Evaluating the Effectiveness of Large Language Models in Automated Unit Test Generation

Tharindu Godage
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
tharindunirmala99@gmail.com

Sivaraj Nimishan
University of Peradeniya
Peradeniya, Sri Lanka
nimishan@eng.pdn.ac.lk

Shanmuganathan Vasanthapriyan
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
priyan@appsc.sab.ac.lk

Vigneshwaran Palanisamy
Charles Darwin University
Casuarina, Australia
p.vickey22@gmail.com

Charles Joseph
University of Southern Queensland
Springfield, Australia
charles.joseph@unisq.edu.au

Selvarajah Thuseethan
Charles Darwin University
Casuarina, Australia
thuseethan.selvarajah@cdu.edu.au

2025 5th International Conference on Advanced Research in Computing (ICARC) | 979-8-3315-3098-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICARC64760.2025.10962997

***Abstract*—The increasing use of Artificial Intelligence (AI) in software development underscores the need to select suitable Large Language Models (LLMs) for automating software unit test generation. No prior work has been conducted to evaluate the performance of LLM in this domain. To address this gap, this study evaluates the effectiveness of four prominent LLMs—GPT-4, Claude 3.5, Command-R-08-2024 and Llama 3.1—in generating unit test cases. This study particularly aims to evaluate the performance of these models in real-world testing scenarios. Hence, 106 test cases from 23 test suites based on interviews with software experts and QA engineers are used to ensure relevance and comprehensiveness. These test cases are analyzed using JavaScript Engines Specification Tester (JEST) for code coverage and Stryker for mutation testing while adopting both quantitative and qualitative analysis. The findings reveal that Claude 3.5 consistently outperforms the other models against test success rate, statement coverage, and mutation score with the achieved accuracy of 93.33%, 98.01%, and 89.23% respectively. The results also provide insights into the capabilities of LLMs for automated unit test generation and their integration into the continuous software integration pipeline. Further, the findings authenticated the importance of systematically comparing LLMs for test case generation.**

***Keywords*—large language models, software unit testing, test automation, mutation testing, javascript**

I. Introduction

In its early stages, natural language processing (NLP) techniques were extensively applied across various domains, including emotion recognition [1]. In recent years, advancements in Artificial Intelligence (AI) have resulted in the development of Large Language Models (LLMs), which have brought transformative innovations to the field of software engineering, with a particular emphasis on automated software testing. Presently, the most popular and widely used LLMs are OpenAI's GPT, Google's Bard [2] and Meta's Llama AI. The growing dependence on software applications in daily life underscores the vital importance of software testing as a fundamental element of the Software Development Lifecycle (SDLC) [3]. As a result, ensuring that software meets essential performance and security requirements is crucial to maintaining user trust and satisfaction. Unit tests can be classified as a specific type of function or method that will act independently [4]. Among the tests considered for software testing, unit tests can be considered an important fundamental type of test [5]. However, manually generating unit tests that also meet code-coverage requirements and execute in a short period is a hassle. Javascript is one of the most popular languages that is used in full-stack development [6], [7]. It is widely utilized for both client-side and server-side web development.

This study explores four major LLMs, such as GPT-4o, Claude 3.5, Command-r-plus-08-2024 and Llama 3.1, developed by OpenAI, Anthropic, Cohere and Meta, respectively. JavaScript Engines Specification Tester (**JEST**), an open-source unit-testing framework for JavaScript and TypeScript testing, was utilized to measure code coverage statistics [7], [8]. In addition, Stryker was used to perform mutation test coverage, commonly referred to as mutation analysis which is defined as a quality-defining metric for test suites. This fault-based testing technique assesses the effectiveness of a test suite by introducing small modifications, known as *mutations*, to the source code of the original program. These modified versions, referred to as *mutants*, are produced through the application of mutation operators. Mutants are generated by substituting specific syntactic elements within the code with alternative constructs, thereby introducing potential faults intentionally. [9]. Currently, most approaches for mutation testing focus on modifying a set of mutation operators, which in turn generate different sets of tests [10].

While numerous LLMs are available for unit test generation, the effectiveness and efficiency of generated test cases vary from one to another. This variability highlights the challenge of determining which LLM is most effective in generating efficient unit tests that achieve shorter execution times while maximizing both code coverage and mutation test coverage in JavaScript. Inefficient and incomplete unit test cases cause issues in future software testing. Therefore. the study conducted a series of experiments to compare selected LLMs, identifying

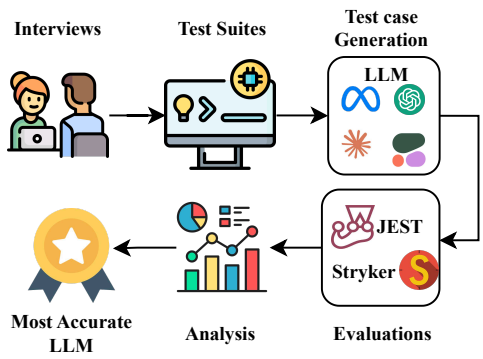

Fig. 1. Illustration of the overall process of the proposed approach.

those that achieved the highest levels of test coverage and mutation test coverage while providing the shortest execution time. The NodeJs *20.15.0* version was utilized for the functions that were tested, a server-side javascript development framework that uses the open-source V8 engine as the default JavaScript engine for code execution [11]. No prior research has been identified that comprehensively analyzes all the aforementioned aspects across multiple LLMs [5]. Therefore, this study aims to ensure which LLM is the most suitable and efficient for software testers to use for unit test generation for JavaScript. Building on this objective, the following research questions (RQs) will be addressed in this study:

- **RQ1:** How do different LLMs compare in terms of test rate success and code coverage for unit test case generation in JavaScript Software development?
- **RQ2:** How do mutation testing scores compare across different LLMs when generating unit tests for JavaScript Software development?
- **RQ3:** How do LLMs perform in generating diverse and comprehensive test cases for JavaScript software development compared to traditional test generation methods?

Addressing these RQs, this study makes the following contributions to the domain of automated software unit test generation:

1) Comparison of existing state-of-the-art LLMs for unit test case generation, specifically using JEST for the JavaScript programming language.
2) An analysis of the challenges and opportunities encountered in the utilization of LLMs for unit test case generation within the Node.js environment.

In this experimental study, 106 test cases are generated from 23 initial test suites, developed through interviews with software industry experts and quality assurance engineers in Sri Lanka. After conducting a series of interviews with industry experts, the key industries were identified. Based on their responses, the interviewees were systematically questioned about the test suites and associated test cases. These interviewees offered valuable insights into common challenges and critical aspects of software testing, thereby ensuring that the test suites were both relevant and comprehensive. The generated test cases were analyzed using JEST and Stryker. The findings of this study can be integrated into a continuous integration/continuous deployment (CI/CD) pipeline, enabling the generation of unit test cases in response to the latest code changes. Fig.1 illustrates the overall process conducted in this study.

The rest of this paper is structured as follows. A brief survey of the related works on LLM-based testing is presented in Section II. Section III introduces the methodology. Section IV presents the experimental results and discussions. Section V provides the conclusion and future works.

II. Related Works

Numerous works have been carried out for automated testing using LLM approaches. Notably, an LLM-based tool has been proposed for automated unit test generation for JavaScript-oriented implementation. However, it achieved a median statement coverage of 70.2% and branch coverage of 52.8% [12]. Similarly, Davis et al. [8] developed NaNofuzz, an interactive tool for identifying NaN-related bugs. It tested with 28 software engineers in a trial experiment, NaNofuzz users identified bugs 30% more accurately, were 20% more confident, and completed tasks 30% faster than those using the Jest testing tool. Meantime, compared to traditional automated tools Guilherme et al. [5] proposed Java unit tests generated by an OpenAI LLM to enhance the quality compared to traditional automated tools using metrics like code coverage and mutation score. The study used 33 programs as a baseline, generating test sets automatically and analyzing their performance. By excluding selected programs from their analysis, they obtained mutation scores ranging from 77 to 90. However, with regard to security concerns on automated testing such approaches are lacking due to its complexity and evolving threat landscape. In order to address such drawbacks an advanced LLM-based tool was introduced to investigate the efficacy in detecting software vulnerabilities [13]. The results have shown that GPT-4o, with a Step-by-Step prompt, achieved the highest F1 score of 0.90, while Claude-3.5 showed an F1 score of 0.89. In contrast, GPT-3.5 Turbo had lower performance, with an F1 score of 0.67.

Most recently, ChatUniTest [14], LLM4Fin [15], and Test-Driven Development (TDD) [16] offered distinct approaches to leveraging LLMs for automates testing, each with unique strengths and limitations. ChatUniTest's [14] adaptive focal context mechanism and generation-validation-repair approach deliver superior line coverage compared to traditional tools like TestSpark [17] and EvoSuite [18], but its reliance on LLMs necessitates additional validation and repair steps to ensure accuracy. LLM4Fin [15], tailored for stock-trading software, achieves impressive business scenario coverage (up to 98.18%) and faster test generation, outperforming both advanced LLMs and human experts, though its domain-specific design may limit applicability to other contexts. Meanwhile, integrating TDD [16] into AI-assisted code generation improves code quality and ensures requirements are met but demands upfront effort in creating suitable tests, potentially offsetting the

TABLE I
OVERVIEW OF LLM MODELS USED IN THIS STUDY.

Model Name	Developer	Release Year	Architecture	Parameters	Primary Use Cases
GPT-4o	OpenAI	May 2024	Transformer-based	1.8 trillion	Natural language generation, coding, Q&A
Claude 3.5	Anthropic	Oct 2024	Transformer-based	175 billion	Safe conversational AI, summarization
Llama 3.1	Meta	Jul 2024	Optimized Transformer	405 billion	Research, conversational AI, language modeling
Command-r-plus-08-2024	Cohere	Aug 2024	Optimized Transformer	104 billion	Code analysis, task automation, software testing

automation benefits. Together, these approaches demonstrate the potential and challenges of applying LLMs in automated testing, balancing efficiency, adaptability, and domain specificity.

Although such approaches contributed to the enhancement, several key issues remain unaddressed. Specifically, these experiments lack a comprehensive way that systematically compares multiple LLMs when it comes to utilization for unit test script generation. Further, studies have focused on their effectiveness in generating unit tests that meet code coverage requirements and test mutation scores. Addressing these gaps, the proposed approach highlights the need to develop and refine LLM-based methods that can effectively be utilized for JavaScript-based software testing. For generating high-quality unit tests in real-world Software projects, ensuring both high test accuracy and strong generalization.

III. METHODOLOGY

By adopting a quantitative and qualitative analysis approach, this study aims to generate new insights into the performance, accuracy, and reliability of LLMs in automated test case generation. The intricacies of the study on automated test case generation necessitate a mixed methods exploration, employing techniques such as model output evaluation, and error rate analysis on generated tests. The rationale behind selecting ChatGPT, Claude, LLaMA, and Command LLMs for this study is grounded in their relevance to natural language generation tasks and their increasing adoption in software development contexts.

A. Data/Scope Considered

For the experiment, a set of test case scenarios was outlined for the LLMs to generate unit test cases. The primary focus was on the most commonly used web applications in business, including e-commerce, banking/financial services, hospitality, and media sectors. Interview results were subsequently gathered from these domains.

B. Large Language Models

This study evaluates the capabilities of four distinct LLMs, namely GPT-4o, Claude 3.5, Llama 3.1, and Command-r-plus-08-2024 [13], in the generation of software units. Table I provides the basic details of these LLMs. In particular, Llama 3.1, the first open model with 405 billion parameters, has shown competitive performance among other AI models in multiple domains [19]. Claude 3.5 Sonnet is a highly advanced model that rivals GPT-4o by Open AI with improved cost efficiency and performance speed compared to its predecessor. Based on the model description provided by Anthropic, Claude 3.5 Sonnet has been trained with data up to April 2024 and has surpassed GPT-4o and Gemini 1.5 Pro in various types of domains [20]. Additionally, Command-r-plus-08-2024 was employed along with other selected models, specifically referred to for conversational interaction and long-context tasks.

C. Experimental Setup

The block diagram depicted in Fig. 2 illustrates the flow of the testing process. Phase 01 - Generating test cases using LLMs from test scenarios; Phase 02 - Storing the test files in the test directory and source files in the source directory; Phase 03 - Evaluating the test success rate and test coverage rate with JEST, and mutation testing with Stryker. To ensure a comprehensive evaluation of the LLMs, as shown in the figure, 106 test scenarios are employed in the experiment. Designed to cover a broad spectrum of potential use cases, these scenarios establish a strong foundation for evaluating the effectiveness of LLMs in generating accurate unit test cases. These test scenarios are based on commonly used instances, with a primary emphasis on back-end test coverage, test success rate and mutation score.

The experiment setup was configured with the i7-1165G7 processor which consists of 4 crores that can run two threads per core, running at 2.80GHz with a maximum clock speed of 2803 MHz and 20 GB RAM. Testing was conducted in a Node.js version 20.15.0 environment, with Stryker-mutator version 8.6.0 and jest version 29.7.0.

D. Generation of Source Code

The primary objective of this study is to evaluate how effectively LLMs can generate unit tests. Specifically, the focus is on evaluating their potential to outperform the manual unit test generation by consistently delivering superior results. In order to achieve this, JavaScript class modules are selected as the source code structure. Using GPT-4o as the LLM, the source code is generated and stored in the src directory within each LLM folder. Each source code file is saved in *.js* format, following a standardized naming convention. The files are further categorized into subdirectories based on the specific functionality they are designed to address. For this experiment, four distinct LLM directories were created to separately track test coverage and mutation score statistics for each LLM, along with individual scores for every test suite.

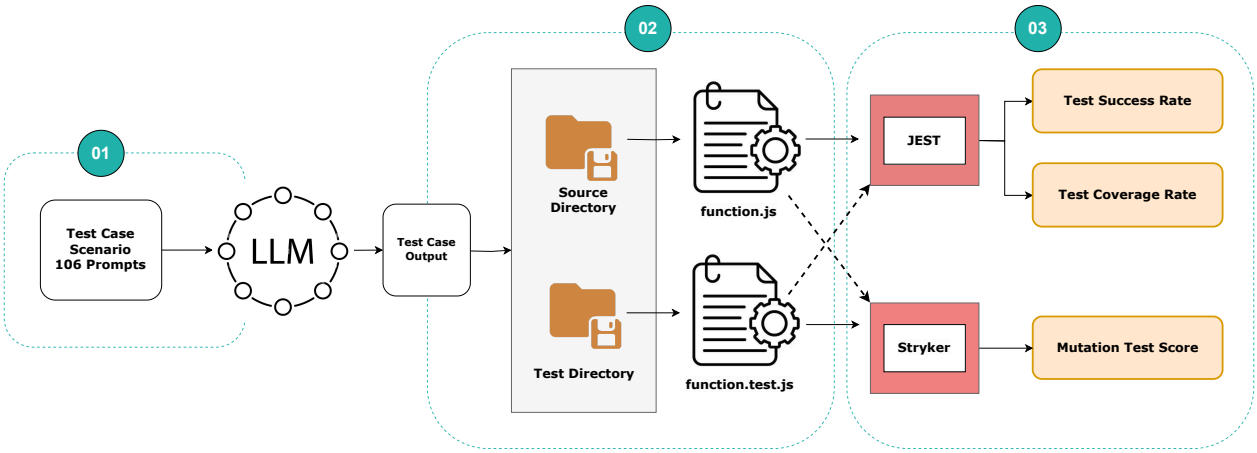

Fig. 2. The block diagram of the testing process: Phase 1 - Generate test cases using LLMs; Phase 2 - Store test files in directories; Phase 3 - Evaluate test success, coverage, and mutation testing.

E. Generating Test Cases

This study analyzed widely used test scenarios from the industry to generate test cases using the outlined process. The selection of test cases in this study is informed by the analysis of responses obtained through a structured interview conducted with local software testing professionals. Each set of test cases generated by the respective LLM is organized into dedicated files with a standardized nomenclature to ensure traceability and support comprehensive coverage verification. The prompt provided in Verbatim III-E is utilized for generating the test cases. Two main inputs are considered for the prompt: (1) the functional code enables the LLM to understand the logic and (2) the test scenario description allows the LLM to generate the unit tests.

Verbatim 1 Prompt used for test case generation

```
<Module/Function Code>,

Based on the above module, create the test
cases below in Jest. Ensure they satisfy
all functional and mutation testing
requirements:

- <Test Scenario 1 Description>
- <Test Scenario 2 Description>
- ..............................
- <Test Scenario n Description>
```

The script shown in Verbatim 2 is used to generate and record the test success rate and test coverage percentage. In the script, the command `test` effectively executes the command `jest --verbose`. The `--verbose` flag is included to ensure that during the execution of the test suite, both successfully executed tests and those that failed to execute are displayed in the output.

Verbatim 2 NPM script configuration for test cases

```
"scripts": {
"test": "jest --verbose",
"coverage": "jest --coverage"
}
```

IV. Results and Discussion

This section presents the results obtained from the analysis. The evaluation metrics, such as test coverage, test success rate, and mutation test score are used to evaluate the performance of the proposed LLM-based approach. These metrics are fine-tuned based on the generated JSON scripts and supplemented by qualitative feedback from local software testing professionals.

A. Test Success Rate

The test success rate serves as the primary metric for evaluating the performance of the proposed LLM approach in generating test cases. This is calculated as the percentage of unit test cases successfully executed relative to the total number of test cases in the corresponding test suite, as defined in *(No of successful test cases / No of total test cases)* $\times$ *100*.

TABLE II
Success rates recorded for the LLMs used in this study.

LLM	Test Success Rate (%)
GPT-4o	88.57
Claude 3.5	**93.33**
Command-r-plus-08-2024	71.43
Llama 3.1	69.52

The success rate of the test cases generated by the LLM models are recorded, as shown in Table II. GPT-4o achieved a success rate of 88.57%, while Claude 3.5 outperformed all

others with a success rate of 93.33%. In contrast, Command-r-plus-08-2024 and Llama 3.1 showed lower success rates of 71.43% and 69.52%, respectively. These results highlight the varying performance of different LLMs in executing the test cases, with Claude 3.5 achieving the highest success rate, and successfully passing 98 out of 106 unit test scenarios. The higher accuracy of Claude 3.5 can be attributed to its advanced training on a diverse range of data and its improved fine-tuning capabilities. Additionally, its more robust error-handling mechanisms likely contributed to its superior performance compared to the other models.

TABLE III
COMPARISON OF TEST COVERAGE PERCENTAGE ACROSS LLMS. IN THIS TABLE, S: STATEMENTS, B: BRANCH, F: FUNCTIONS AND L: LINES

LLMs	Metrics (%)			
	S	B	F	L
GPT-4o	87.11	76.53	95.33	88.22
Claude 3.5	**98.01**	**95.39**	**99.22**	**98.30**
Command-r-plus-08-2024	76.76	67.76	83.65	78.64
Llama 3.1	85.81	81.88	89.83	87.27

B. Test Coverage Percentage

Test coverage percentage measures the extent of codebase execution during software unit testing and is calculated using *(Number of lines executed / Total number of lines)* $\times$ *100*. The test coverage percentages are recorded for four distinct aspects of the codebase: statements, branches, functions and lines. These aspects offer deeper insights into the test cases and help identify potential gaps within the coverage. The script command `coverage` in Verbatim 2 effectively executes the test command `jest --coverage`. This command generates the outputs, providing the test coverage percentages for all four aspects, as shown in Table III.

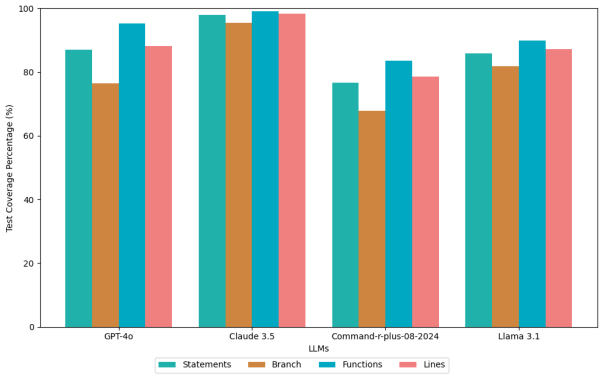

Fig. 3. Test coverage percentages obtained for the LLMs used in this study.

As presented in Table III and visually illustrated in Fig. 3, Claude 3.5 achieved a statement coverage percentage of 98.01%, which demonstrates that nearly all executable statements within the code are tested, thereby surpassing the performance of GPT-4o. Notably, Claude 3.5, leading the performance chart in all four aspects, achieved a statement coverage percentage of 98.01%, a branch coverage percentage of 95.39%, a function coverage percentage of 99.22%, and a line coverage percentage of 98.30%. This highlights its ability to generate unit test cases with higher coverage levels and hence the most suitable and reliable LLM.

C. Test Mutation Score

In the test mutation score metric, four aspects are considered, such as total mutation score, mutants killed, mutants survived and no coverage. The command `npx stryker run` in the Stryker tool effectively executes the mutation testing process. The visual illustration comparing the total mutation score and mutation score covered is presented in Fig. 4. Further, Table IV presents the results obtained for the test suites and test cases, including the total mutation score. Consistent with other metrics, Claude 3.5 attained the highest total mutation score, recording 89.23%, thereby outperforming the other LLMs.

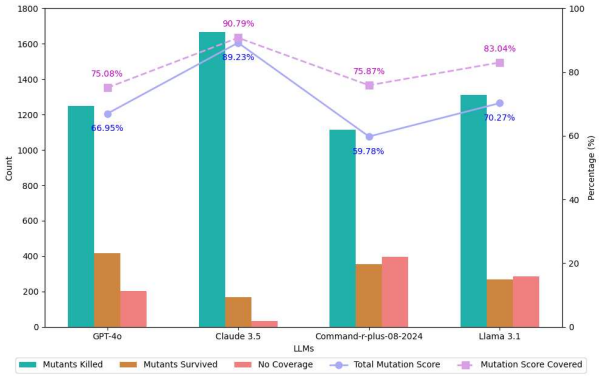

Fig. 4. The test mutation scores obtained for the LLMs used in this study.

In the test mutation score metric, four aspects are considered, such as total mutation score, mutants killed, mutants survived and no coverage. The command `npx stryker run` in the Stryker tool effectively executes the mutation testing process. Table IV presents the results obtained for the test suites and test cases, including the total mutation score and mutation score covered. The total mutation score measures the percentage of killed mutants, while the mutation score covered evaluates the test suite's effectiveness by considering only the covered mutants and calculating the percentage detected. Both of these metrics aim to assess the quality and thoroughness of tests in the specific areas they cover.

The visual illustration comparing the mutants killed, mutants survived and no coverage along with total mutation score and mutation score covered is presented in Fig. 4. Consistent with other metrics, Claude 3.5 attained the highest total mutation score of 89.23% and mutation score covered of 90.79%, thereby outperforming all other LLMs compared in this study. Further, Claude 3.5 killed 1666 mutants, which demonstrates its higher level of mutation detection ability compared to other LLMs. It is interesting to note that Claude

TABLE IV
COMPARISON OF MUTATION TESTING METRICS ACROSS LLMS. THE TOTAL MUTATION SCORE AND MUTATION SCORE COVERED ARE PRESENTED AS PERCENTAGES, WHILE THE OTHER THREE METRICS ARE REPORTED AS COUNTS.

LLMs	Metrics				
	Total Mutation Score	Mutation Score Covered	Mutants Killed	Mutants Survived	No Coverage
GPT-4o	66.95	75.08	1250	415	202
Claude 3.5	**89.23**	**90.79**	1666	169	32
Command-r-plus-08-2024	59.78	75.87	1116	355	396
Llama 3.1	70.27	83.04	1312	268	287

3.5 showcased the fewest mutants with no coverage (i.e., 32). In comparison, Command-r-plus-08-2024 had the highest number of uncovered mutants (i.e., 396). Overall, Claude 3.5 demonstrated exceptional performance across all evaluated mutation metrics. GPT-4o and Llama 3.1 demonstrated reasonable performance. However, they exhibited a significant gap compared to Claude 3.5. Command-r-plus-08-2024 exhibited the weakest performance, achieving a low mutation score of 59.78 and recording the highest number of uncovered mutants. This suggests that Command-r-plus-08-2024 exhibits limitations in its underlying architecture and mutation detection mechanisms when compared to other LLMs.

V. CONCLUSION

The application of LLMs for automated software unit test generation remains an unexplored area. To address this gap, in this study, the effectiveness of LLMs in automating unit test generation without human intervention in test case assessments for JavaScript is explored. The performance of four widely used LLMs, such as GPT-4o, Claude 3.5, Command-r-plus-08-2024 and Llama 3.1 evaluated in generating software test cases. The standard evaluation metrics, such as test success rate, test coverage percentage and test mutation score are employed to evaluate and compare the effectiveness of results. The analysis of 106 test case scenarios demonstrates that Claude 3.5 surpasses other LLM models in generating unit test cases with higher accuracy. Claude 3.5 consistently achieved the highest success rate, statement coverage, and mutation score, while GPT-4 and Llama 3.1 delivered satisfactory performance across these metrics. In the future, this work can be extended to other programming languages and explore various optimization techniques to further improve LLM-driven testing in diverse development environments.

REFERENCES

[1] S. Thuseethan, S. Janarthan, S. Rajasegarar, P. Kumari, and J. Yearwood, "Multimodal deep learning framework for sentiment analysis from text-image web data," in *2020 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. IEEE, 2020, pp. 267–274.

[2] M. U. Hadi, Q. Al Tashi, A. Shah, R. Qureshi, A. Muneer, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu *et al.*, "Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects," *Authorea Preprints*, 2024.

[3] S. Pargaonkar, "A comprehensive review of performance testing methodologies and best practices: software quality engineering," *International Journal of Science and Research (IJSR)*, vol. 12, no. 8, pp. 2008–2014, 2023.

[4] H. Cheddadi, S. Motahhir, and A. E. Ghzizal, "Google test/google mock to verify critical embedded software," *arXiv preprint arXiv:2208.05317*, 2022.

[5] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*, 2023, pp. 15–24.

[6] A. Shukla, "Modern javascript frameworks and javascript's future as a full-stack programming language," *Journal of Artificial Intelligence & Cloud Computing. SRC/JAICC-156. DOI: doi. org/10.47363/JAICC/2023 (2)*, vol. 144, pp. 2–5, 2023.

[7] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "Jest: N+ 1-version differential testing of both javascript engines and specification," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 13–24.

[8] M. C. Davis, S. Choi, S. Estep, B. A. Myers, and J. Sunshine, "Nanofuzz: A usable tool for automatic test generation," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1114–1126.

[9] M. Roos, "Faster mutation testing through simultaneous mutation testing," Master's thesis, University of Twente, 2024.

[10] F. Tip, J. Bell, and M. Schäfer, "Llmorpheus: Mutation testing using large language models," *arXiv preprint arXiv:2404.09952*, 2024.

[11] D. P. Mishra, K. K. Rout, and S. R. Salkuti, "Modern tools and current trends in web-development," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 24, no. 2, pp. 978–985, 2021.

[12] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[13] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques," *Electronics*, vol. 13, no. 13, p. 2657, 2024.

[14] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 572–576.

[15] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, "From llms to llm-based agents for software engineering: A survey of current, challenges and future," *arXiv preprint arXiv:2408.02479*, 2024.

[16] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1583–1594.

[17] A. Sapozhnikov, M. Olsthoorn, A. Panichella, V. Kovalenko, and P. Derakhshanfar, "Testspark: Intellij idea's ultimate test generation companion," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 30–34.

[18] J. Callan and J. Petke, "Multi-objective genetic improvement: A case study with evosuite," in *International Symposium on Search Based Software Engineering*. Springer, 2022, pp. 111–117.

[19] R. Vavekanand and K. Sam, "Llama 3.1: An in-depth analysis of the next-generation large language model," 2024.

[20] C. White, S. Dooley, M. Roberts, A. Pal, B. Feuer, S. Jain, R. Shwartz-Ziv, N. Jain, K. Saifullah, S. Naidu *et al.*, "Livebench: A challenging, contamination-free llm benchmark," *arXiv preprint arXiv:2406.19314*, 2024.