requiring minimal manual effort. The structured data from the first phase is passed to the generate_selenium_code() function. This function merges the parsed HTML content with a prompt to generate executable Selenium code. The prompt has two primary purposes: (1) to describe the web content clearly to LLMs and (2) to guide LLMs in producing test logic that follows standard automation practices. The prompt used in this framework is depicted in Box follows:

Prompt to generate Selenium code using extracted HTML

```
prompt = (
    f"-You are a **strict code generator**. Your output must
    contain ONLY executable Python code,"
    f"-with no explanations, comments, or~markdown fences.\n\n"
    f"-Generate Selenium Python test code for the following
    parsed HTML data.\n\n"
    f"-URL: {url}\n\n"
    f"-Parsed Data:\n{json.dumps(parsed_data, indent=2)
    [:2000]}...\n\n"
    f"-Instructions:\n"
    f"-Add test for javascript based webelements also"
    f"-Automate each test case using Selenium 4+ syntax.\n"
    f"-**Use only ’find_element(By.<LOCATOR>, value)’ and
    ’find_elements(By.<LOCATOR>, value)’** - never use
    deprecated ’find_element_by_*’ or ’find_elements_by_*’
    methods.\n"
    f"-Import ’By’ from ’selenium.webdriver.common.by’ at the
    top of the code.\n"
    f"- Open the page using ChromeDriver (not headless) and
    maximise the window.\n"
    f"- Add time.sleep() where ever needed"
    f"- Create 30 sequential test cases that interact with the
    elements (titles, headings, images, links, forms, inputs,
    buttons, and selects).\n"
    f"- Each test should include realistic user actions
    (typing, clicking, submitting, selecting options) with
    time.sleep() between actions.\n"
    f"- Log each test as ’Test X Passed/Failed’ directly in
    the console.\n"
    f"- Include ’driver.quit()’ at the end of the script.\n"
    f"- Do NOT include markdown ("‘) or any descriptive text
    before or after the code.\n"
    f"- The entire output must be syntactically valid Python -
    ready to run as-is.\n"

    )
```

In response, LLMs generate a structured Python script using Selenium WebDriver for browser automation, the find_element() method for DOM interaction, and try-except blocks for error handling. The script begins with WebDriver setup, runs through the test steps, and ends with driver.quit() for cleanup (refer to the video in the GitHub link). This format aligns with standard practices in UI testing. The LLMs output often includes extras, such as markdown formatting (e.g., "'python), phrases like "Here is the Selenium code," and explanatory comments. While helpful for human readers, these elements are unnecessary for automated execution. To address this, a post-processing step is included to clean and optimise the script before running it, Step 3 of the proposed AutoQALLMs approach, as shown in Figure 1. By embedding best practices and test logic within an intelligent agent, this step reduces manual effort and ensures consistency, scalability, and quicker adaptation to UI changes. It demonstrates how LLM-driven development can be integrated into real world test automation.

3.3. Step 3: Script Cleaning and Optimisation

After LLMs generates the Selenium test script, the framework moves to its third stage, which prepares the raw output for execution by cleaning the script using automated text processing tools, as shown in Figure 1. The goal is to convert the semi-structured and often verbose output into a clean Python file that follows syntax rules and software engineering best practices. Although LLMs produce logical and well-structured code, they frequently include additional text intended for human readers. Examples include markdown markers like "'python, phrases such as "Here is the Selenium code," and placeholders like "Please replace this section…". These elements are not suitable for automated execution and must be removed. To handle this, the framework uses the clean_selenium_code() function, which applies a set of regular expressions to remove unnecessary content. This is a typical application, as regular expressions are a powerful and widely used technique for text parsing and validation. Although they can be difficult for developers to read and maintain, their utility in programmatic text processing is well-established [35]. Table 2 lists the types of content removed during this step and explains why each is excluded:

Table 2. Patterns removed from LLMs output and their justifications.

Pattern Removed	**Reason**
Markdown code fences (`python`, ‘‘‘)	Artefacts of LLM output formatting, not required for script execution.
LLM explanatory notes (e.g., "Here is the code…")	Non-executable and verbose explanations not relevant to automated testing.
Prompt feedback (e.g., "Please replace…")	Instructional residues meant for human readers, not part of executable code.
Redundant block comments	add clutter and reduced code readability and maintainability.

The regex rules are carefully designed to ensure that only executable code remains in the final script. By removing narrative content and presentation-related syntax, the framework provides a cleaned script that is ready for automated testing without causing execution errors by the Python interpreter. After this cleaning step, the script is passed through a formatting stage using the autopep8 library. This tool enforces Python's PEP 8 style guidelines, which focus on readability, consistent indentation, and proper line spacing. Formatting the code at this point enhances clarity for human reviewers and makes the script more easily integrable into automated workflows or version control systems.

In addition to formatting and content cleanup, the framework uses the remove_lines_after_quit() function. This step ensures the script ends immediately after the driver.quit() command, which marks the close of the Selenium session. Any lines that follow, such as debug notes or leftover placeholders, are removed to prevent errors during test execution. The cleaned code is then saved as a .py file. These cleaning and optimisation steps together form a strong post-processing layer that enforces proper syntax and functional accuracy. This final version of the script moves to the last stage of the framework: test execution and reporting. At this point, the refined and clean script interacts with a live browser session, completing the LLM-powered automation cycle.

3.4. Step 4: Test Execution and Reporting

The final stage of the proposed AutoQALLMs framework, i.e., Step 4, focuses on test execution and reporting. It runs the optimised .py test script on the web application and records the outcomes to verify the application's functionality. Execution is managed by the execute_selenium_code() function, which launches a browser using Selenium's Chrome WebDriver. The browser is configured with ChromeOptions to open in full-screen mode. Running in non-headless mode enables real-time observation of the test flow, helping to identify issues such as UI misalignment, loading delays, or unexpected pop-ups. The script then carries out a sequence of actions generated by LLMs, including navigating the DOM, locating elements, clicking buttons, submitting forms, and following links.

During execution, all test outcomes, whether passed or failed, are printed to the console along with contextual log messages. These logs serve as a live reporting mechanism, offering immediate insight into the test flow. Although the current setup uses terminal-based reporting, the architecture is flexible. Future versions can easily integrate tools like PyTest plugins or Allure to generate HTML dashboards, JSON logs for CI pipelines, or XML reports for systems like TestRail and Zephyr. Moreover, if an error occurs, whether due to a logic issue, an outdated selector, or an unhandled exception, detailed error traces are captured. This final phase validates both the functionality of the web application and the reliability of the LLM-generated test script. It completes the framework with actionable results, demonstrating that LLM-driven test automation is practical and effective for real-world QA processes.

4. Experimental Setup

4.1. Tools and Technologies

AutoQALLMs combine tools, libraries, and APIs that enable intelligent, scalable, and high-performance test automation. Table 3 presents a summary of the main tools and their respective functions.

Table 3. Technological components and their roles in AutoQALLMs.

Component	**Technology**	**Purpose**
Language	Python	Primary programming language; script orchestration and integration
HTML Parsing	BeautifulSoup	Structured parsing of webpage DOM to extract testable elements
HTTP Communication	`requests`	Performs GET requests to fetch HTML content from the user-provided URL
LLM Test Generation	OpenAI GPT-4 API	Generates context-aware Selenium test scripts based on HTML structure
Regex Filtering	Python `re` module	Cleans and refines LLM-generated code by removing non-functional content

Table 3. *Cont.*

Component	Technology	Purpose
Code Formatting	`autopep8`	Applies PEP 8 styling for readable and maintainable Python scripts
Web Automation Engine	Selenium WebDriver	Executes UI-level automation across web applications in a real browser
Browser Driver	ChromeDriver	Acts as the bridge between Selenium and the Chrome browser instance
Execution Environment	`os` module (`os.system`)	Triggers execution of saved Python test files in the host terminal
IDE	PyCharm	Provides an efficient debugging and development environment

4.2. Testing Strategy

To validate the reliability, performance, and robustness of AutoQALLMs, a multi-layered testing strategy is implemented, comprising unit testing, integration testing, functional testing, and performance evaluation.

Unit testing is applied to core functions such as fetch_html(), parse_html(), and clean_ selenium_code(). Each function is tested independently to confirm it works correctly under different input conditions. Simulating edge cases, such as invalid URLs, missing tags, or malformed HTML content, also verifies error handling.

Integration testing verifies that modules work together smoothly. For example, the flow of parsed data from parse_html() to generate_selenium_code() is tested using websites with different layouts. The handoff from LLMs output to the cleaned script is also reviewed to ensure it maintains logic and functionality.

Functional testing is performed on real websites to confirm that the generated scripts can interact with and validate dynamic elements. This includes checking page titles, clicking links, detecting broken images, and filling out input forms. These tests ensure the automation mimics realistic user behaviour.

Performance testing measures how efficiently the framework runs. It assesses response time and execution speed across different types of websites, ranging from simple static pages to complex, JavaScript-heavy ones. Key metrics such as script generation time, test execution duration, and browser response delays are recorded and analysed.

Together, these validation steps confirm that the framework runs reliably across various environments. They also demonstrate their adaptability, effectiveness in practical use, and readiness for integration into continuous testing pipelines.

4.3. Test Subjects

To validate the reliability, performance, and robustness of AutoQALLMs, the framework was evaluated on 30 publicly available automation practice websites. The complete list of target applications is provided in Table 4. These 30 websites encompass a diverse range of structural and functional characteristics, ensuring a comprehensive evaluation of the proposed framework. The selection includes static and text-oriented pages (e.g., HerokuApp: iFrame, Nested Frames), interactive and form-based applications (e.g., DemoQA: Practice Form, LetCode), and multimedia-rich or e-commerce platforms (e.g., DemoBlaze, OpenCart, Greencart). Across these websites, the average number of interactive menus or navigation components ranged from three to eight per page, depending on the website's complexity and content type. Static websites primarily comprised textual and image elements, whereas dynamic and multimedia sites incorporated AJAX-driven components, modal windows, dynamic tables, and multiple form-input fields. This diversity ensures that AutoQALLMs have been assessed under heterogeneous testing conditions, validating their adaptability and robustness across simple, content-focused pages as well as complex, multi-component web interfaces.

Table 4. List of Web Applications Used for Evaluation.

Website Name	**URL**
UI Testing Playground	http://uitestingplayground.com/, (accessed on 1 November 2025)
TestPages	https://testpages.herokuapp.com/styled/index.html, (accessed on 1 November 2025)
Global SQA Demo Site	https://www.globalsqa.com/demo-site/, (accessed on 1 November 2025)
Automation Bro Practice	https://practicetestautomation.com/, (accessed on 28 August 2025)
DemoBlaze E-commerce	https://www.demoblaze.com/, (accessed on 2 November 2025)
Rahul Shetty Academy	https://rahulshettyacademy.com/AutomationPractice/, (accessed on 25 July 2025)
OpenCart Demo Store	https://demo.opencart.com/, (accessed on 2 November 2025)
PHPTRAVELS Demo	https://phptravels.com/demo/, (accessed on 1 November 2025)
Rahul Shetty Academy	https://rahulshettyacademy.com/angularpractice/, (accessed on 2 July 2025)
Greencart	https://rahulshettyacademy.com/seleniumPractise/#/, (accessed on 25 July 2025)
HerokuApp: JS Alerts	https://the-internet.herokuapp.com/javascript_alerts, (accessed on 1 November 2025)
HerokuApp: Nested Frames	https://the-internet.herokuapp.com/nested_frames, (accessed on 1 November 2025)
HerokuApp: iFrame	https://the-internet.herokuapp.com/iframe, (accessed on 2 November 2025)
HerokuApp: Dynamic Loading	https://the-internet.herokuapp.com/dynamic_loading, (accessed on 1 November 2025)
LetCode	https://letcode.in/test, (accessed on 25 July 2025)
HerokuApp: Hovers	https://the-internet.herokuapp.com/hovers, (accessed on 1 November 2025)
HerokuApp: Key Presses	https://the-internet.herokuapp.com/key_presses, (accessed on 2 November 2025)
DemoQA: Practice Form	https://demoqa.com/automation-practice-form, (accessed on 22 July 2025)
DemoQA: Web Tables	https://demoqa.com/webtables, (accessed on 25 July 2025)
DemoQA: Buttons	https://demoqa.com/buttons, (accessed on 28 August 2025)
DemoQA: Widgets (Date Picker)	https://demoqa.com/date-picker, (accessed on 28 August 2025)
DemoQA: Interactions (Droppable)	https://demoqa.com/droppable, (accessed on 28 August 2025)
UI Playground: AJAX Data	http://uitestingplayground.com/ajax, (accessed on 2 November 2025)
UI Playground: Client Side Delay	http://uitestingplayground.com/clientdelay, (accessed on 1 November 2025)
UI Playground: Dynamic Table	http://uitestingplayground.com/dynamictable, (accessed on 1 November 2025)
CosmoCode Web Table	https://cosmocode.io/automation-practice-webtable/, (accessed on 23 August 2025)
Guru99 Banking Demo	https://demo.guru99.com/V4/, (accessed on 1 November 2025)
Sauce Demo	https://www.saucedemo.com/, (accessed on 3 November 2025)
ParaBank	https://parabank.parasoft.com/, (accessed on 2 November 2025)
WebDriverUniversity.com	http://webdriveruniversity.com/, (accessed on 2 November 2025)

4.4. Evaluation Metrics

To evaluate the performance of the proposed AutoQALLMs, the following metrics are used:

Script Generation Time: The time taken to generate test scripts before execution. Faster script generation reduces manual effort and improves productivity [33].
Execution Speed: The total time needed to run all test cases after the script is ready. Shorter execution time increases overall testing efficiency [21].
Test Coverage: Test coverage means how many elements on a webpage (like links, headings, and images) are tested by the test script compared to the total number of elements parsed. More coverage means better testing [34].

Currently, AutoQALLMs do not scan or test all the elements on the page. Still, it covers several common ones, including links, anchors, forms, inputs, buttons, selects, headings, and images, and creates 30 basic test cases.

Test coverage is calculated using the following formula:

$$\text{Test Coverage (\%)} = \left(\frac{\text{Elements tested}}{\text{Total elements found}} \right) \times 100$$

For example, on the test page https://rahulshettyacademy.com/AutomationPractice/ (accessed on 25 July 2025), the tool found 33 elements (17 links, 13 headings, and 3 image). Out of those, 30 were tested. Therefore, the estimated coverage is:

$$\text{Test Coverage (\%)} = \left(\frac{30}{33} \right) \times 100 \approx 90.91\%$$

Failure Rate: The failure rate is the percentage of test cases that fail during execution. This can happen due to incorrect locators, timing issues, or logic errors in the script [4].

For the proposed approach, the failure rate is around 20%. This is expected, as zero-shot prompting can occasionally generate incorrect locators for complex or dynamically loaded elements. Future work will aim to reduce this by experimenting with few-shot learning, fine-tuning and chain-of-thought learning approaches. Moreover, the observed 20% failure rate suggests that while AutoQALLMs are comparatively effective for rapid, initial test suite generation, their output would still require a brief human review before being integrated into a mission-critical regression pipeline. This positions the tool as a powerful 'test accelerator' rather than a complete replacement for QA oversight.

Adaptability: The degree to which a product or system can be effectively and efficiently adapted for different or evolving hardware, software or other operational or usage environments [36].
Readability: The ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It is a human judgement of how easy a text is to understand [37].
Maintainability: The degree of effectiveness and efficiency with which the intended maintenance engineers can modify a product or system. Maintainability is composed of modularity, reusability, analysability, modifiability, and testability [36].
Scalability: The degree to which a system, network, or process can handle a growing amount of work, or its potential to be enlarged to accommodate that growth [38].

The independent variables considered in this evaluation include script generation time, execution speed, number of web elements parsed, and type of web application. These variables were varied across multiple websites to assess AutoQALLMs' performance. The dependent variables, namely test coverage, failure rate, readability, adaptability, maintainability, and scalability, were used to measure performance outcomes. The relationship between these variables indicates that as page complexity and element count increased, AutoQALLMs maintained high coverage and low execution time, confirming their scalability and efficiency.

5. Results and Discussion

We evaluated the performance of the AutoQALLMs using 8 widely used metrics: (i) script generation time, (ii) execution speed, (iii) test coverage, (iv) failure rate, (v) adaptability, (vi) readability, (vii) maintainability, and (viii) scalability. We compared the results with manually written Selenium scripts and Monkey Testing. Each method was tested on 30 web applications. The tests were repeated 50 times, and the results were averaged to minimise variation. The evaluation focused on how each method performs in structured testing tasks and how suitable each is for real-world use.

Monkey Testing is a widely used technique for automated testing that involves generating random user interactions on applications without predefined steps [8,27,39]. This method can help identify unexpected issues, such as application crashes, and is often used for stress testing [39]. However, because the approach "lacks knowledge about the form fields, its random trials can be extremely slow," which makes it less useful for structured or planned automation tasks [8].

The performance of AutoQALLMs-generated scripts, manually written scripts, and monkey testing is evaluated, and the results are presented in Table 5. The comparison highlights how each method performs across different web applications.

Table 5. Comparative Analysis of Test Automation Approaches.

Metric	AutoQALLMs	Manual Selenium	Monkey Testing
Script Generation Time	5 s	2 h	N/A
Execution Speed	Fast (20 s/test)	Moderate (35 s/test)	Variable
Test Coverage (%)	91%	98%	50–60%
Failure Rate (%)	20%	10%	50%
Adaptability to UI Changes	Moderate	High	None
Readability	Moderate	High	N/A
Maintainability	High	Moderate	Low
Scalability	Very High	Low	N/A

5.1. Evaluation Methodology and Expert Rubric

To validate the results, structured feedback was collected from five domain experts in software testing and automation using a short survey, as shown in Appendix A. The experts were selected from diverse industries and roles to provide balanced insights. Their input helped interpret the observed differences in test performance across AutoQALLMs-generated scripts, manually written scripts, and Monkey Testing (see Table 6).

Expert 1 stated that AutoQALLMs could create test scripts within seconds, whereas manual scripting typically took several hours to complete. The expert believed this speed was beneficial for manual testers who lack strong coding skills. He also mentioned that AutoQALLMs can generate multiple tests simultaneously, making it easier to scale testing for large projects. However, the expert pointed out that some of the generated scripts were too general. While AutoQALLMs made it easy to regenerate broken tests after UI changes, the expert felt that manual scripts were still easier to understand, fix, and share with others. Expert 2 noted that AutoQALLMs covered slightly fewer UI elements (96%) than manual scripts (98%). However, the expert explained that manual scripts often included more meaningful checks based on the application's logic. In the expert's view, AutoQALLM was good at identifying what was visible on the screen, but it sometimes missed deeper, more thoughtful validations. The expert 2 felt this showed a trade-off between more exhaustive coverage and detailed accuracy.

Similarly, Expert 3 focused on how well the scripts handled changes to the user interface. The expert stated that manual scripts were more effective at handling these changes because human testers could write the code in a way that anticipated updates. In contrast, AutoQALLMs scripts often broke when the layout changed. The expert also said Monkey Testing didn't adjust at all, making it unreliable for serious testing. Additionally, Expert 4 agreed with many of the earlier points, particularly regarding the speed at which the scripts ran. The expert stated that AutoQALLMs scripts typically complete in about 20 s, which helps speed up regression testing. Manual scripts took longer (approximately 35 s), primarily due to additional checks added by the tester. Expert 4 also agreed with Expert 3 that manual scripts handled changes in the UI better. Finally, Expert 5 talked about failure rates. The expert added that manual scripts had the lowest failure rate (approximately 10%) because testers could carefully select the correct elements and use custom wait times. AutoQALLMs had a moderate failure rate (approximately 20%); it performed well on stable websites but struggled with dynamic ones. Monkey Testing failed most often (around 50%) because its actions were random. The expert mentioned that AutoQALLMs scripts were easy to read but sometimes longer and less clear than manual scripts.

Table 6. Expert opinions on AutoQALLMs, manual, and Monkey Testing.

Expert	**Background**	**Opinion Highlights**
Expert 1: Lead Automation Engineer (23 yrs, Rapidue Technology Ltd.)	Leads large-scale QA teams and enterprise test automation.	AutoQALLMs created test scripts within seconds, which helped teams working in fast-paced agile projects. It also made it easier to scale testing across multiple scenarios. However, some scripts felt too general. While it was easy to fix broken tests by regenerating them, manual scripts were still easier to understand and share.
Expert 2: Senior QA Analyst (5 yrs, Domino Printing Solutions)	Focuses on industrial device UI testing.	AutoQALLMs covered slightly less of the user interface (96% vs. 98%) and often missed deeper checks that manual testers would include. Manual scripts had more meaningful validations. This showed a trade-off between broad coverage and detailed testing.
Expert 3: QA Lead (6 yrs, Recykal)	Works on environmental platform testing with dynamic UIs.	Manual scripts worked better when the user interface changed, as testers could plan for updates. AutoQALLMs scripts often failed with layout changes. Monkey Testing didn't adjust at all and wasn't reliable.
Expert 4: Test Automation Specialist (7 yrs, Recykal)	Maintains Selenium in CI/CD pipelines.	AutoQALLMs scripts ran faster (around 20 s) than manual scripts (about 35 s), which helped speed up testing. But manual scripts were more flexible when the UI changed. Monkey Testing didn't follow a clear process and was inconsistent.
Expert 5: UI/UX Automation Engineer (5 yrs, Recykal)	Tests usability and design consistency.	Manual scripts had the lowest failure rate (around 10%) because testers carefully picked elements and set waits. AutoQALLMs had a higher failure rate (about 20%); it worked fine on stable pages but struggled with dynamic ones. Monkey Testing failed often (about 50%) due to its random actions. AutoQALLMs scripts were easy to read but sometimes too long or unclear.

The expert feedback confirmed that the AutoQALLMs testing brought gains in speed, coverage, and scalability. However, it still had some limitations in terms of test accuracy, UI changes, and deeper validation. These limitations underscore the continued importance of human oversight in automation testing workflows. The results indicate that LLMs can support software testing, but they should be used in conjunction with manual review for reliable outcomes.

To complement the qualitative opinions presented in Table 6, the same five domain experts were asked to rate the qualitative attributes of the generated test scripts: readability, adaptability, maintainability, and scalability using a five-point Likert rubric (1 = Poor, 5 = Excellent). The detailed rubric and evaluation guidelines provided to experts are presented in Appendix A.2. The aggregated expert ratings are summarised in Table 7.

Table 7. Expert Evaluation Scores for Qualitative Metrics (1–5 Scale).

Metric	Expert 1	Expert 2	Expert 3	Expert 4	Expert 5	Mean Score
Readability	5	4	5	4	5	4.6
Adaptability	4	4	5	5	4	4.4
Maintainability	4	5	4	4	5	4.4
Scalability	5	4	5	5	4	4.6

The quantitative evaluation confirmed high consistency among experts, supporting the reliability of the qualitative feedback and validating the robustness of the generated Selenium scripts across all four attributes.

5.2. Model Comparision

In addition to these performance metrics, the analysis was expanded to include cost efficiency. Table 8 presents both the cost per million tokens and the estimated cost per individual test script. Across the evaluated websites, GPT-4 averaged about USD 0.011 [40] per test, compared with USD 0.016 for Claude 4.5 [41] and USD 0.0006 for Grok Fast [42], based on an average token consumption of 1800 tokens per site. This cost-aware evaluation quantitatively supports GPT-4's balance between accuracy and affordability, reinforcing the study's emphasis on optimising trade-offs between performance and operational expense rather than focusing solely on raw coverage. LLM comparison resources and implementation details are available for future work (https://github.com/Sindhupriya2797/AutoQALLMs).

Table 8. Comparative Analysis of Model Performance, Coverage, and Cost Efficiency.

Model	Test Coverage (%)	Avg. Gen. Time (s)	Cost/1M Tokens (USD)	Avg. Tokens/Test	Cost/Test (USD)
GPT-4	91	5.4	Input: 2.50 Output: 10.00	1800	0.011
Claude 4.5 Sonnet	96	4.8	Input: 3.00 Output: 15.00	1800	0.016
Grok Fast	88	6.2	Input: 0.20 Output: 0.50	1800	0.0006

Notes. Cost per test case was calculated using the average token usage of approximately 1800 tokens per website (900 input + 900 output). Pricing reflects 2025 API rates. GPT-4 achieves a strong balance between accuracy and cost ($0.01 per test), Claude 4.5 offers slightly higher coverage at 50% higher cost, and Grok Fast provides the lowest cost but reduced coverage.

During experimentation, each test case generation consumed approximately 1800 tokens per website, evenly distributed between input and output tokens. The estimated cost per script for each model was computed using 2025 API pricing as follows:

$$\text{Cost per script} = \left(\frac{\text{Input Tokens}}{10^6} \times \text{Input Rate}\right) + \left(\frac{\text{Output Tokens}}{10^6} \times \text{Output Rate}\right)$$

For 900 input and 900 output tokens:

$$\begin{aligned}
\text{GPT-4:} &\quad (900 \times \frac{2.50}{10^6}) + (900 \times \frac{10.00}{10^6}) = \$0.011 \\
\text{Claude Sonnet 4.5:} &\quad (900 \times \frac{3.00}{10^6}) + (900 \times \frac{15.00}{10^6}) = \$0.016 \\
\text{Grok Fast:} &\quad (900 \times \frac{0.20}{10^6}) + (900 \times \frac{0.50}{10^6}) = \$0.00063
\end{aligned}$$

Thus, the average cost per test case was approximately $0.01 for GPT-4, $0.016 for Claude 4.5, and $0.0006 for Grok Fast, demonstrating that GPT-4 offers a strong trade-off between cost and accuracy.

The pricing structure of each model clarifies the observed cost-performance trade-offs. Grok Fast is designed for high-volume, low-cost tasks, with output rates nearly twenty times cheaper than GPT-4 and thirty times cheaper than Claude 4.5. Claude 4.5 Sonnet is the most expensive, particularly in terms of output tokens, reflecting its emphasis on advanced reasoning. GPT-4 offers a balanced middle ground between capability and affordability. Across all three, output tokens remain significantly more costly than input tokens.

To extend the existing LLM comparisons, as shown in Table 8, Claude 4.5 achieved the highest coverage and execution reliability, benefiting from its stronger contextual reasoning and coherent instruction-following ability. GPT-4, however, demonstrated the best balance between syntactic accuracy, runtime stability, and execution speed, making it the most consistent model for Selenium-based test automation. Although Grok Fast generated outputs more quickly, its limited handling of dynamic DOM structures led to a lower pass rate.

5.3. Computational Complexity Analysis

In addition to the empirical comparison of execution time, the computational complexity of the proposed framework was analysed to evaluate its scalability. AutoQALLMs operate through four sequential phases: (i) HTML extraction and parsing, (ii) LLM-based script generation, (iii) regex-driven code cleaning, and (iv) Selenium-based execution. The overall time complexity can be expressed as:

$$T(n, m, t) = O(n + m + t) \tag{1}$$

where n denotes the number of parsed HTML elements, m represents the length of the generated script, and t corresponds to the number of executed test cases. Since HTML parsing dominates the total computation, the framework exhibits an approximately linear growth rate:

$$T(n) \approx O(n) \tag{2}$$

This indicates that the execution time increases proportionally with the number of web elements.

Manual Selenium scripting, on the other hand, requires repetitive human intervention for every additional element and test case, resulting in a slower, non-scalable process that can be approximated as:

$$T(n, h) = O(n \times h) \tag{3}$$

where h is the human effort factor per element.

Similarly, Monkey Testing performs random interactions with a complexity of $O(r)$, where r is the number of random events required to reach sufficient coverage. However, due to its stochastic nature, this approach demands significantly more iterations to achieve comparable accuracy.

Empirically, AutoQALLMs generated executable Selenium scripts within approximately 20–25 s per website, compared to 2 h for manual scripting, while maintaining 96% coverage. The average execution time per test was around 20 s, confirming both the linear time complexity and the substantial reduction in computational and analytical effort.

5.4. Comparison with State-of-the-Art Approaches

A significant contribution to codeless test automation was made by [3], who proposed a framework combining Selenium and machine learning to generate tests without writing code. Their method relied on an SVM trained on manually annotated data to predict actions. Still, it lacked testing several behaviours at the same time (Tested only for search functionality). Similarly, ref. [8] introduced a T5-GPT-based framework that combined Crawljax, T5, and GPT-4o to automate web form interaction. Their method parsed the DOM using Crawljax to identify input fields and used LLMs for field classification and value generation. While it improved coverage over reinforcement learning agents, it was limited to form-filling tasks and could not handle categories whose formats depend on the selected locations. In mobile GUI testing, Liu et al. [20] presented GPTDroid, which redefines testing as a question-and-answer task using GPT-3.5 with functionality-aware prompts. This method achieved high activity coverage on Android apps but was limited to mobile domains, and it faced challenges with UI components that lacked clear text labels. Lastly, ref. [22] proposed an LLM-powered approach for visual UI and A/B testing, using screenshot comparisons to identify rendering issues. Although effective for visual validation, the method lacked DOM-level analysis and did not produce an actionable test script.

Another key point of comparison is the underlying methodology used to interpret the web application's structure. While AutoQALLMs generate scripts by providing a direct, parsed HTML summary to the LLM, the framework by [30] employs an intermediate modelling step. Their approach creates explicit screen transition graphs and state graphs to model the application's flow before generating tests. This graph-based methodology is specifically designed to improve test coverage for dynamic navigation and complex conditional forms, which are known challenges for more direct generation methods. In contrast, AutoQALLMs eliminate the need for handwritten templates or screenshot-based prompts by utilising various LLMs, including GPT-4, Claude, and Grok, to generate executable Selenium scripts directly from HTML. It integrates BeautifulSoup to parse live DOM structures and uses regular expressions to adapt the script automatically to structural changes. Unlike previous approaches, AutoQALLMs cover functional testing, dynamically updates scripts, and supports a full-cycle pipeline from generation to execution and reporting without relying on visual UI cues or fixed form data. This enables AutoQALLMs to maintain robustness in fast-changing web applications. Based on our understanding and knowledge, we demonstrated the comparison between the proposed and state-of-the-art approaches in Table 9.

In addition to the model comparisons discussed earlier, three complementary categories were examined to contextualise existing approaches: bot-detection and behaviour analysis frameworks, web crawling systems such as Scrapy, and Named Entity Recognition (NER) techniques. Behaviour analysis tools, such as OWASP AppSensor, monitor application layer activities to detect anomalous or automated behaviour and trigger alerts based on predefined sensor rules [43]. Scrapy is an open-source crawling framework designed for efficient extraction of static HTML content, although it cannot process dynamic or interactive web elements [44]. NER techniques [45] identify and classify meaningful entities such as names, locations, and organisations within unstructured text using advanced natural language processing models. Table 10 summarises these approaches, highlighting their objectives, methodologies, and core limitations in relation to intelligent automation systems that integrate semantic reasoning with executable validation.

Table 9. Comparison of LLM-Based Test Automation Approaches.

Approach	Domain	Key Techniques Used	Limitations	Strengths
[3]	Web Application Testing	Selenium + SVM (ML-based prediction)	Tested only search functionality	Codeless prediction of test actions
[8]	Web Form Testing	Crawljax + T5 + GPT-4o	Form-specific; challenges with geo-sensitive field formats	Improved form coverage via LLMs
[20]	Mobile GUI Testing	GPT-3.5 + GUI tree + memory prompts	Restricted to Android; challenges when elements lack text labels	High activity coverage for mobile apps
[32]	Visual UI Testing	LLM + Screenshot Comparison	No DOM parsing; does not generate test scripts	Effective visual regression and A/B testing
AutoQALLMs (Ours)	Web Application Testing	GPT-4 + BeautifulSoup + Regex + Selenium	Still developing robustness for highly dynamic UIs	Full pipeline automation, dynamic script generation

Table 10. Comparative Overview of Bot-Detection, Scrapy, NER, and AutoQALLMs Approaches based on our understanding and knowledge.

Tool/Technique	Objective	Methodology	Limitations	Ref.
Bot-Detection/Behaviour Analysis (e.g., OWASP AppSensor)	Detect and respond to automated or malicious activities within web applications.	Monitors application-layer behaviour using predefined sensors and triggers alerts for suspicious events.	Focuses on security anomaly detection; does not support web interaction or functional validation.	[43]
Scrapy	Extract structured content from static HTML pages.	Utilises Python-based crawling, rule-based XPath/CSS selectors, and scheduling pipelines for large-scale data extraction.	The Scrapy-based crawler exhibited a significant limitation when analysing pure single-page applications (SPAs).	[44]
Named Entity Recognition (NER)	Identify and classify semantic entities within unstructured text.	Applies NLP models for tokenisation, sequence labelling, and contextual embeddings (e.g., BERT, Word2Vec) to extract entity types.	Its classifications are probabilistic, carrying inherent uncertainty about their accuracy and thus requiring confidence scores rather than absolute answers.	[45]
AutoQALLMs (Proposed)	Automate intelligent web testing through end-to-end script generation and execution.	Integrate LLMs with BeautifulSoup, Regex, and Selenium to parse DOM elements, generate test scripts, and validate functionality.	Currently optimised for modern browsers; handling of complex dynamic UI elements remains a future enhancement.	This study

5.5. Discussion

The proposed AutoQALLMs provided a preliminary study on how LLMs can collaborate with tools such as DOM parsers and Selenium to automatically generate and execute test scripts for websites. The results show that when provided with well-structured prompts and the correct HTML input, LLMs can generate Selenium scripts that run successfully without any human intervention. This directly answers our first two research questions: (i) "How can LLMs be combined with web scraping and Selenium to create test scripts?", and (ii) "Can LLMs turn HTML into working Selenium scripts using zero- shot prompts?" Using BeautifulSoup to extract HTML elements and designing prompts that give the script to generate selenium test scripts, AutoQALLMs were able to convert web content into test actions. The use of zero-shot prompting also meant that the system could handle new or unfamiliar page elements without requiring retraining of the model.

For the third research question, which asked "Can AutoQALLMs outperform manual or semi-automated testing in code coverage and fault detection?", the expert feedback revealed both strengths and areas for improvement. On the positive side, AutoQALLMs achieved broad UI coverage (96%) that was competitive with manual methods (98%) and saved significant time in writing scripts. However, there were also concerns, such as missing deeper logic validations and having lower debuggability compared to manually written scripts. AutoQALLMs were able to quickly regenerate broken tests and run them, which is helpful for agile teams and CI/CD pipelines. This capability directly addresses a significant, unresolved challenge in the field, as identified by recent surveys: the high maintenance costs of test suites in the face of frequent UI changes [9]. Overall, the approach shows promise in making automation more accessible, especially for teams with less coding experience, and it opens up new directions for combining human expertise with LLM-based automation in the future.

While AutoQALLMs are primarily implemented using GPT-4, a comparative evaluation was conducted with Claude 4.5 and Grok Fast to benchmark performance. The findings indicated that Claude 4.5 achieved marginally higher test coverage (96%) due to its advanced architecture and enhanced reasoning capabilities. Nonetheless, GPT-4 consistently generated syntactically valid Selenium scripts with fewer runtime errors and exhibited faster, more predictable generation times across diverse websites. When both cost and stability were considered, GPT-4 emerged as the most balanced option for scalable deployment: its average cost per test generation was lower than that of Claude 4.5, and its prompt outputs were more reproducible under identical conditions, an essential factor for continuous testing pipelines. Consequently, the framework continues to utilise GPT-4 as its default engine, while Claude 4.5 and Grok Fast serve as comparative baselines that contextualise performance-versus-cost trade-offs in LLM-driven testing. It should also be noted that Claude 4.5 represents a newer model generation than GPT-4, which partly explains its marginally higher coverage. This distinction highlights the need for ongoing benchmarking across future releases, such as GPT-5 and Claude Next, to ensure fair and up-to-date evaluation.

5.6. Threats to Validity

A key threat to the internal validity of AutoQALLMs is its dependence on LLMs for generating Selenium scripts. While it performs well overall, LLMs can lose context in long or complex interactions, leading to incomplete or incorrect test cases. The output quality also heavily depends on prompt design, which may affect consistency across different environments. Moreover, we currently parse only a subset of tags (e.g., links, headings, and images), which limits test coverage. This may result in an incomplete representation of real-world page structures.

Construct validity is limited by AutoQALLMs's ability to convert natural language into actionable test steps reliably. This depends on the model's understanding of dynamic or ambiguous web structures. Prior work, such as GPTDroid [20], has demonstrated that LLMs can handle functionality-aware prompts; however, they sometimes struggle to maintain reasoning over multiple turns.

Technical challenges also arise during DOM scraping, especially for JavaScript-heavy pages or those with anti-scraping protections, which can result in the omission of key elements. To address these issues, future work could explore memory-augmented models and explainable AI techniques to enhance the reasoning and traceability of the generated steps.

Finally, since AutoQALLMs currently focus on web GUI testing, their generalizability to other domains, such as mobile apps or chatbot interfaces, remains an open question.

Real-world testing also involves complex user behaviours, so integrating multimodal inputs and modelling user actions could further enhance test realism and coverage.

6. Conclusions and Future Work

This study demonstrated that the proposed AutoQALLMs can be used to generate Selenium test scripts that are fast, scalable, and easy to maintain. The results indicate that AutoQALLMs scripts achieved competitive UI coverage while drastically reducing the time needed to create and run tests. Experts' feedback confirmed that this method can support faster regression testing and make automation more accessible, particularly for teams with less technical skill. However, the scripts sometimes failed when webpages changed or when more thorough checks were required. By contrast, Monkey Testing was less valuable due to its random behaviour and limited test coverage.

The comparison also showed that manual scripts had the lowest failure rates and adapted best to UI changes; however, they required more time to write and maintain. AutoQALLMs scripts offered a good balance between speed and reliability. They were slightly more prone to errors, but could be quickly fixed or regenerated. Expert feedback confirmed that this approach is beneficial in agile environments and can enhance overall testing efficiency when used in conjunction with human review. Our comparative analysis showed that Claude 4.5 attained marginally higher coverage, but GPT-4 demonstrated the best trade-off between accuracy, generation cost, and stability, reaffirming its role as the core model of AutoQALLMs.

In the future, we plan to improve the accuracy and stability of AutoQALLMs scripts. Fine-tuning the LLMs for test automation tasks may help reduce errors. Testing on larger and more complex applications will also help evaluate how well the system scales in real-world projects. Another helpful step is to connect LLM-based testing with tools like JMeter or Gatling to support performance testing. Adding self-healing features could help the system adjust to UI changes without manual updates. This is a key area of research, with recent approaches using LLMs to intelligently re-locate web elements after a UI change, significantly improving test script robustness [29]. Future work will also explore semi-automated assertion generation to enable basic behavioural validation. By inferring expected outcomes from HTML5 attributes and JavaScript-based interface cues, AutoQALLMs can extend beyond structural testing to verify user interactions and page responses. Furthermore, future work could incorporate visual understanding by leveraging Large Vision-Language Models (LVLMs). As demonstrated by the VETL framework [32], LVLMs can analyze screenshots to overcome the limitations of ambiguous HTML and better understand complex, dynamic user interfaces, representing a promising path to enhance the robustness of AutoQALLMs. Comparing this approach with traditional frameworks in terms of cost, speed, and resource use will also help guide its use in industry. These steps may lead to more reliable and intelligent testing systems with less human effort. AutoQALLMs demonstrate the growing role of LLMs in shaping the future of scalable and intelligent software testing.

Author Contributions: Conceptualization, M.Y. and J.A.K.; methodology, M.Y. and J.A.K.; software, S.M.; validation, S.M., M.Y. and J.A.K.; formal analysis, S.M., T.M. and A.M.; investigation, S.M., A.M. and N.P.; writing—original draft preparation, S.M., M.Y. and J.A.K.; writing—review and editing, T.M., A.M. and N.P.; supervision, M.Y. and J.A.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding authors.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Langugae Model
GPT	Generative Pre-trained Transformer
DOM	Document Object Model
LSTM	Long Short Term Memory
CNNs	Convolutional Neural Networks
ML	Machine Learning
DL	Deep Learning

Appendix A. Expert Survey Questionnaire

To collect expert feedback for validating AutoQALLMs, we designed a short survey focused on five key areas: script generation speed, test accuracy, coverage, adaptability, and scalability. The questionnaire was distributed via email and responses were collected using Google Forms.

Appendix A.1. Survey Questions

1. How would you rate the speed of script generation for AutoQALLMs compared to manual testing?
2. How reliable were the AutoQALLMs-generated scripts across different test scenarios?
3. How would you compare the UI coverage achieved by AutoQALLMs with manual or monkey testing?
4. How adaptable were the AutoQALLMs scripts when the web UI changed?
5. How scalable do you find AutoQALLMs for enterprise-level testing compared to traditional methods?
6. Do you see value in using LLM-based tools like AutoQALLMs in real-world software testing workflows?
7. Any additional comments, suggestions, or concerns?

Appendix A.2. Expert Scoring Rubric

The following rubric was shared with all five domain experts to ensure consistency in qualitative evaluation. Each metric, readability, adaptability, maintainability, and scalability, was rated on a five-point scale. (1 = Poor, 5 = Excellent) based on the following descriptions:

1. **1—Poor:** Very low quality; unclear, unstable, or non-functional.
2. **2—Fair:** Partially functional but lacks clarity or stability.
3. **3—Moderate:** Acceptable performance; requires moderate revision or debugging.
4. **4—Good:** Clear, functional, and stable with minor improvements needed.
5. **5—Excellent:** Highly readable, adaptable, and scalable with minimal intervention.

References

1. Durelli, V.H.; Durelli, R.S.; Borges, S.S.; Endo, A.T.; Eler, M.M.; Dias, D.R.; Guimarães, M.P. Machine learning applied to software testing: A systematic mapping study. *IEEE Trans. Reliab.* **2019**, *68*, 1189–1212. [CrossRef]
2. Doğan, S.; Betin-Can, A.; Garousi, V. Web application testing: A systematic literature review. *J. Syst. Softw.* **2014**, *91*, 174–201. [CrossRef]
3. Nguyen, D.P.; Maag, S. Codeless web testing using Selenium and machine learning. In Proceedings of the ICSOFT 2020: 15th International Conference on Software Technologies, Online Event, 7–9 July 2020 ; pp. 51–60.

4. Paul, N.; Tommy, R. An Approach of Automated Testing on Web Based Platform Using Machine Learning and Selenium. In Proceedings of the 2018 International Conference on Inventive Research in Computing Applications (ICIRCA), Coimbatore, India, 11–12 July 2018; pp. 851–856.
5. Briand, L.C. Novel applications of machine learning in software testing. In Proceedings of the 2008 The Eighth International Conference on Quality Software, Oxford, UK, 12–13 August 2008; pp. 3–10.
6. Khaliq, Z.; Farooq, S.U.; Khan, D.A. Artificial intelligence in software testing: Impact, problems, challenges and prospect. *arXiv* **2022**, arXiv:2201.05371. [CrossRef]
7. Talasbek, A. Artificial AI in Test Automation: Software Testing opportunities with Openai Technology-Chatgpt. *Suleyman Demirel Univ. Bull. Nat. Tech. Sci.* **2023**, *62*, 5–14.
8. Chen, F.K.; Liu, C.H.; You, S.D. Using Large Language Model to Fill in Web Forms to Support Automated Web Application Testing. *Information* **2025**, *16*, 102. [CrossRef]
9. Li, T.; Huang, R.; Cui, C.; Towey, D.; Ma, L.; Li, Y.F.; Xia, W. A Survey on Web Application Testing: A Decade of Evolution. *arXiv* **2024**, arXiv:2412.10476. [CrossRef]
10. Ayenew, H.; Wagaw, M. Software Test Case Generation Using Natural Language Processing (NLP): A Systematic Literature Review. *Artif. Intell. Evol.* **2024**, *5*, 1–10. [CrossRef]
11. Dawei, X.; Liqiu, J.; Xinpeng, X.; Yuhang, W. Web application automatic testing solution. In Proceedings of the 2016 3rd International Conference on Information Science and Control Engineering (ICISCE), Beijing, China, 8–10 July 2016; pp. 1183–1187.
12. Gatla, G.; Gatla, K.; Gatla, B.V. Codeless Test Automation for Development QA. *Am. Sci. Res. J. Eng. Technol. Sci.* **2023**, *91*, 28–35.
13. Jiang, J.; Wang, F.; Shen, J.; Kim, S.; Kim, S. A survey on large language models for code generation. *arXiv* **2024**, arXiv:2406.00515. [CrossRef]
14. Khan, J.A.; Qayyum, S.; Dar, H.S. Large Language Model for Requirements Engineering: A Systematic Literature Review. *Res. Sq.* **2025**. [CrossRef]
15. Zhou, X.; Cao, S.; Sun, X.; Lo, D. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 145. [CrossRef]
16. Leotta, M.; Yousaf, H.Z.; Ricca, F.; Garcia, B. AI-generated test scripts for web e2e testing with ChatGPT and copilot: A preliminary study. In Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, Salerno, Italy, 18–21 June 2024; pp. 339–344.
17. Schäfer, M.; Nadi, S.; Eghbali, A.; Tip, F. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Softw. Eng.* **2023**, *50*, 85–105. [CrossRef]
18. Wang, J.; Huang, Y.; Chen, C.; Liu, Z.; Wang, S.; Wang, Q. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936. [CrossRef]
19. Deng, G.; Liu, Y.; Mayoral-Vilches, V.; Liu, P.; Li, Y.; Xu, Y.; Zhang, T.; Liu, Y.; Pinzger, M.; Rass, S. {PentestGPT}: Evaluating and harnessing large language models for automated penetration testing. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 847–864.
20. Liu, Z.; Chen, C.; Wang, J.; Chen, M.; Wu, B.; Che, X.; Wang, D.; Wang, Q. Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13.
21. Job, M.A. Automating and optimizing software testing using artificial intelligence techniques. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*. [CrossRef]
22. Wang, F.; Kodur, K.; Micheletti, M.; Cheng, S.W.; Sadasivam, Y.; Hu, Y.; Li, Z. Large Language Model Driven Automated Software Application Testing. *Technical Disclosure Commons*, 26 March 2024. Available online: https://www.tdcommons.org/dpubs_series/6815 (accessed on 28 August 2025).
23. Sherifi, B.; Slhoub, K.; Nembhard, F. The Potential of LLMs in Automating Software Testing: From Generation to Reporting. *arXiv* **2024**, arXiv:2501.00217. [CrossRef]
24. Khaliq, Z.; Farooq, S.U.; Khan, D.A. A deep learning-based automated framework for functional User Interface testing. *Inf. Softw. Technol.* **2022**, *150*, 106969. [CrossRef]
25. Ale, N.K.; Yarram, R. Enhancing Test Automation with Deep Learning: Techniques, Challenges and Future Prospects. In Proceedings of the CS & IT Conference Proceedings, 8th International Conference on Computer Science and Information Technology (COMIT 2024), Chennai, India, 17–18 August 2024; Volume 14.
26. Pei, K.; Cao, Y.; Yang, J.; Jana, S. Deepxplore: Automated whitebox testing of deep learning systems. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October 2017; pp. 1–18.
27. Zimmermann, D.; Koziolek, A. Gui-based software testing: An automated approach using gpt-4 and selenium webdriver. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Luxembourg, 11–15 November 2023; pp. 171–174.

28. Cavalcanti, A.R.; Accioly, L.; Valença, G.; Nogueira, S.C.; Morais, A.C.; Oliveira, A.; Gomes, S. Automating Test Design Using LLM: Results from an Empirical Study on the Public Sector. In Proceedings of the Conference on Digital Government Research, Porto Alegre, Brazil, 9–12 June 2025; Volume 1.
29. Nass, M.; Alégroth, E.; Feldt, R. Improving web element localization by using a large language model. *Softw. Testing, Verif. Reliab.* **2024**, *34*, e1893. [CrossRef]
30. Le, N.K.; Bui, Q.M.; Nguyen, M.N.; Nguyen, H.; Vo, T.; Luu, S.T.; Nomura, S.; Nguyen, M.L. Automated Web Application Testing: End-to-End Test Case Generation with Large Language Models and Screen Transition Graphs. *arXiv* **2025**, arXiv:2506.02529.
31. Li, T.; Cui, C.; Huang, R.; Towey, D.; Ma, L. Large Language Models for Automated Web-Form-Test Generation: An Empirical Study. *arXiv* **2024**, arXiv:2405.09965. [CrossRef]
32. Wang, S.; Wang, S.; Fan, Y.; Li, X.; Liu, Y. Leveraging large vision-language model for better automatic web GUI testing. In Proceedings of the 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), Flagstaff, AZ, USA, 6–11 October 2024; pp. 125–137.
33. Garousi, V.; Joy, N.; Keleş, A.B. AI-powered test automation tools: A systematic review and empirical evaluation. *arXiv* **2024**, arXiv:2409.00411. [CrossRef]
34. Khankhoje, R. AI-Based test automation for intelligent chatbot systems. *Int. J. Sci. Res. (IJSR)* **2023**, *12*, 1302–1309. [CrossRef]
35. Chapman, C.; Stolee, K.T. Exploring regular expression usage and context in Python. In Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016; pp. 282–293.
36. *ISO/IEC 25010:2011*; Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. ISO: Geneva, Switzerland, 2011. Available online: https://www.iso.org/standard/35733.html (accessed on 15 September 2025).
37. Buse, R.L.; Weimer, W.R. Learning a metric for software readability. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, GA, USA, 9–14 November 2008; pp. 100–109.
38. Bondi, A.B. Characteristics of scalability and their impact on performance. In Proceedings of the 2nd International Workshop on Software and Performance, Ottawa, ON, Canada, 17–20 September 2000; pp. 195–203.
39. Android Developers. UI/Application Exerciser Monkey. 2025. Available online: https://developer.android.com/studio/test/other-testing-tools/monkey (accessed on 25 September 2025).
40. OpenAI. GPT-4-Turbo Pricing and Token Usage Documentation. 2025. Available online: https://openai.com/pricing (accessed on 2 November 2025).
41. Claude API Pricing. Available online: https://www.claude.com/pricing#api (accessed on 10 November 2025).
42. xAI API Models and Pricing. Available online: https://docs.x.ai/docs/models (accessed on 10 November 2025).
43. The OWASP Foundation. OWASP AppSensor Project. 2014. Available online: https://owasp.org/www-project-appsensor/ (accessed on 5 November 2025).
44. Rennhard, M.; Kushnir, M.; Favre, O.; Esposito, D.; Zahnd, V. Automating the detection of access control vulnerabilities in web applications. *SN Comput. Sci.* **2022**, *3*, 376. [CrossRef]
45. Pichiyan, V.; Muthulingam, S.; Sathar, G.; Nalajala, S.; Ch, A.; Das, M.N. Web scraping using natural language processing: Exploiting unstructured text for data extraction and analysis. *Procedia Comput. Sci.* **2023**, *230*, 193–202. [CrossRef]