Automated Generation of End-to-End Web Test Cases via a Generic AI Agent: A Comparative Study of DeepSeek V3 and Claude Sonnet 4

Caio E. O. Monteiro[1,2]
[1]Departamento de Computação,
Universidade Federal de São Carlos
São Carlos, São Paulo, Brazil
caiomonteiropro@gmail.com

Lucca R. Guerino[1,2]
[1]Departamento de Computação,
Universidade Federal de São Carlos
São Carlos, São Paulo, Brazil
luccaguerino@estudante.ufscar.br

Guilherme F. Fernandes
[2]Centro de Excelência em Inteligência Artificial,
Universidade Federal de Goiás
Goiânia, Goiás, Brazil

Marcos H. Pereira
[2]Centro de Excelência em Inteligência Artificial,
Universidade Federal de Goiás
Goiânia, Goiás, Brazil

Juliana P. de Souza-Zinader
[2]Centro de Excelência em Inteligência Artificial,
Universidade Federal de Goiás
Goiânia, Goiás, Brazil

Renata D. Braga
[2]Centro de Excelência em Inteligência Artificial,
Universidade Federal de Goiás
Goiânia, Goiás, Brazil

Viviane C. B. Pocivi
[2]Centro de Excelência em Inteligência Artificial,
Universidade Federal de Goiás
Goiânia, Goiás, Brazil

Auri M. R. Vincenzi[3,2]
[3]Faculdade de Engenharia,
Universidade do Porto
Porto, Porto, Portugal
auri@fe.up.pt

ABSTRACT

Web applications are widespread and can be accessed from anywhere, in theory, using a web browser on a computer or smartphone. Primarily due to the diversity of web browsers and frameworks available for developing web application interfaces, testing such applications is a challenging task. With the advent of large language models, several works are utilizing them to automate software engineering tasks, including test case generation. This use of LLMs for test case generation prioritizes unit testing. More recently, we have seen the advent of Generic Artificial Intelligence Agents, which are tools that utilize LLMs and also possess the ability to run additional tools, such as cloning repositories, navigating websites, and compiling programs. In this work, which is part of a research and development project, we evaluate a specific Generic AI Agent Assistant regarding its capability to navigate web applications and create fully automated end-to-end test cases, utilizing Selenium WebDriver and JUnit 5 framework. Results show that, considering a set of nine websites, in overall end-to-end test case generation, Suna configured with DeepSeek V3 produced 165 successful test cases out of 481 generated tests, a success rate of 34.3%. On the other hand, Suna configured with Claude Sonnet 4 produced 336 successful test cases out of 479 generated tests, a success rate of 70.1%, which is very impressive, mainly due to the complexity of creating end-to-end testing. In terms of cost, we used a free and a paid LLM model. The paid model generates successful test cases at an average price of $ 0.15 per test case.

In: Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia'2025). Rio de Janeiro, Brazil. Porto Alegre: Brazilian Computer Society, 2025.

ISSN 2966-2753

KEYWORDS

web applications, test web applications, automatic test case generation, end-to-end testing

1 INTRODUCTION

A web application operates in a client-server environment and is accessible via the Internet or a local network, using standard protocols such as HTTP or HTTPS for communication [16].

The visual interface of this system is represented by a Graphical User Interface (GUI), which consists of various interactive components such as buttons, text fields, menus, and other visual elements that enable user interaction with the system [6]. According to Martinez-Caro et al. [19], the number of websites available on the Internet has surpassed 1 billion, while the number of globally connected users has reached approximately 4 billion.

Testing involves evaluating whether it meets the requirements for which it was designed, with the primary goal of identifying failures, errors, or gaps in the specifications. In addition, testing helps reduce maintenance costs by identifying and resolving application issues early, thus preventing more serious problems in the future [35]. Tests can be performed either manually or automatically, aiming to ensure the correct functioning and overall quality of the deployed system [8].

Among automated tests, end-to-end testing stands out due to its complexity, as it verifies the complete system by simulating the end user's behavior in real-world scenarios. These tests assess the entire workflow of the application, from the user interface to communication with external services and databases, ensuring that all system components work together as expected [3, 13].

Among the several challenges for automating web testing, such as the varying behavior of different browsers and the diversity of

technologies used for graphical user interface (GUI) implementation of web applications, there is also the need to locate web page elements to interact with them automatically. For this, it is expected that each element has one or more locators and, in general, depending on the locator strategy adopted during development, even small changes to the web application GUI will result in script testing failures Ricca et al. [27].

Large Language Models (LLMs) are artificial intelligence systems, typically based on deep neural networks, designed to process and generate natural language text [5]. They change the way we do software engineering [11, 23, 32], and they are also extensively studied to support software testing automation, especially at the unit testing level [31, 35].

These models have proven to be highly useful in the context of web system testing, offering new approaches to automate and enhance testing efficiency. Their applications include automatic test case generation, user interface interaction testing, error analysis, result validation, security testing, and vulnerability detection, as well as improvements in element locators, among others [15, 21, 36].

This work is part of a research and development (R&D) project in partnership with a company that aims to automate the amplification and generation of automated testing for web applications. We began investigating the use of LLM prompts to create fully automated, end-to-end testing for web applications.

However, despite recent advances, a significant gap remains in the practical application of LLMs for generating robust, maintainable, and GUI-resilient end-to-end tests — especially in dynamic web environments, where frequent interface changes lead to test fragility. This research is a first step towards addressing this gap by proposing and evaluating an LLM-based approach to enhance the automation and resilience of web application testing.

Initial efforts to interact directly with various LLMs via chat-based interfaces consistently produced unsatisfactory test cases, many of which failed even to compile. This was observed even when using state-of-the-art models such as DeepSeek V3 and Anthropic Claude Sonnet 4. In parallel with the evolution of LLMs, numerous generic AI agents have emerged. These systems integrate LLMs with external tools, enabling them to autonomously perform complex tasks that exceed the scope of LLMs alone. Notable examples include proprietary agents such as Manus[1] and Genspark[2], as well as open-source alternatives like OpenManus[3] and Suna[4].

Motivated by the limitations encountered when relying solely on LLMs, this study explores the use of a generic AI agent — Suna[5] — to automatically generate Selenium WebDriver test cases for a set of nine real-world web applications. Suna is configured using two LLM models accessed via OpenRouter[6]: DeepSeek V3 0324 (`deepseek/deepseek-chat-v3-0324:free`) and Claude Sonnet 4 (`anthropic/claude-sonnet-4`). For brevity, we refer to these models as DeepSeek V3 and Claude Sonnet 4 throughout the remainder of this paper.

We intend to answer the following research questions:

- **RQ1**: What is the capacity of Suna to create successful test cases for automating web-application testing?
- **RQ2**: What are the common faults resulting from automated test cases for web applications created by Suna?
- **RQ3**: What is the cost/benefit relation of paid versus free models on generating test cases for web-application testing using Suna?

The contributions of this work are:

- The evaluation of a generic AI agent (Suna) capability on generating automated test cases for testing web applications in a fully automated way.
- The analysis of types of errors and failures that Suna's generated test cases produce when configured with different LLM models.
- A cost-benefit analysis considering the number of successful test cases and the money spent to generate them.

The remainder of this text is organized as follows. In Section 2, we present the background required to understand the paper. In Section 3 we describe the related work. In Section 4, we describe the experimental design and how we collected the data to answer the research questions. In Section 5, we answer the research questions and present a discussion about our results. In Section 6, we present some lessons learned in this process. Finally, in Section 7, we conclude the paper and present some future work we intend to continue developing.

2 BACKGROUND

This section reviews the core concepts relevant to our work, including the architecture and challenges of web applications, key principles of software testing, and the respective roles of Large Language Models (LLMs) and generic AI agents in software engineering.

2.1 Web Applications

The proliferation of online services has made developing scalable, robust web applications a fundamental priority for competitive organizations [19]. Web systems have two main layers: the front-end — the user-facing interface — and the back-end, which executes business logic and manages data [1, 6]. Frameworks streamline development by providing reusable components and conventions that accelerate implementation [6].

Architectural styles such as monolithic, microservices, and serverless designs define how these layers interact, influencing scalability and maintainability [12, 34]. Despite architectural advances, automating software testing for web systems remains challenging. Common issues include test fragility when GUI elements change, overdependence on specific implementations, and incomplete test coverage [27]. Moreover, web interfaces are dynamic and primarily designed for human interaction, making element identification and automation difficult [13, 37]. To mitigate this, practitioners use tools like Selenium, Playwright, and Cypress, with Selenium being the most established [25].

[1] https://manus.im/
[2] https://www.genspark.ai/
[3] https://openmanus.org/
[4] https://www.suna.so/
[5] https://github.com/kortix-ai/suna
[6] https://openrouter.ai/

2.2 Software Testing

Software testing ensures reliability and early defect detection but often faces practical constraints such as limited time or resources [25]. Core activities include defining test criteria, designing and executing test cases, monitoring progress, documenting outcomes, and closure [4]. Two main strategies apply [2]: black-box testing, which evaluates external behavior, and white-box testing, which analyses internal code and logic.

Testing occurs at multiple levels—unit, integration, system, and acceptance (end-to-end)—to expose defects early [39]. Automated testing, enabled by scripts or tools, supports rapid, repeatable test execution and broad coverage, becoming crucial for regression testing [20, 37]. Manual testing remains valuable for exploratory and usability evaluation but is slower and more error-prone [3, 38].

2.3 LLMs and Generic AI Agents

Artificial Intelligence (AI) is transforming software testing through automated test generation and fault localization [10, 11, 32]. Large Language Models (LLMs), based on transformer architectures, excel at natural language and code generation tasks [18, 23, 26, 33, 35]. However, early experiments using Deepseek V3 revealed that purely conversational interactions often yield flawed or incomplete test scripts, exposing the limitations of prompt-based use.

Generic AI agents extend LLMs by providing memory, planning, and tool integration [9, 11, 18]. As shown in Figure 1, they typically include a Planner that decomposes complex tasks, an Evaluator that ensures consistency and quality, and an Executor that performs actions using available tools. These agents orchestrate dynamic workflows beyond simple language modeling. Recent platforms such as Manus, OpenManus, Genspark, and Suna exemplify this trend. Our research explores Suna's capacity to autonomously generate Selenium WebDriver tests with JUnit 5 in a black-box mode, given only the target web application's URL.

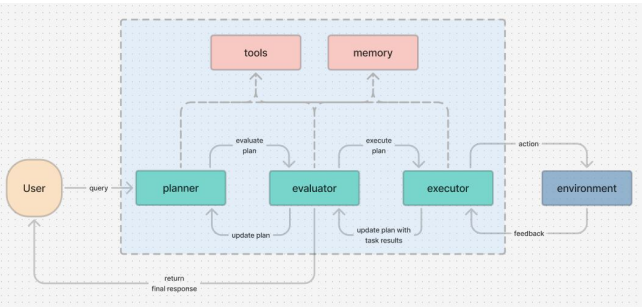

Figure 1: General architecture of generic AI agent tools[7].

3 RELATED WORK

As highlighted by Wang et al. [35], several initiatives have been proposed for utilizing LLMs to support software testing activities, including automatic test case generation. The vast majority of these studies are related to the generation of automated unit testing.

[7]Picture extracted from http://bit.ly/3HsoXCT

Roychowdhury et al. [28] describe an innovative approach that combines static analysis with large language models (LLMs) for the automatic generation of unit tests in Java. The authors propose enriching the prompt with precise and concise information automatically extracted through static code analysis. This technique enables LLMs to generate syntactically correct unit tests, even for newly created or highly complex methods, as demonstrated in both commercial and open-source Java projects. The main limitation identified is the exclusive focus on test generation, without assessing aspects like code coverage, test smells, or assertion effectiveness. Static analysis itself is restricted in handling dynamic dependencies, and the approach's applicability to other programming languages remains untested. Additionally, integration with in-context learning or fine-tuning methods has not yet been pursued, and the method's usefulness beyond unit test generation—such as for integration or system tests—has not been validated.

El Haji et al. [7] empirically evaluate GitHub Copilot's ability to generate unit tests in Python, considering different contexts (with or without an existing test suite) and varying styles of comments in test methods. The authors analyzed 290 tests generated by Copilot based on 53 real tests from seven open-source projects. They found that 45.28% of the tests generated within the context of an existing suite were usable, whereas only 7.55% were usable outside of that context. The useful tests typically mimicked other tests in the same file, suggesting that Copilot heavily relies on the immediate context to generate effective test cases. A key limitation is the number of unusable tests — those with syntax or runtime errors — mainly when Copilot operates without context, with nearly 92.5% of generated tests affected. The model frequently produces hallucinated, or fabricated, code snippets. The evaluation focused on whether the tests could be used immediately, rather than on their fault-detection capability. Additionally, the study was restricted to Python, limiting the generalization to other languages, domains, or testing levels.

Considering testing at the system level, the study by Wang et al. [36] proposes the Vision-Enhanced Test Loop (VETL), a technique for automated testing of graphical user interfaces (GUIs) that combines LLMs with Large Vision-Language Models (LVLMs). The study highlights that traditional methods, such as WebExplor, struggle to generate coherent inputs and to correctly identify interactive elements, particularly given the complexity and dynamic nature of modern web applications. VETL frames the element selection task as a Visual Question Answering (VQA) problem, where the model receives interface screenshots and determines which element to interact with based on visual context. Furthermore, the technique adopts an exploration mechanism based on multi-armed bandits, which balances the discovery of new states with efficient interaction with interface components. Experimental results show that VETL outperforms WebExplor by detecting 25% more unique actions on benchmark websites and uncovering functional bugs in popular commercial sites. Although interesting, test generation is an interactive task, and we intend to evaluate how it can be fully automated, minimizing human intervention as much as possible.

Pan and Zhang [24] propose the MoT prompting technique to enhance code generation by LLMs. Inspired by the modularization principles of traditional software engineering, MoT structures the reasoning process into a Multi-Level Reasoning Graph (MLR Graph), which organizes problem-solving steps across three hierarchical

levels: high-level, intermediate, and detailed. This modularized framework enables LLMs to better understand complex problems, resulting in more accurate and well-structured code. Experiments conducted with the GPT-4o-mini and DeepSeek-R1 models on six widely used benchmarks (such as HumanEval, HumanEval+, MBPP, among others) demonstrated that MoT significantly outperforms other prompting approaches. As limitations, the technique requires the prior construction of a Multi-Level Reasoning Graph (MLR Graph), which introduces additional complexity. Despite encouraging results, it faces issues with scalability and computational demands, and may not readily extend to tasks such as object-oriented or multi-file programming. The study did not evaluate how readable or maintainable the generated code is for developers. Additionally, the approach relies on models with strong capabilities in logical decomposition and semantic analysis.

The study by Olianas et al. [22] introduces a new tool called TESTQUEST, a plugin developed in Kotlin for IntelliJ IDEA, designed to enhance the quality of locators and page objects in web testing. This tool seeks to optimize these elements during the software testing process. The authors note that manually creating test cases, even with the aid of auxiliary tools, can be a time-consuming and labor-intensive task. To mitigate this issue, automatic test generation techniques have been explored, including systematic exploration of web pages, scriptless approaches based on graphical user interface (GUI) modifications, and reinforcement learning algorithms. However, most of these techniques heavily rely on locators, which are considered one of the primary sources of fragility in web testing, as they are directly tied to the DOM structure. Consequently, even minor changes in the GUI during software evolution can compromise locators, breaking their association with page objects and impairing test functionality. To address these challenges, the work introduces the design pattern known as Page Object Model (POM) as a practical approach, adding a layer of abstraction between the test logic and the structure of web pages. Furthermore, it is highlighted that gamification has gained prominence as a strategy to make testing activities more engaging by incorporating game-inspired elements into the software testing context.

There are also other works relating to the use of LLM for testing generation for web applications, such as the work of Li et al. [17]. They report some success, but also encounter problems, such as the phenomenon of hallucination in LLMs, which typically occurs when the context window is extrapolated. The LLM becomes confused and is unable to answer questions accurately.

Considering the published work in the field of software test automation using large language models, we understand that this is the first to comprehensively evaluate an AI agent in the automatic generation of test cases for web applications, based solely on a single command-line prompt, without providing any additional instructions such as page credentials or other essential components required for server navigation.

4 EXPERIMENTAL DESIGN

Our experiment aims to investigate whether a Generic AI Agent Assistant, named Suna, can produce correct Selenium Webdriver test cases for a set of websites, traditionally used for web automation testing tools. We chose Suna because it is an open-source Generic AI Agent, and it is possible to configure it to use different LLM models. Therefore, we designed our experiment to evaluate the performance of two LLMs in generating test cases automatically, using the Suna orchestration layer.

As illustrated in Figure 2, the workflow begins with a natural language prompt sent to Suna (Step 1), which creates a task list delegating some of them to the selected LLM via the OpenRouter API[8]. The response contains the generated Selenium WebDriver test set (Step 2). The generated test set is included in a Maven project for test compilation, execution, and report generation (Step 3). Finally, reports are analyzed to extract relevant information to answer our research questions (Step 4). This automated pipeline enables consistent and reproducible test case generation and execution, supporting both qualitative and quantitative evaluations of the models involved.

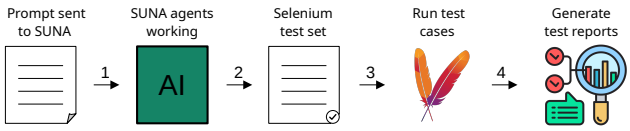

Figure 2: Workflow for LLM-based test case generation and execution.

We ran this pipeline five times on alternate days for each model and each of the websites under testing. Each sequence was conducted on a different day to capture potential variations in model behavior over time—a concern raised by Sallou et al. [30], who highlight factors such as internal updates, server load fluctuations, and dynamic model adjustments.

Each day, one sequence was executed for both models. For example, on the first day, Suna configured with DeepSeek V3 or Claude Sonnet 4, each of which applied to the same set of nine websites, labeled as WS-01 to WS-09 (see Table 1), aiming to create a valid test set for each website. On the second day, the process is repeated to make the second sequence of test sets for each website, and so on, until the fifth day.

Thus, each model was evaluated through 45 website interactions (5 sequences × 9 sites), allowing for a comparative analysis. Repeating the sequences with the same prompt was intended to observe response variability, taking into account the stochastic nature of LLMs and the potential for hallucinations. Although Sallou et al. [30] recommend a larger number of repetitions per scenario — such as ten — we stopped after the fifth run due to cost constraints on using the paid model.

The tests were conducted through the Suna AI platform, an environment that enables the integration of LLMs into test automation workflows. Suna was selected following an exploratory search for tools aligned with the study's goals, standing out for its simplicity and support for the selected models. We analyzed other alternatives: Manus is a proprietary generic agent, expensive, and does not allow for choosing the underlying LLM model — the same limitation applies to Genspark. OpenManus, which is also open-source, was tried with the same prompt presented in Listing 1 and the same

[8]https://openrouter.ai/

LLM models, presenting inferior quality. Therefore, we decided to use Suna in our experiment.

We submit the same prompt (Listings 1) to Suna, once configured with DeepSeek V3, and others with Claude Sonnet 4, to evaluate the success of creating valid Selenium Webdriver test sets using the JUnit 5 framework. All outputs were stored for subsequent qualitative and quantitative analysis. All generated data is available in a public repository[9].

4.1 Web sites

Table 1 provides an overview of all websites used as subjects in this experiment. For each site, it outlines the type of application, the specific elements used for testing, and the reasons why the site was chosen for inclusion in the study. We decided to use these websites because they are commonly referred to in studies and web testing automation experiments. Moreover, they represent different types of web applications or have varying resources for evaluating the capabilities of web testing tools.

4.2 Prompt definition

The success of LLM in performing a task correctly is highly dependent on the prompt and the context it brings. In the case of a Generic AI Agent, this is not different. Therefore, we conducted an empirical evaluation of different prompt versions until we identified a minimal set of instructions that produced satisfactory results. We then decided to use this resultant prompt in our experiment. The complete prompt is in Listing 1. It is essential to note that we consistently submit the same prompt to Suna, regardless of the LLM model configuration. There are two placeholders in this prompt: `{URL}`, replaced by the main website URL; and `{PACKAGE_NAME}`, which defines the Java package in which the test files would be generated to ease automation and data collection.

```
You are a specialized web tester. Navigate through the
 page of the main URL provided below and build test
 cases using Selenium WebDriver with Java and JUnit 5 to
 test this page and all other pages one level below it,
 including external links present on the website under
 testing.
Main URL: {URL}
As output, create the .java file  with the complete
 test set. Save it to the workspace.
The test set file must be in a package named
 ***{PACKAGE_NAME}***
Each test must be complete.
DO click on an HTML element or fill it if necessary.
DO click on buttons.
DO include asserts.
DO create test cases per page that cover all clickable
 elements.
DO create only JUnit test files. DO NOT create the
 complete Maven project.
Save all generated files to the workspace.
```

Listing 1: LLM Prompt for Initial Test Suite Generation

4.3 Data collection

Table 2 shows the raw data collected from Surefire reports after running "`mvn clean test-compile test`" command. The first column indicates the website and sequence, columns TC, E, F, S, and %S correspond to the number of generated test cases, number of tests with errors, number of tests with failures, number of successful test cases, and the success rate ($(S/TS) * 100$).

Yellow cells highlight the best success rate for a specific website and sequence. Red cells indicate situations where Suna creates test cases, but all are executed with failure or error; gray cells indicate that no test set was generated at all.

5 RESULTS AND DISCUSSION

Based on the data from Table 2, in this section, we discuss the results to answer each research question of interest.

5.1 Capability to create success test cases

As can be observed in Table 2, Suna with DeepSeek V3 and Suna with Claude Sonnet 4 produce almost the same number of test cases in total, 481 for DeepSeek V3 and 479 for Claude Sonnet 4, but for Claude Sonnet 4 336 out of 479 tests run successfully (a success rate above 70%) and for DeepSeek V3, only 165 out of 481 run successfully (34.3% of success rate).

Considering average values, both models achieve a similar success rate of 29% for DeepSeek V3 and 33.8% for Claude Sonnet 4. This is mainly due to several executions Claude Sonnet 4 that generated no valid test cases. DeepSeek V3 always creates valid test cases[10] ($TC \neq 0$), but not always do their test cases run successfully ($S = 0$). In almost all situations where Claude Sonnet 4 creates valid test cases, they are more successful than DeepSeek V3 test cases. Only in one case, `ws02.seq01` DeepSeek V3 creates more successful test cases than Claude Sonnet 4, with Claude Sonnet 4 generating valid test cases.

Figure 3 summarizes the aggregate success rates per model and website. As can be observed, the models produce different success rates. For some websites, on average, DeepSeek V3 produces more test cases that run successfully, and on others, Claude Sonnet 4 yields better results.

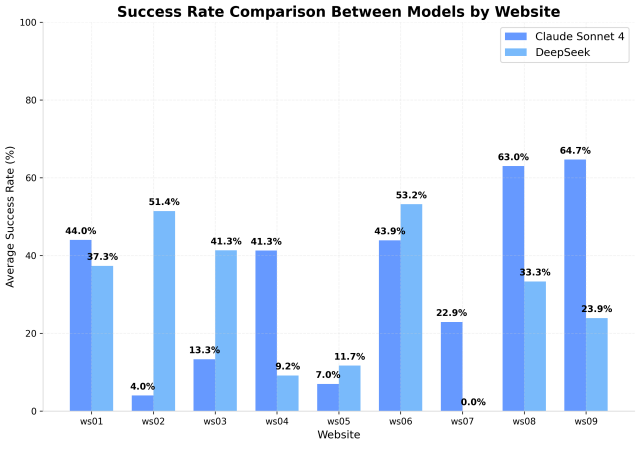

Figure 3: Successful rate comparison by LLM model.

Specifically, Claude Sonnet 4 achieved higher success rates across almost all websites, with notable performance on `ws08` (63.0%) and

[9]https://github.com/aurimrv/suna-selenium/

[10]We refer to valid test cases as the ones generated that compile successfully.

Table 1: List of websites under testing

ID	Website	Type	Elements to Test	Why Use It
WS01	Basic Site	Static	Buttons, navigation, simple form elements	Simple site, useful for testing basic interactions and recording tools
WS02	Swag Labs	Dynamic (E-Commerce)	Login, product listing, shopping cart	Classic automation test environment with multiple UI elements
WS03	ParaBank	Dynamic (Banking)	Login, account creation, transfers, statements	Simulates banking operations, ideal for testing form-based applications and authentication
WS04	BugBank	Dynamic (Banking)	Login, transactions, user registration, statement	Modern system with real banking features, excellent for end-to-end testing
WS05	XPath Practice Page	Static + Dynamic	Elements with various attributes, dynamic IDs, XPath location	Excellent for practicing element location via XPath and complex selectors
WS06	Demo AUT	Static	Forms, required fields, simple validations	Basic test automation environment with input fields and form validation
WS07	Select2 3.5.3	Dynamic (UI Library)	Selection fields, searchable lists, custom elements	Useful for testing complex form widgets like 'select' with search and multi-selection
WS08	Login Wt	Static + Dynamic	Login, field validation, error messages	Simple and straightforward, great for validating authentication and feedback messages
WS09	Customer Service Center TAT	Static	Contact form, validations, file upload	Simulates a customer service center with common form elements, ideal for functional testing

`ws09` (64.7%). On the other hand, DeepSeek V3 presented more limited performance, with a maximum success rate of 53.22% on `ws06` and significantly lower (or null) results on others, such as `ws07`. This indicates that the generated test cases seem complementary. For some websites, one LLM yielded better results, while for others, it is essential to explore a different model.

Answering RQ1: The Claude Sonnet 4 model showed a higher average success rate in generating test cases compared to the DeepSeek V3 model. Therefore, the paid model generally yielded better results; however, combining different models for test case generation can be beneficial when the generated test cases complement each other.

5.2 Common Errors and Failures

In the case of DeepSeek V3, from the 481 generated test cases, 236 (49.1%) ran with errors and 80 (16.6%) with failures. JUnit 5[11] distinguishes problems with test cases into these two categories, where "failures" occur when your test cases fail due to issues in assertions; and "errors" represent unexpected errors/exceptions thrown during test execution.

Table 3 details the data about errors and failures per LLM model during test execution. In general, DeepSeek V3 produces more errors in more categories than Claude Sonnet 4, and, in general, there are few intersections of errors produced by test cases generated when Suna is configured with DeepSeek V3 and Claude Sonnet 4. Only in the error types `ElementClickInterceptedException`, `NoSuchElementException`, and `TimeoutException`, both models have some intersection. In the case of DeepSeek V3, 50% of errors are of `NoSuchElementException` and for Claude Sonnet 4, 56% are of `TimeoutException`.

Figure 4 depicts the average errors and failure rates per model per website. In a general view, DeepSeek V3 produces test cases with more errors and failures than Claude Sonnet 4. Claude Sonnet 4 error rate is around 23.4% and failure rate around 6.5% considering all generated test cases. On the other hand, DeepSeek V3 error rate reaches 49.1% and failure rate 16.6%.

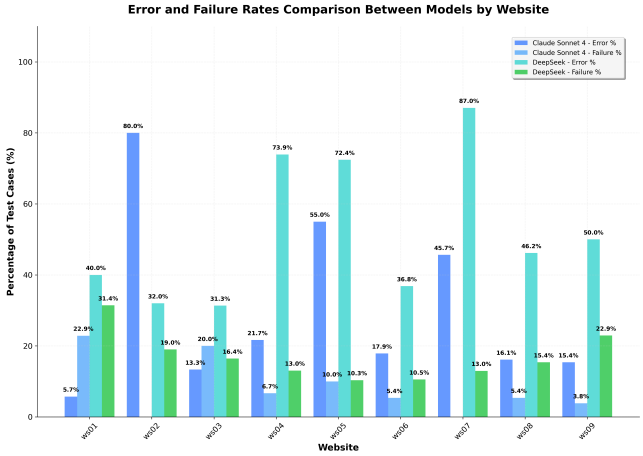

Figure 4: Errors and failures rate comparison.

Considering the complexity of creating test cases for web applications, we did not expect perfect test cases. Moreover, it is essential to identify these limitations and the specific types of errors and failures each model produces, to implement mechanisms that utilize LLMs to potentially correct generated tests based on error messages [14]. Different errors and failures may require different prompt strategies and contexts to resolve the issue. Although we did not implement auto-fix in this experiment, we intend to do so in future work.

Answering RQ2: Both models produce test cases with errors and failures. In general, Suna with Claude Sonnet 4 reduces the error rate by 25.7% and the failure rate by 10.2% compared to Suna with DeepSeek V3. For DeepSeek V3, 50% of errors are related to `NoSuchElementException` and for Claude Sonnet 4 56.3% of errors are related to `TimeoutException`.

[11] https://junit.org/

Table 2: Resultant data for each test set sequence.

Test Set	DeepSeek V3					Claude Sonnet 4				
	TC	E	F	S	%S	TC	E	F	S	%S
ws01.seq01	5	5	0	0	0.0	10	1	3	6	60.0
ws01.seq02	6	0	2	4	66.7	0	0	0	0	0.0
ws01.seq03	5	0	3	2	40.0	10	0	0	10	100.0
ws01.seq04	14	9	5	0	0.0	15	1	5	9	60.0
ws01.seq05	5	0	1	4	80.0	0	0	0	0	0.0
ws02.seq01	5	2	0	3	60.0	20	16	0	4	20.0
ws02.seq02	34	9	10	15	44.1	0	0	0	0	0.0
ws02.seq03	33	10	8	15	45.5	0	0	0	0	0.0
ws02.seq04	9	5	0	4	44.4	0	0	0	0	0.0
ws02.seq05	19	6	1	12	63.2	0	0	0	0	0.0
ws03.seq01	15	6	4	5	33.3	15	2	3	10	66.7
ws03.seq02	23	7	2	14	60.9	0	0	0	0	0.0
ws03.seq03	12	4	2	6	50.0	0	0	0	0	0.0
ws03.seq04	1	1	0	0	0.0	0	0	0	0	0.0
ws03.seq05	16	3	3	10	62.5	0	0	0	0	0.0
ws04.seq01	6	3	1	2	33.3	25	2	1	22	88.0
ws04.seq02	1	1	0	0	0.0	0	0	0	0	0.0
ws04.seq03	8	6	1	1	12.5	20	4	3	13	65.0
ws04.seq04	1	1	0	0	0.0	0	0	0	0	0.0
ws04.seq05	7	6	1	0	0.0	15	7	0	8	53.3
ws05.seq01	10	8	1	1	10.0	20	11	2	7	35.0
ws05.seq02	19	12	2	5	26.3	0	0	0	0	0.0
ws05.seq03	10	9	1	0	0.0	0	0	0	0	0.0
ws05.seq04	1	0	1	0	0.0	0	0	0	0	0.0
ws05.seq05	18	13	1	4	22.2	0	0	0	0	0.0
ws06.seq01	12	10	1	1	8.3	15	2	0	13	86.7
ws06.seq02	9	5	2	2	22.2	30	17	0	13	43.3
ws06.seq03	16	6	1	9	56.3	0	0	0	0	0.0
ws06.seq04	12	0	1	11	91.7	0	0	0	0	0.0
ws06.seq05	8	0	1	7	87.5	67	1	6	60	89.6
ws07.seq01	7	1	6	0	0.0	0	0	0	0	0.0
ws07.seq02	10	10	0	0	0.0	26	17	0	9	34.6
ws07.seq03	7	7	0	0	0.0	0	0	0	0	0.0
ws07.seq04	24	23	1	0	0.0	20	4	0	16	80.0
ws07.seq05	6	6	0	0	0.0	0	0	0	0	0.0
ws08.seq01	5	2	0	3	60.0	26	1	3	22	84.6
ws08.seq02	7	3	1	3	42.9	27	8	0	19	70.4
ws08.seq03	6	5	0	1	16.7	20	1	1	18	90.0
ws08.seq04	4	1	3	0	0.0	0	0	0	0	0.0
ws08.seq05	17	7	2	8	47.1	20	5	1	14	70.0
ws09.seq01	14	4	2	8	57.1	18	2	1	15	83.3
ws09.seq02	8	1	2	5	62.5	20	2	1	17	85.0
ws09.seq03	15	11	4	0	0.0	20	5	1	14	70.0
ws09.seq04	9	8	1	0	0.0	0	0	0	0	0.0
ws09.seq05	2	0	2	0	0.0	20	3	0	17	85.0
Total	481	236	80	165	34.3	479	112	31	336	70.1
AVG	10.7	5.2	1.8	3.7	29.0	10.6	2.5	0.7	7.5	33.8
SD	7.6	4.6	2.1	4.5	28.9	13.5	4.5	1.4	10.9	37.9
AVG ≠ 0	10.7	6.2	2.4	5.9	46.7	21.8	5.3	2.4	15.3	69.1

5.3 Cost and Benefits Relation

Considering the cost-benefit analysis, Claude Sonnet 4, even as a paid model, delivered almost 40% more successful test cases than DeepSeek V3. In total, Claude Sonnet 4 created 336 successful test cases out of 479, and we spent $49.11 to make all these test cases. This gave us a cost per successful test case of $0.15.

Figure 5 illustrates the cost per day of experimentation. Observe that on the first day, Claude Sonnet 4 spent more money, but it was the day when it had more success in creating valid test sets. From the second to the fifth day, it had more failures. Unfortunately, we did not store the Suna logs for all executions. However, we speculate that once we are running Claude Sonnet 4 from OpenRouter, we may encounter instability problems with the model provider used for OpenRouter, and failures may be due to the provider's unavailability rather than a wrong model response.

Table 3: Details about the errors and failures of generated test cases per LLM model

Type	DeepSeek V3		Claude Sonnet 4	
	#	%	#	%
Number of Errors	236	100.0	112	100
NullPointerException	6	2.5	–	–
NoSuchElementException	1	0.4	–	–
ElementClickInterceptedException	6	2.5	20	17.9
ElementNotInteractableException	3	1.3	–	–
InvalidArgumentException	2	0.8	–	–
InvalidElementStateException	–	–	1	0.9
InvalidSelectorException	–	–	1	0.9
NoSuchElementException	118	50.0	25	22.3
NoSuchSessionException	1	0.4	–	–
NoSuchDriverException	20	8.5	–	–
StaleElementReferenceException	2	0.8	2	1.8
UnexpectedTagNameException	1	0.4	–	–
TimeoutException	76	32.2	63	56.3
Number of Failures	80	100.0	31	100.0
AssertionFailedError	80	100.0	31	100.0

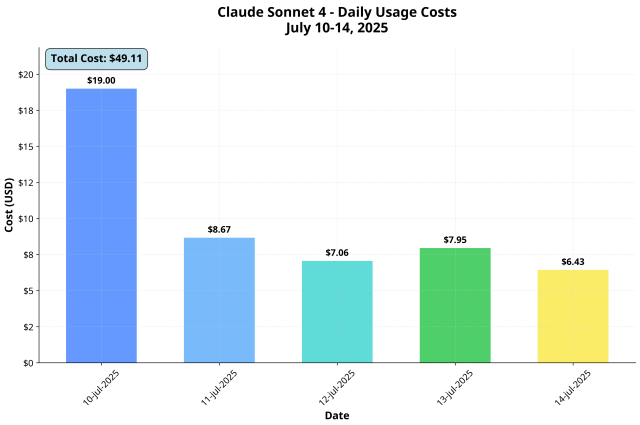

Figure 5: cs cost per day.

On the other hand, we have DeepSeek V3, which can be run for free once you have an OpenRauter account with credits. In this case, we achieved 165 successful test cases out of 481 generated test cases with no additional cost by using the model.

Once models seem complementary, we are in favor of using an incremental testing generation strategy, starting with free models to create initial test cases. Later, we may utilize more specialized prompts and potentially high-quality paid models to obtain additional test cases.

Answering RQ3: At least in this experiment, better results had a higher cost. Claude Sonnet 4 reaches 70% of success rate and consumed 49.11 to accomplish this task, which results in a cost of $0.15 per successful test case. We consider this a reasonable cost/benefit rate mainly due to the complexity of creating successful end-to-end test cases. Nevertheless, DeepSeek V3, which can be run with no cost, can be used first to create an initial test set, and later, we may use a more sophisticated model, like Claude Sonnet 4, in an incremental testing strategy, possibly minimizing the overall cost.

6 LESSONS LEARNED

Using AI agents to generate test cases proved to be more effective than interacting directly with standalone LLMs. One key advantage lies in the agents' ability to orchestrate the prompt, manage execution steps, and apply structured output formatting — features that reduce the likelihood of malformed results. Moreover, platforms like Suna, which integrate LLMs via APIs such as OpenRouter, provide a convenient abstraction layer that simplifies model management and enables experiment traceability, including cost monitoring. Additionally, the open-source nature of the project allows us to create a specialized version explicitly tailored for end-to-end test generation of web applications.

During the experiments, the open-source model DeepSeek V3 demonstrated faster test case generation compared to the commercial Claude Sonnet 4. Initially, this speed appeared to be a strength. However, further analysis revealed that this rapidity often correlated with lower output quality, likely due to the model's limited capacity in handling more complex generation tasks. This was evident in the higher number of failures during test execution, particularly when test cases lacked sufficient structure or completeness.

In contrast, Claude Sonnet 4 exhibited a longer generation time but produced more consistent and higher-quality test cases. Interestingly, while the free model DeepSeek V3 completed all sequences without crashing or halting, Claude Sonnet 4 failed in some sequences — highlighting the stochastic and sometimes unpredictable behavior of LLM-based systems, even when operating under controlled prompts. Although OpenRouter allows us to define a `seed` parameter[12], it is not utilized in this experiment. We intend to explore its use in future experiments to observe whether LLM models exhibit more deterministic behavior when running with the same seed value.

As highlighted by Schmidt [31], the use of AI to assist in software testing activities must be done with care, especially in regulated domains. AI "hallucinations", nondeterministic behavior, compliance and ethics, and bias and fairness are some problems he highlights about the usage of AI tools in supporting test activities. Although he focuses on automatic generation of unit testing with AI support, as presented in this paper, AI can also helps automatic generate test cases for end-to-end scenarios with relatively success but, as stated by Schmidt [31], we also agree that it is very important human experts reviewing the results and guide improvements for a better quality assurance process, i.e., the use of generative AI for software engineering tasks, including test, must be human-centered [29].

Finally, it is essential to note that we generate test cases in a black-box way, without providing any additional information about each website's functionality. Moreover, the paper does not evaluate the quality or coverage of the generated tests, since we performed closed-box testing and do not have access to the web application source code. Nevertheless, we are developing a tool to identify UI elements touched by the test set as a measure of quality.

ACKNOWLEDGMENTS

The authors would like to thank the "Centro de Excelência em Inteligência Artificial – CEIA/UFG" and the Brazilian funding agencies that helped carry out this work: CAPES (Grant nº 001), and FAPESP (Grant nº 2019/23160-0 and 2023/00001-9).

7 CONCLUSION

The results presented in this study highlight the strong potential of AI agents in generating automated end-to-end test cases. The commercial model (Claude Sonnet 4) reached an overall success rate of up to 70%, while the open-source model (DeepSeek V3) achieved an overall performance of 34.3% on generating successful test cases. These findings suggest that, while open-source models may offer faster and more stable generation at lower cost, commercial models can deliver higher-quality outputs when reliability is a key priority. Therefore, the choice between them should consider not only performance metrics, but also cost constraints and the acceptable trade-off between generation speed and output accuracy.

From test cases with errors and failures, we also observe very few similarities with models. DeepSeek V3 created more test cases that ran with failures and errors, with texttttNoSuchElementException representing 50% of the errors. Claude Sonnet 4 on its turn concentrated 56.3% of test cases with errors on `TimeoutException` category. Although we did not perform an auto-fix step in this experiment, understanding specific error types is crucial for creating specialized fix agents that aim to convert non-successful test cases into successful ones.

As future work, we intend to extend the experiment to other sets of websites, also including ones for which we have the source code available, making it possible to compute code coverage after running end-to-end test cases. We intend to explore other possible Generic AI agents for end-to-end test case generation, such as OpenManus, Manus, and Genspark, so that we can compare their effectiveness considering different agents.

LLM models' evolution is impressive. Currently, OpenRouter offers 56 free models and another 420 paid models[13]. We also intend to extend the experiment by considering other LLM models and combining them in an incremental test generation strategy to support the generation of high-quality end-to-end test cases. To reach this objective, we need to develop a quality assessment for black-box generated test cases and also improve prompt context, providing additional information about website functionality and testing objectives.

[12] https://openrouter.ai/docs/api-reference/parameters#seed

[13] https://openrouter.ai/models

REFERENCES

[1] Faten Imad Ali, Tariq Emad Ali, and Ziad Tarik Al-Dahan. 2023. Private Backend Server Software-Based Telehealthcare Tracking and Monitoring System. *Int. J. Online Biomed. Eng.* 19, 1 (2023), 119–134.

[2] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing* (1 ed.). Cambridge University Press, Cambridge, Reino Unido.

[3] Sebastian Balsam and Deepti Mishra. 2025. Web application testing – Challenges and opportunities. *Journal of Systems and Software* 219 (2025), 112186. https://doi.org/10.1016/j.jss.2024.112186

[4] Eric J. Braude and Michael E. Bernstein. 2010. *Software engineering: modern approaches* (2 ed.). John Wiley & Sons, Hoboken, NJ. bibtex*[owner=auri;timestamp=2010.08.15].

[5] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2024. A Survey on Evaluation of Large Language Models. *ACM Trans. Intell. Syst. Technol.* 15, 3 (March 2024), 45. https://doi.org/10.1145/3641289 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[6] Pratiksha D Dutonde, Shivani S Mamidwar, Monali Sunil Korvate, Sumangala Bafna, and Dhiraj D Shirbhate. 2022. Website development technologies: A review. *Int. J. Res. Appl. Sci. Eng. Technol* 10, 1 (2022), 359–366.

[7] Khalid El Haji, Carolin Brandt, and Andy Zaidman. 2024. Using GitHub Copilot for Test Generation in Python: An Empirical Study. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24)*. Association for Computing Machinery, New York, NY, USA, 45–55. https://doi.org/10.1145/3644032.3644443 event-place: Lisbon, Portugal.

[8] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. arXiv:2310.03533 [cs.SE] https://arxiv.org/abs/2310.03533

[9] Chen Gao, Xiaochong Lan, Nian Li, Yuan Yuan, Jingtao Ding, Zhilun Zhou, Fengli Xu, and Yong Li. 2024. Large language models empowered agent-based modeling and simulation: a survey and perspectives. *Humanities and Social Sciences Communications* 11, 1 (Sept. 2024), 1259. https://doi.org/10.1057/s41599-024-03611-3

[10] Jia He, Reshmi Ghosh, Kabir Walia, Jieqiu Chen, Tushar Dhadiwal, April Hazel, and Chandra Inguva. 2024. Frontiers of Large Language Model-Based Agentic Systems - Construction, Efficacy and Safety. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24)*. Association for Computing Machinery, New York, NY, USA, 5526–5529. https://doi.org/10.1145/3627673.3679105 event-place: Boise, ID, USA.

[11] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5 (May 2025), 30. https://doi.org/10.1145/3712003 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[12] Chris Kerslake and Ouldooz Baghban Karimi. 2021. Project-based Learning of Web Systems Architecture. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 656. https://doi.org/10.1145/3456565.3460067

[13] Iva Kertusha, Gebremariem Assress, Onur Duman, and Andrea Arcuri. 2025. A Survey on Web Testing: On the Rise of AI and Applications in Industry. https://arxiv.org/abs/2503.05378

[14] Michael Konstantinou, Renzo Degiovanni, Jie M. Zhang, Mark Harman, and Mike Papadakis. 2025. YATE: The Role of Test Repair in LLM-Based Unit Test Generation. https://arxiv.org/abs/2507.18316

[15] Maurizio Leotta, Hafiz Zeeshan Yousaf, Filippo Ricca, and Boni Garcia. 2024. 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 (EASE '24)*. Association for Computing Machinery, New York, NY, USA, 339–344. https://doi.org/10.1145/3661167.3661192 event-place: Salerno, Italy.

[16] Mengyuan Li, Lei Jia, Xiangzhen Chen, Yongxin Li, Dan Zhao, Lina Zhang, Tongqian Zhao, and Jun Xu. 2024. Web system-assisted ratiometric fluorescent probe embedded with machine learning for intelligent detection of pefloxacin. *Sensors and Actuators B: Chemical* 407 (2024), 135491.

[17] Yihao Li, Pan Liu, Haiyang Wang, Jie Chu, and W. Eric Wong. 2025. Evaluating large language models for software testing. *Computer Standards & Interfaces* 93 (2025), 103942. https://doi.org/10.1016/j.csi.2024.103942

[18] Yikang Lu, Alberto Aleta, Chunpeng Du, Lei Shi, and Yamir Moreno. 2024. LLMs and generative agent-based models for complex systems research. *Physics of Life Reviews* 51 (2024), 283–293. https://doi.org/10.1016/j.plrev.2024.10.013

[19] Jose-Manuel Martinez-Caro, Antonio-Jose Aledo-Hernandez, Antonio Guillen-Perez, Ramon Sanchez-Iborra, and Maria-Dolores Cano. 2018. A Comparative Study of Web Content Management Systems. *Information* 9, 2 (2018), 15 pages. https://doi.org/10.3390/info9020027

[20] George Murazvu, Simon Parkinson, Saad Khan, Na Liu, and Gary Allen. 2024. A survey on factors preventing the adoption of automated software testing: A principal component analysis approach. *Software* 3, 1 (2024), 1–27.

[21] Michel Nass, Emil Alégroth, and Robert Feldt. 2024. Improving Web Element Localization by Using a Large Language Model. *Software Testing, Verification and Reliability* 34, 7 (2024), e1893. https://doi.org/10.1002/stvr.1893

[22] Dario Olianas, Diego Clerissi, Maurizio Leotta, and Filippo Ricca. 2025. TESTQUEST: A Web Gamification Tool to Improve Locators and Page Objects Quality. arXiv:2505.24756 [cs.SE] https://arxiv.org/abs/2505.24756

[23] Ipek Ozkaya. 2023. Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications. *IEEE Software* 40, 3 (2023), 4–8. https://doi.org/10.1109/MS.2023.3248401

[24] Ruwei Pan and Hongyu Zhang. 2025. Modularization is Better: Effective Code Generation with Modular Prompting. https://arxiv.org/abs/2503.12483

[25] Elis Pelivani and Betim Cico. 2021. A comparative study of automation testing tools for web applications. In *2021 10th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, Piscataway, NJ, 1–6.

[26] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. 2024. An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project. In *Proceedings of the 1st International Workshop on Large Language Models for Code (LLM4Code '24)*. Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/3643795.3648379 event-place: Lisbon, Portugal.

[27] Filippo Ricca, Maurizio Leotta, and Andrea Stocco. 2019. Chapter Three - Three Open Problems in the Context of E2E Web Testing and a Vision: NEONATE. In *Advances in Computers*, Atif M. Memon (Ed.). Vol. 113. Elsevier, London, United Kingdom, 89–133. https://doi.org/10.1016/bs.adcom.2018.10.005

[28] Sujoy Roychowdhury, Giriprasad Sridhara, A. K. Raghavan, Joy Bose, Sourav Mazumdar, Hamender Singh, Srinivasan Bajji Sugumaran, and Ricardo Britto. 2025. Static Program Analysis Guided LLM Based Unit Test Generation. https://arxiv.org/abs/2503.05394

[29] Daniel Russo, Sebastian Baltes, Niels van Berkel, Paris Avgeriou, Fabio Calefato, Beatriz Cabrero-Daniel, Gemma Catolino, Jürgen Cito, Neil Ernst, Thomas Fritz, Hideaki Hata, Reid Holmes, Maliheh Izadi, Foutse Khomh, Mikkel Baun Kjærgaard, Grischa Liebel, Alberto Lluch Lafuente, Stefano Lambiase, Walid Maalej, Gail Murphy, Nils Brede Moe, Gabrielle O'Brien, Elda Paja, Mauro Pezzè, John Stouby Persson, Rafael Prikladnicki, Paul Ralph, Martin Robillard, Thiago Rocha Silva, Klaas-Jan Stol, Margaret-Anne Storey, Viktoria Stray, Paolo Tell, Christoph Treude, and Bogdan Vasilescu. 2024. Generative AI in Software Engineering Must Be Human-Centered: The Copenhagen Manifesto. *Journal of Systems and Software* 216 (2024), 112115. https://doi.org/10.1016/j.jss.2024.112115

[30] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. https://doi.org/10.1145/3639476.3639764 event-place: Lisbon, Portugal.

[31] Douglas C. Schmidt. 2025. Software Testing in the Generative AI Era: A Practitioner's Playbook. *Computer* 58, 7 (2025), 147–152. https://doi.org/10.1109/MC.2025.3562940

[32] Viktoria Stray, Geir Kjetil Hanssen, Astri Barbala, Darja Šmite, and Klaas-Jan Stol. 2025. What is Generative AI good for? Introduction to the special issue on Generative AI in software engineering. *Information and Software Technology* 187 (Nov. 2025), 107857. https://doi.org/10.1016/j.infsof.2025.107857

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, \Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17, Vol. 1)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010. event-place: Long Beach, California, USA.

[34] Zhiyuan Wan, Yun Zhang, Xin Xia, Yi Jiang, and David Lo. 2023. Software architecture in practice: Challenges and opportunities. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, 1457–1469.

[35] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (Feb. 2024), 911–936. https://doi.org/10.1109/TSE.2024.3368208 Publisher: IEEE Press.

[36] Siyi Wang, Sinan Wang, Yujia Fan, Xiaolei Li, and Yepang Liu. 2024. Leveraging Large Vision-Language Model for Better Automatic Web GUI Testing. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 125–137. https://doi.org/10.1109/ICSME58944.2024.00022

[37] Yuqing Wang, Mika V Mäntylä, Zihao Liu, Jouni Markkula, and Päivi Raulamo-jurvanen. 2022. Improving test automation maturity: A multivocal literature review. *Software Testing, Verification and Reliability* 32, 3 (2022), e1804.

[38] James A. Whittaker. 2009. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, Upper Saddle River, NJ.

[39] Muhammad Nouman Zafar, Wasif Afzal, and Eduard Enoiu. 2022. Evaluating system-level test generation for industrial software: A comparison between manual, combinatorial and model-based testing. In *Proceedings of the 3rd ACM/IEEE international conference on automation of software test*. ACM, New York, NY, 148–159.