LLMs for Intelligent Software Testing: A Comparative Study

Mohamed Boukhlif
mohamed.boukhlif.rd@gmail.com
LTI Laboratory, National School of Applied Sciences, Chouaïb Doukkali University
El Jadida, Morocco

Nassim Kharmoum
nkharmoum@gmail.com
IPSS Team, Faculty of Sciences, Mohammed V University in Rabat
Rabat, Morocco

Mohamed Hanine
hanine.m@ucd.ac.ma
LTI Laboratory, National School of Applied Sciences, Chouaïb Doukkali University
El Jadida, Morocco

ABSTRACT

The need for effective and timely testing processes has become critical in the constantly changing field of software development. Large Language Models (LLMs) have demonstrated promise in automating test case creation, defect detection, and other software testing tasks through the use of the capabilities of machine/deep learning and natural language processing. This work explores the field of intelligent software testing, with a focus on the use of LLMs in this context. The purpose of this comparative study is to assess the corpus of research in the field in terms of used LLMs, how to interact with them, the use of fine-tuning, and prompt engineering, and explore the different technologies and testing types automated using LLMs. The findings of this study not only contribute to the growing body of knowledge on intelligent software testing but also guide fellow researchers and industry engineers in selecting the most suitable LLM for their specific testing needs.

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Large Language Models, Software Testing, Test Case Generation, Natural Language Processing, Comparative Study

ACM Reference Format:

Mohamed Boukhlif, Nassim Kharmoum, and Mohamed Hanine. 2024. LLMs for Intelligent Software Testing: A Comparative Study. In *The 7th International Conference On Networking, Intelligents Systems and Security (NISS 2024), April 18–19, 2024, MEKNES, AA, Morocco.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3659677.3659749

1 INTRODUCTION

In the field of software engineering, the search for optimal testing methodologies has been a perpetual journey, propelled by the inexorable evolution of technology and the ever-increasing complexity of software systems. As software permeates every facet of modern life, from critical infrastructure to personal devices, the imperative to ensure its reliability, security, and functionality has never been more pronounced [3, 7, 19].

Software testing plays a pivotal role in ensuring the reliability, functionality, and quality of software systems in today's increasingly digital world. As software applications continue to permeate various aspects of daily life, from critical systems in healthcare [5] and finance to consumer-facing mobile apps and web platforms, the need for rigorous testing methodologies becomes ever more apparent [15]. Software bugs, glitches, and vulnerabilities can have far-reaching consequences, leading to system failures, security breaches, financial losses, and even endangering human lives in safety-critical domains [16, 18]. Therefore, the importance of thorough and systematic software testing cannot be overstated. Moreover, effective testing practices instill confidence in software reliability, enhance user satisfaction, and ultimately contribute to the overall success and competitiveness of software products and systems in the marketplace [21].

The application of artificial intelligence (AI) techniques has revolutionized the field of software testing, offering innovative solutions to address the growing complexity and scale of modern software systems [4]. AI-driven approaches, powered by advanced machine learning, deep learning, and graph theory algorithms, have emerged as promising tools to automate and augment various aspects of the testing process [8]. These techniques leverage the capabilities of AI to analyze code, generate test cases, detect anomalies, and predict potential failure points, thereby streamlining testing workflows and improving overall efficiency. By harnessing the vast amounts of data available in software repositories, AI-driven testing methodologies can identify patterns, trends, and correlations that may not be apparent to human testers, enabling more comprehensive test coverage and deeper insights into software behavior [10]. Moreover, AI techniques enable adaptive and self-learning testing systems that can continuously evolve and adapt to changing software environments, improving resilience and responsiveness in the face of evolving requirements and challenges [1].

The integration of LLMs into software engineering represents a paradigm shift in the way code development is approached, offering unprecedented capabilities in automated code generation, code analysis, and defect detection. Utilizing LLMs, enable testers to benefit from their natural language understanding capabilities to interpret and generate test scenarios from textual descriptions, code comments, documentation, and other sources of information [2].

At its core, this paper aims to answer the following questions :

RQ1 : What are the most used LLMs in the field of software testing
RQ2 : What input do we leverage to the LLM in order to get what output?

NISS 2024, April 18–19, 2024, MEKNES, AA, Morocco

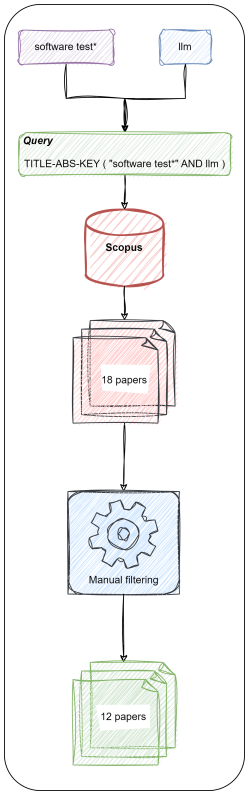
ACM ISBN 979-8-4007-0929-6/24/04
https://doi.org/10.1145/3659677.3659749

RQ3 : To what extent do scholars use fine-tuning and prompt engineering in their research?
RQ4 : What testing technologies and types are most used in the research field?
RQ5 : Do scholars develop domain-focused or generic approaches?
RQ6 : How open are the solutions provided by the researchers in the field?

The rest of this paper is structured as follows: Section 2 explains the data collection strategy we opted for, then we give a small summary analysis of each studied paper in Section 3, and then Section 4 presents the results and discusses the output of the comparative study, Section 5 presents insights, future work and limitations of this study.

2 DATA COLLECTION

Figure 1: Search strategy



In this section, we detail the process by which we gathered and curated the information necessary for our comparative study. This involved a meticulous approach to sourcing relevant papers from the Scopus database, utilizing the two keywords "software test*" and "llm", these two words are the two pillars of our research corpus. Additionally, we added a manual filtering step where we excluded papers that were not relevant to our research, ensuring the integrity and relevance of the studies included in our analysis. We ended up with 12 papers to be studied and analyzed to answer the presented research questions. Our data collection methodology prioritized comprehensiveness and rigor, aiming to capture a representative sample of research contributions in the field of automated testing methodologies for thorough comparative analysis. Figure 1 summarizes the search strategy we adopted.

We chose Scopus as the database for our research paper due to its comprehensive coverage of scholarly literature across a wide range of disciplines, including computer science, artificial intelligence, and software engineering. Scopus offers a vast repository of peer-reviewed journals, conference proceedings, and technical reports, providing access to high-quality research publications from reputable sources. Moreover, Scopus's robust database ensures the reliability and credibility of the research papers included in our comparative study, enhancing the validity and rigor of our findings.

3 STUDIED PAPERS

To our knowledge, this is the first paper to compare LLMs for intelligent software testing. In this section, we showcase some papers reviewing LLMs' use in software testing or software engineering.

Paper [24] reviews the utilization of LLMs for generating test scripts in mobile applications to address issues related to script capture and reproduction across various devices and platforms. It emphasizes the significance of quality and reliability in mobile apps and advocates for the use of LLMs like ChatGPT for test script generation and migration. The study focuses on scenario-based test generation, cross-platform test migration, and cross-app test migration while acknowledging challenges such as context memory issues and API randomness. It contributes to advancing the understanding of LLMs in mobile app testing and lays the groundwork for future research in leveraging AI technologies to improve app quality and reliability.

Paper [20] investigates LLMs for automated unit test generation, prompted with function signatures, implementations, and usage examples from documentation. When a test fails, the model is prompted with the error message to generate a new test. TestPilot for JavaScript achieves notable coverage on 25 npm packages with 1,684 API functions, surpassing Nessie's state-of-the-art results. Results demonstrate effective test suite generation, with novel tests and consistent performance across different LLMs, indicating effectiveness influenced by LLM size and training rather than model dependency.

Paper [17] explores integrating LLMs, like OpenAI's Codex, into Search-based Software Testing (SBST) methodologies. It introduces CODAMOSA, combining SBST with LLMs to enhance coverage by generating test cases for under-covered functions. Evaluation on 486 benchmarks shows CODAMOSA achieves significantly higher coverage than SBST and LLM-only baselines, with minimal reduction on other benchmarks. Automated test case generation targets all program behaviors.

Paper [9] suggests using LLMs as automated testing assistants to address software testing challenges, reducing the need for specialized expertise and extensive developer effort. It introduces a taxonomy classifying LLM-based testing agents by autonomy level and discusses benefits of higher autonomy. The paper highlights the advantageous aspect of "hallucination" in LLMs for testing and identifies tangible benefits of LLM-driven testing agents alongside potential limitations.

Paper [13] explores leveraging large-language models, like GPT3.5, to enhance penetration testing by providing AI sparring partners alongside human testers. It examines two main use cases: high-level task planning and low-level vulnerability hunting within a virtual machine. For the latter, a closed-feedback loop is implemented between LLM-generated actions and a vulnerable virtual machine via SSH. The paper discusses initial promising results, improvement opportunities, and ethical considerations regarding AI sparring partners in penetration testing.

Three papers introduce innovative workflows leveraging autoregressive transformer-based LLMs for automated tasks in software testing. The first paper [22] focuses on variable extraction from scientific software user manuals, demonstrating significant accuracy improvement. The second paper [23] addresses the oracle problem in scientific software testing, proposing LLM-driven variable extraction, showcasing effectiveness in comparison to manual methods. The third paper [25] presents a pilot study on using ChatGPT for automated generation of metamorphic relations in testing autonomous driving systems, emphasizing efficiency, quality, and scalability benefits while highlighting practical implications and future research avenues.

Paper [11] tackles the challenge of identifying subtle compiler bugs through fuzzing, aiming to enhance both the quantity and quality of generated test cases. It introduces a novel code generation approach based on LLMs, employing a filter strategy to ensure test cases are error-free and a seed schedule strategy for efficiency. Tested on the Golang compiler, the method outperforms prior approaches, achieving higher coverage and lower syntax error rates. With an average coverage of 3.38%, surpassing GoFuzz-generated tests, and minimal syntax errors, the paper demonstrates LLM-based code generation's effectiveness in compiler testing, offering significant improvements.

Paper [6] presents a novel automated testing technique merging LLMs with search-based fuzzing. It explores LLMs and search-based methods to automate gem5 software testing, introducing new mutation operators and custom mutators for enhanced test input generation and coverage. Combining LLMs with AFL++ search, the approach is evaluated on parameterized C programs using GPT-3.5-turbo, showing its effectiveness in bug finding and coverage improvement in complex simulation systems like gem5.

Paper [14] presents Libro, a framework aiming to improve automated test generation's ability to reproduce bugs from bug reports, an essential yet often overlooked aspect of software development. Leveraging LLMs, Libro focuses on post-processing to assess test effectiveness and rank them based on validity, compensating for LLMs' inability to directly execute buggy code. Evaluation on the Defects4J benchmark demonstrates Libro's capability in generating failure-reproducing test cases for many bugs, with bug-reproducing tests frequently ranked as the top suggestion. Additionally, evaluation against post-training bug reports shows promising results. Overall, Libro offers the potential to enhance developer efficiency by automating test generation from bug reports, addressing a crucial aspect often neglected in software testing.

In [12], the quality of Java unit tests generated by an OpenAI Large Language Model (LLM) algorithm is evaluated against traditional automated test generation tools. 33 programs commonly used in test generation research were chosen, and 33 unit test sets were generated for each program using the OpenAI API, without human intervention. Metrics such as code line coverage, mutation score, and test execution success rate were collected to assess efficiency and effectiveness. Results indicate that LLM-generated test sets perform similarly to traditional tools across all metrics, highlighting LLMs' potential in automated Java test generation, despite the experiment's simplicity and lack of human analysis.

4 RESULTS & DISCUSSION

In this section, we provide a comprehensive analysis of the data collected, highlighting key trends, patterns, and insights gleaned from our investigation into automated testing methodologies and answering the presented research questions. We begin by presenting the quantitative results obtained from our comparative analysis, detailing the outcomes of various experiments and measurements conducted across the included papers. Through critical analysis and synthesis of the results, we aim to elucidate the strengths, limitations, and implications of different approaches to automated testing, shedding light on emerging trends, best practices, and avenues for future research. Table 1 summarizes the comparative study.

4.1 Used LLMs

Figure 2: Used LLMs in the studied papers

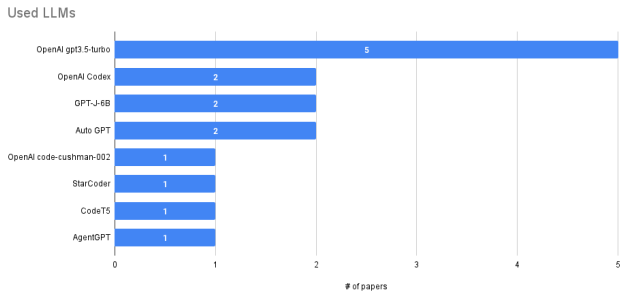


Figure 2 outlines the distribution of various LLMs used in the study.

- OpenAI GPT-3.5-turbo: This model was the most frequently used, with 5 papers. It indicates a preference for leveraging the capabilities of this specific model, possibly due to its known effectiveness in generating text-based content.
- OpenAI Codex: With 2 papers, this model also sees significant usage. Codex is specifically designed for code-related tasks, suggesting that some tasks in the study required specialized code-generation capabilities.
- GPT-J-6B, Auto GPT: Also with 2 papers each, GPT-J-6B is a general-purpose model with 6 billion parameters, like any other GPT model, it can generate human-like text, perform text completion, answer questions, and more. AutoGPT on the other hand, is a concept rather than a specific model. It refers to the automated process of designing, training, and fine-tuning GPT-based models using techniques like neural

Table 1: Comparison between the papers

Papers / Characteristecis	Used LLM								Input					Output					Prompt Engineering		Implementation Programming Language			Target Programming Language						Target Testing Framework			Application Domain							Type of Testing					Open Source
	OpenAI gpt3.5-turbo	OpenAI code-cushman-002	StarCoder	CodeT5	OpenAI Codex	GPT-J-6B	Auto GPT	AgentGPT	Natural language	Source code	Documentation	User Manual	Bug Report	Test scripts	Unit Tests	Test Plan	Test cases	Metamorphic Variables	Fine tuning		TypeScript	Python	Bash	Java	JavaScript	TypeScript	Go	Python	Julia	C Language	Mocha	JUnit	Pynguin	Mobile applications	Web Applications	System Administration	Command-line Applications	Compiler	Autonomous driving systems	Generic	Usability testing	Unit Testing	Metamorphic Testing	Security Testing	System Testing	
LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities	V								V					V					N/A	V	N/A						V			N/A								V					N/A
An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	V	V	V							V	V				V				N/A	V	V				V	V				V								V		V				V
CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models					V					V					V				N/A	V		V						V			V								V					V
Towards Autonomous Testing Agents via Conversational Large Language Models							V		V						V				N/A	V	N/A							V		N/A									V		V				N/A
Getting pwn'd by AI: Penetration Testing with Large Language Models							V	V	V							V	V		N/A	V		V			N/A					N/A		V	V								V		V
Large Language Models: The Next Frontier for Variable Discovery within Metamorphic Testing?						V					V	V						V	N/A	V	N/A				N/A					N/A				V					V					N/A
Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software						V					V	V						V	N/A	V	N/A				N/A					N/A				V					V					N/A
LLM-Based Code Generation Method for Golang Compiler Testing				V						V					V				N/A	V	N/A					V					N/A					V				V				V
SearchGEM5: Towards Reliable Gem5 with Search Based Software Testing and Large Language Models	V								V	V				N/A					N/A	V		V	V						V		N/A					V							V	V
Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction					V								V				V		N/A	V		V		V							V							V		V				V
An initial investigation of ChatGPT unit test generation capability	V									V					V				N/A	V		V		V							V							V		V				V
Automated Metamorphic-Relation Generation with ChatGPT: An Experience Report	V								V									V	N/A	N/A	N/A										N/A						V				V				V

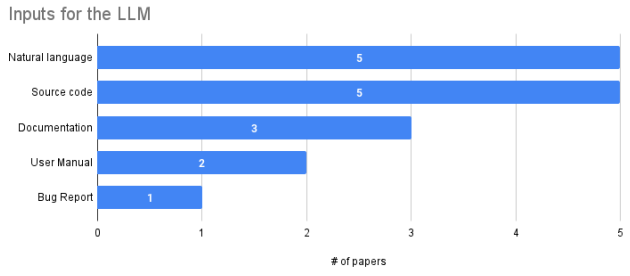
architecture search, which can lead to more efficient and effective models tailored to particular applications.

- OpenAI code-cushman-002, StarCoder, CodeT5, AgentGPT: These models each had 1 paper of usage. While less frequently used, their inclusion suggests an exploration of various LLMs' capabilities and potential differences in performance across different models.

This section answered **RQ1** of our research questions.

4.2 Inputs for the LLM

Figure 3: Inputs for the LLM



This section delineates the types of input utilized for the LLMs in the study, which is part of the answer to **RQ2**. As shown in Figure 3, among the input categories, natural language emerges as the most prevalent, appearing in 5 papers. This suggests a predominant reliance on text-based information, such as descriptions or requirements, for tasks within the study. Additionally, source code is equally represented with 5 papers, implying that the study involved tasks related to code generation, analysis, or manipulation. This alignment with LLMs designed for code-related tasks, such as Codex or GPT-J-6B, underscores the study's focus on software development or programming-related activities. Documentation serves as input in 3 papers, indicating that the study involved tasks requiring extraction or interpretation of information from software documentation, such as APIs or libraries. This highlights the utilization of LLMs for tasks involving understanding and processing technical documentation. Additionally, user manuals are utilized in 2 papers, suggesting a focus on tasks requiring comprehension or interpretation of user-facing instructions or guidelines. Bug reports serve as input in 1 paper, indicating the study's involvement in tasks related to software quality assurance or debugging. In such tasks, LLMs may be employed to analyze or generate responses based on reported issues, underscoring their utility in the context of software testing and debugging. Overall, the inclusion of diverse input types reflects the versatility of LLMs and their applicability across various domains and tasks, ranging from natural language understanding to code generation and software documentation analysis.

Figure 4: Outputs of the LLM

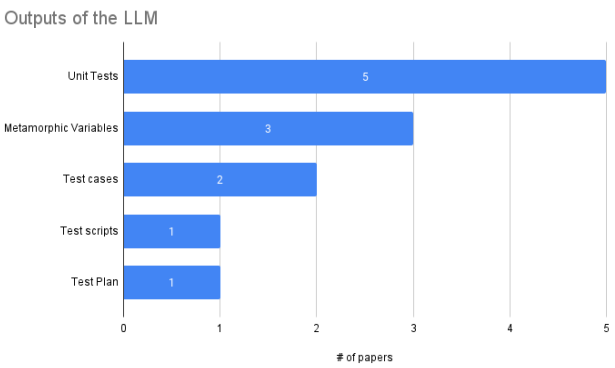
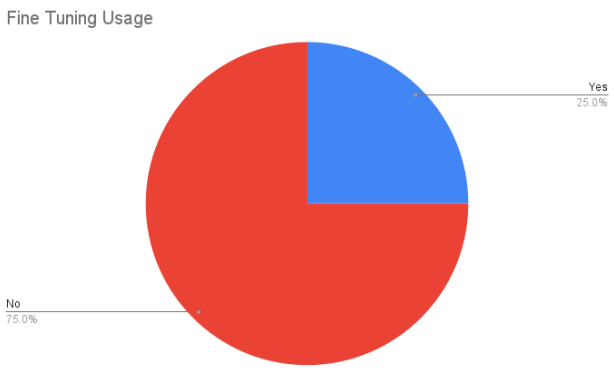


Figure 5: Fine Tuning Usage



4.3 Outputs of the LLM

This section is part of the answer to **RQ2**. As we can see in Figure 4, unit tests emerge as the most common output of the LLM, appearing in 5 papers. This suggests a significant focus on the automated generation of unit tests by the LLMs, reflecting their utility in enhancing software testing processes by automating the creation of test cases for individual units of code. Metamorphic Variables follow with 3 papers, indicating a notable application of LLMs in generating metamorphic relations or variables for metamorphic testing. This suggests a recognition of the importance of metamorphic testing in ensuring software quality and reliability, with LLMs playing a role in automating the generation of test inputs and transformations. Additionally, Test cases are generated in 2 papers, indicating a broader application of LLMs in generating general test cases beyond unit tests. This suggests the versatility of LLMs in automating various aspects of software testing, beyond just unit-level testing. Furthermore, Test scripts and Test Plans each appear in 1 paper, indicating more specific applications of LLMs in generating higher-level testing artifacts such as test scripts or comprehensive test plans. This suggests the potential of LLMs to automate the entire testing process, from test case generation to test execution and evaluation. This diversity in output types reflects the versatility of LLMs in automating various aspects of software testing, ranging from unit test generation to the creation of comprehensive test plans, thereby enhancing efficiency and effectiveness in software testing processes.

4.4 Fine Tuning Usage

In the papers included in the study, fine-tuning was used only in 3 papers, while 9 papers did not utilize it as shown in Figure 5. This indicates a mixed approach to Fine Tuning among the studies, with some opting to fine-tune the LLMs for specific tasks or domains, while others relied on pre-trained models without further tuning. The decision to fine-tune or not depends on factors such as the complexity of the task, the availability of domain-specific data, and the desired level of model customization. The usage of fine-tuning reflects the varying strategies adopted by researchers in leveraging LLMs for automated testing tasks. This section answered a part of **RQ3**.

4.5 Prompt Engineering Usage

Figure 6: Prompt Engineering Usage

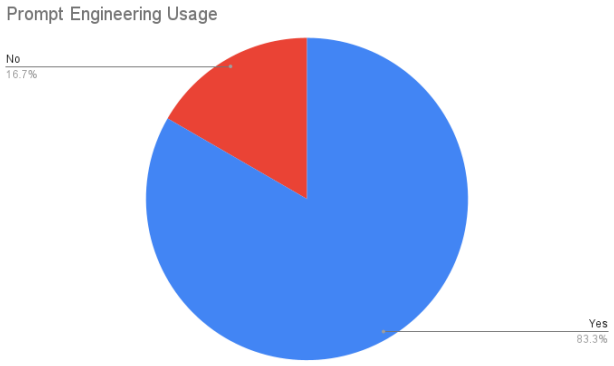
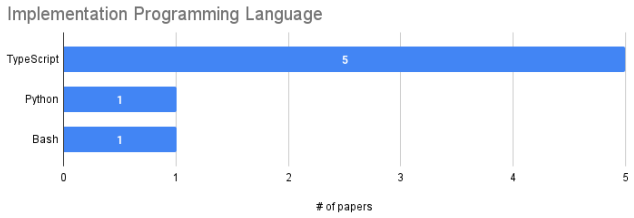


Figure 6 shows that 10 papers of the studied corpus have used prompt engineering, while 2 papers did not employ it. This suggests a prevalent use of prompt engineering as a strategy to guide the LLMs in generating desired outputs for automated testing tasks. Prompt engineering involves crafting specific prompts or instructions to direct the behavior of the LLMs toward producing relevant and high-quality outputs. The widespread usage of prompt engineering indicates its importance in ensuring the effectiveness and relevance of the generated outputs for the intended testing purposes, and its adoption reflects the careful consideration given by researchers to optimize the performance of LLMs in automated testing scenarios. This section answered a part of **RQ3**.

Figure 7: Implementation Programming Language

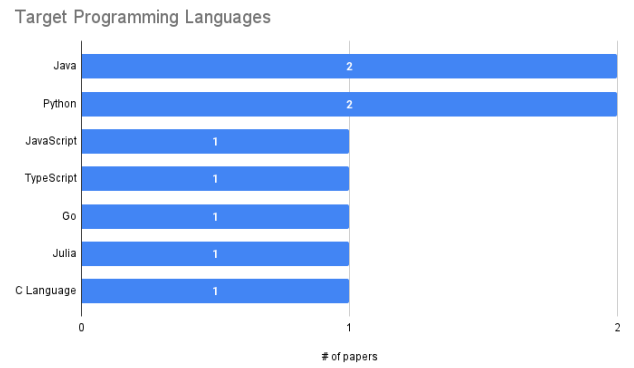


4.6 Implementation Programming Language

Among the papers included in the study, TypeScript was the most commonly used programming language for implementation, appearing in 5 papers as stats Figure 7. This indicates a significant preference for TypeScript among researchers when implementing their automated testing methodologies. Python was used in 1 paper, suggesting a less frequent but still notable choice for implementation. Furthermore, Bash was utilized in 1 paper, representing a more niche usage for specific tasks or workflows. The prevalence of TypeScript usage may reflect its popularity in web development and its suitability for tasks related to automated testing, such as building tools or frameworks. Python's inclusion suggests its versatility and widespread adoption in various domains, including software testing. The utilization of Bash indicates its utility for scripting and automation tasks, particularly in environments where command-line operations are prevalent. The distribution of implementation programming languages reflects the diverse needs and preferences of researchers in developing automated testing solutions. This section answered a part of **RQ4**.

4.7 Target Programming Language

Figure 8: Target Programming Language



The target programming languages varied across the papers included in the study as shown in Figure 8. Java and Python were each targeted in 2 papers, indicating their popularity as languages for software development and testing. JavaScript, TypeScript, Go, Julia, and C Language were each targeted in 1 paper, showcasing a diverse range of languages used for implementing automated testing methodologies. The choice of target programming language may depend on factors such as the nature of the software being tested, the preferences of the researchers, and the specific requirements of the study. For example, Java and Python are widely used in software development and have extensive libraries and frameworks available for testing purposes. JavaScript and TypeScript are commonly used for web development, making them suitable choices for testing web applications. Go is known for its efficiency and simplicity, making it a viable option for testing performance-critical systems. Julia's focus on numerical and scientific computing suggests its usage in testing scientific software. C Language, being a low-level language, is targeted for testing system-level software or performance-critical applications. This distribution of target programming languages reflects the diversity of software systems and the flexibility of automated testing methodologies to accommodate various programming languages and environments. This section answered a part of **RQ4**.

4.8 Target Testing Framework

Figure 9: Target Testing Framework

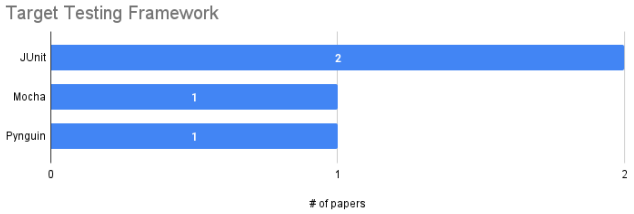


Figure 9 showcases the target testing frameworks across the studied papers. JUnit was the most commonly targeted framework, appearing in 2 papers. This aligns with JUnit's popularity as a testing framework for Java applications, indicating its continued relevance in automated testing research. Mocha and Pynguin were each targeted in 1 paper. Mocha is a popular testing framework for JavaScript, particularly in the context of Node.js applications, while Pynguin is a novel framework specifically designed for automated unit test generation in Python. The choice of target testing framework depends on factors such as the programming language of the software being tested, the preferences of the researchers, and the specific requirements of the study. Those results reflect the choice of researchers in selecting frameworks that best suit their automated testing methodologies and the systems under test. This section answered a part of **RQ4**.

4.9 Application Domains

The application domains targeted by the intelligent testing methodologies in the papers included in the study exhibit a diverse range of areas as stated in Figure 10 :

- Generic: The most commonly targeted domain, with 5 papers, suggests a broad applicability of the automated testing

Figure 10: Application Domains

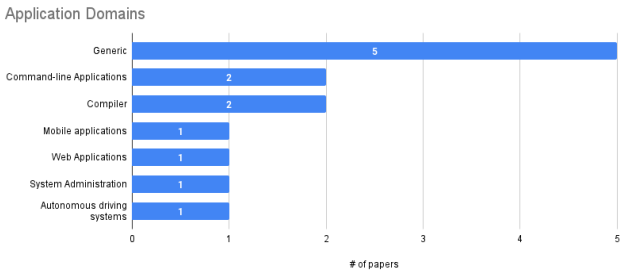
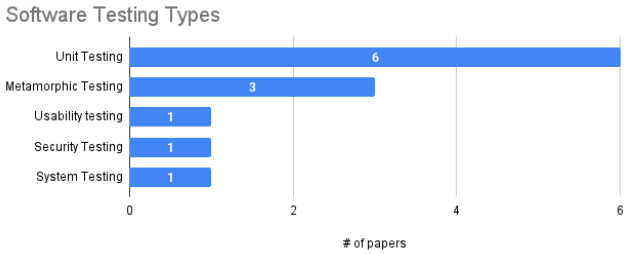


Figure 11: Software Testing Types



methodologies across various software systems and domains. This indicates a generalizability of the approaches proposed in the study.

- Command-line Applications: Targeted in 2 papers, this domain reflects a focus on testing software applications designed to be executed via the command-line interface. Such applications may include utilities, scripts, or tools commonly used in command-line environments.
- Compiler: Also targeted in 2 papers, this domain suggests a specific focus on testing compilers, which are critical components in software development responsible for translating source code into executable machine code.
- Mobile Applications, Web Applications, System Administration, Autonomous Driving Systems: Each targeted in 1 paper, these domains represent specialized areas where automated testing methodologies are applied. Mobile and web applications involve testing software designed for mobile devices and web browsers, respectively. System administration encompasses testing tools or systems used for managing and maintaining computer systems. Autonomous driving systems involve testing software used in autonomous vehicles, highlighting the importance of safety and reliability in critical domains.

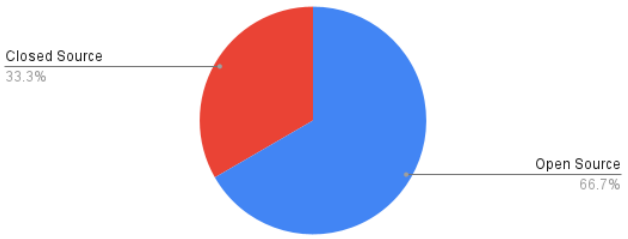
This diversity in application domains reflects the versatility and applicability of intelligent testing methodologies across various software systems and industries, ranging from general-purpose applications to specialized domains with unique testing requirements. This section answered a part of **RQ5**.

4.10 Software Testing Types

Across the papers included in the study, a range of software testing types were targeted, demonstrating a diverse focus on key methodologies as visualized in Figure 11. Unit testing emerged as the most prevalent testing methodology, with 6 papers specifically targeting this approach. Unit testing involves scrutinizing individual units or components of software to verify their functionality in isolation, emphasizing the foundational importance of ensuring each component operates correctly within the larger system. Metamorphic testing, targeted in 3 papers, represents another significant focus. This methodology revolves around testing software based on the consistency of its outputs under various transformations. Its application is particularly valuable for testing complex systems where traditional approaches may fall short, highlighting the importance of robust testing methodologies in ensuring software reliability. Additionally, specialized areas of testing were also addressed, albeit to a lesser extent. Usability testing, security testing, and system testing were each targeted in 1 paper, indicating a more focused approach to evaluating specific aspects of software quality. Usability testing involves assessing a software system's user interface and experience to ensure it is intuitive and user-friendly, while security testing focuses on identifying and mitigating vulnerabilities and risks. System testing encompasses a comprehensive evaluation of the integrated system to verify its compliance with specified requirements and functionality within its intended environment. The variety of software testing types throughout the publications highlights the necessity of a holistic approach to software quality assurance, which includes multiple approaches to address distinct aspects of program security, functionality, dependability, and usability. This section answered a part of **RQ4**.

4.11 Open/Closed Source

Figure 12: Open/Closed Source



As shown in Figure 12, 8 papers opted for open-source implementation of their LLM, while 4 papers chose closed-source implementation. The open-source implementation allows for transparency and collaboration, enabling other researchers to inspect, replicate, and build upon the work. It fosters a culture of sharing and innovation within the scientific community, potentially leading to faster advancements and broader adoption of automated testing methodologies. Furthermore, open-source implementations often encourage peer review and validation, enhancing the credibility and trustworthiness of the research findings. On the other hand, a

closed-source implementation may be chosen for reasons such as intellectual property protection, proprietary technology, or commercial interests. While closed-source implementations may limit access to the underlying codebase, they still contribute valuable insights and advancements to the field of automated testing. This breakdown between open-source and closed-source implementations reflects the diversity of approaches and considerations among researchers in sharing their work with the broader scientific community. This section answered a part of **RQ6**.

5 CONCLUSION

This comparative study serves as a catalyst for further exploration, inquiry, and collaboration in the dynamic intersection of LLMs and software testing. By synthesizing insights from a diverse array of research papers, we have provided a roadmap for leveraging the potential of LLMs to address the challenges and seize the opportunities to ensure the quality, reliability, and security of software systems in an increasingly changing and digitized world. Despite valuable insights, this study has limitations. Generalizability may be affected by specific LLMs, datasets, and metrics, potentially neglecting software testing diversity. Effectiveness might vary due to unexplored factors like data quality and fine-tuning strategies, requiring ongoing validation due to LLM technology's evolution. This study highlights future research paths emphasizing scalability and efficiency in various application domains. Improving LLM training and integration with advanced technologies like symbolic reasoning could enhance testing frameworks. Combining LLM-driven methods with domain-specific knowledge may lead to context-aware strategies, while exploring LLM-based testing in security and compliance could advance software quality assurance.

REFERENCES

[1] Zakaria Aoujil, Mohamed Hanine, Emmanuel Soriano Flores, Md. Abdus Samad, and Imran Ashraf. 2023. Artificial Intelligence and Behavioral Economics: A Bibliographic Analysis of Research Field. *IEEE Access* 11 (2023), 139367 – 139394. https://doi.org/10.1109/ACCESS.2023.3339778

[2] Lenz Belzner, Thomas Gabor, and Martin Wirsing. 2024. Large Language Model Assisted Software Engineering: Prospects, Challenges, and a Case Study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 14380 LNCS (2024), 355 – 374. https://doi.org/10.1007/978-3-031-46002-9_23

[3] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07)*. 85–103. https://doi.org/10.1109/FOSE.2007.25

[4] Mohamed Boukhlif, Mohamed Hanine, and Nassim Kharmoum. 2023. A Decade of Intelligent Software Testing Research: A Bibliometric Analysis. *Electronics (Switzerland)* 12, 9 (2023). https://doi.org/10.3390/electronics12092109

[5] Mohamed Boukhlif, Nassim Kharmoum, Mohamed Hanine, Chaimae Elasri, Wajih Rhalem, and Mostafa Ezziyyani. 2024. Exploring the Application of Classical and Intelligent Software Testing in Medicine: A Literature Review. In *International Conference on Advanced Intelligent Systems for Sustainable Development (AI2SD'2023)*, Mostafa Ezziyyani, Janusz Kacprzyk, and Valentina Emilia Balas (Eds.). Springer Nature Switzerland, Cham, 37–46. https://doi.org/10.1007/978-3-031-52388-5_4

[6] Aidan Dakhama, Karine Even-Mendoza, W.B. Langdon, Hector Menendez, and Justyna Petke. 2024. SearchGEM5: Towards Reliable Gem5 with Search Based Software Testing and Large Language Models. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 14415 LNCS (2024), 160 – 166. https://doi.org/10.1007/978-3-031-48796-5_14

[7] Karim El Bouchti, Soumia Ziti, Fouzia Omary, and Nassim Kharmoum. 2019. A new database encryption model based on encryption classes. *Journal of Computer Science* 15, 6 (2019), 844–854. https://doi.org/10.3844/jcssp.2019.844.854

[8] Chaimae Elasri, Nassim Kharmoum, Fadwa Saoiabi, Mohamed Boukhlif, Soumia Ziti, and Wajih Rhalem. 2024. Applying Graph Theory to Enhance Software Testing in Medical Applications: A Comparative Study. In *International Conference on Advanced Intelligent Systems for Sustainable Development (AI2SD'2023)*, Mostafa Ezziyyani, Janusz Kacprzyk, and Valentina Emilia Balas (Eds.). Springer Nature Switzerland, Cham, 70–78. https://doi.org/10.1007/978-3-031-52388-5_7

[9] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards Autonomous Testing Agents via Conversational Large Language Models. *Proceedings - 2023 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023* (2023), 1688 – 1693. https://doi.org/10.1109/ASE56229.2023.00148

[10] Vahid Garousi, Sara Bauer, and Michael Felderer. 2020. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology* 126 (2020), 106321. https://doi.org/10.1016/j.infsof.2020.106321

[11] Qiuhan Gu. 2023. LLM-Based Code Generation Method for Golang Compiler Testing, Chandra S., Blincoe K., and Tonella P. (Eds.). *ESEC/FSE 2023 - Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), 2201 – 2203. https://doi.org/10.1145/3611643.3617850

[12] Vitor Guilherme and Auri Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. *ACM International Conference Proceeding Series* (2023), 15 – 24. https://doi.org/10.1145/3624032.3624035

[13] Andreas Happe and Jürgen Cito. 2023. Getting pwn'd by AI: Penetration Testing with Large Language Models, Chandra S., Blincoe K., and Tonella P. (Eds.). *ESEC/FSE 2023 - Proceedings of the 31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2023), 2082 – 2086. https://doi.org/10.1145/3611643.3613083

[14] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *Proceedings - International Conference on Software Engineering* (2023), 2312 – 2323. https://doi.org/10.1109/ICSE48619.2023.00194

[15] Nassim Kharmoum, Karim El Bouchti, Naziha Laaz, Wajih Rhalem, and Yassine Rhazali. 2020. Transformations' study between requirements models and business process models in MDA approach. *Procedia Computer Science* 170 (2020), 819–824.

[16] Nassim Kharmoum, Sara Retal, Soumia Ziti, and Fouzia Omary. 2020. A novel automatic transformation method from the business value model to the UML use case diagram. In *Advanced Intelligent Systems for Sustainable Development (AI2SD'2019) Volume 3-Advanced Intelligent Systems for Sustainable Development Applied to Environment, Industry and Economy*. Springer, 38–50. https://doi.org/10.1007/978-3-030-36671-1_4

[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. *Proceedings - International Conference on Software Engineering* (2023), 919 – 931. https://doi.org/10.1109/ICSE48619.2023.00085

[18] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library. https://doi.org/10.1002/9781119202486

[19] Rudolf Ramler, Stefan Biffl, and Paul Grünbacher. 2006. *Value-Based Management of Software Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 225–244. https://doi.org/10.1007/3-540-29263-2_11

[20] Max Schafer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85 – 105. https://doi.org/10.1109/TSE.2023.3334955

[21] Sahar Tahvili and Leo Hatvani. 2022. *Artificial Intelligence Methods for Optimization of the Software Testing Process: With Practical Examples and Exercises*. Elsevier. 1 – 204 pages. https://doi.org/10.1016/C2021-0-00433-8

[22] Christos Tsigkanos, Pooja Rani, Sebastian Muller, and Timo Kehrer. 2023. Large Language Models: The Next Frontier for Variable Discovery within Metamorphic Testing?, Zhang T., Xia X., and Novielli N. (Eds.). *Proceedings - 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023* (2023), 678 – 682. https://doi.org/10.1109/SANER56733.2023.00070

[23] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 14073 LNCS (2023), 321 – 335. https://doi.org/10.1007/978-3-031-35995-8_23

[24] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for Test Script Generation and Migration: Challenges, Capabilities, and Opportunities. *IEEE International Conference on Software Quality, Reliability and Security, QRS* (2023), 206 – 217. https://doi.org/10.1109/QRS60937.2023.00029

[25] Yifan Zhang, Dave Towey, and Matthew Pike. 2023. Automated Metamorphic-Relation Generation with ChatGPT: An Experience Report, Shahriar H., Teranishi Y., Cuzzocrea A., Sharmin M., Towey D., Majumder AKM.J.A., Kashiwazaki H., Yang J.-J., Takemoto M., Sakib N., Banno R., and Ahamed S.I. (Eds.). *Proceedings - International Computer Software and Applications Conference* 2023-June (2023), 1780 – 1785. https://doi.org/10.1109/COMPSAC57700.2023.00275