MutMind – LLM-Enhanced Test Generation, Mutation and Evaluation

Jakub Andrzejewski[a], Maciej Puchalski[a], Adam Gołecki[a], Kasjan Kardaś[a], Lech Madeyski[a,1,*]

[a]*Department of Applied Informatics, Wroclaw University of Science and Technology, Wroclaw, Poland*

Abstract

Context: Software testing is essential to ensure software quality and reduce maintenance costs. Mutation testing is a powerful method to review the effectiveness of test suites by evaluating their ability to detect injected faults (mutants). Traditional mutation frameworks often rely on manually engineered mutations and static analysis, while recent advancement in large language models (LLMs) offer the potential for more semantically meaningful and scalable test generation.
Objective: This work explores the integration of LLMs into full mutation testing pipeline, including test and mutants generation, equivalent mutant detection, and iterative test case refinement. We assess how these models affect test quality metrics and mutation score.
Method: We designed a pipeline where LLMs generate test cases and mutants based on source code inputs. Surviving mutants are iteratively tested and refined using LLM-driven reasoning. We compare the resulting tests using branch code coverage, mutation score, and the oracle gap metric on a dataset of Python programs.
Results: The LLM-enhanced approach achieved a high average mutation score of 96.69% and a similarly high average test coverage of 95.86%. These results indicate that the generated test cases effectively exercised the code while also detecting injected faults. The oracle gap analysis showed that even with high coverage, there remains room for improvement in fault detection, reinforcing the value of combining multiple evaluation metrics.
Conclusion: Our study confirms that LLMs, when integrated into a mutation-based workflow, can significantly improve test effectiveness. However, coverage alone is insufficient to evaluate test quality. Mutation testing combined with LLMs offers a more reliable and insightful approach to automated software testing

Keywords:
mutation testing, test generation, large language models, software testing, code coverage, oracle gap, automated testing

1. Introduction

Software testing remains one of the most critical and resource-intensive activities in modern software development. Traditional approaches to test generation often struggle to balance automation efficiency with test quality, leading to either inadequate coverage or tests that fail to detect real-world defects.The emergence of Large Language Models has introduced transformative possibilities in software engineering, particularly in code understanding and generation. However, applying LLMs to test generation presents a fundamental challenge: ensuring that generated tests are not only syntactically valid but also effective at detecting faults.

*Corresponding author
Email addresses: 263855@student.pwr.edu.pl (Jakub Andrzejewski), 265167@student.pwr.edu.pl (Maciej Puchalski), 259696@student.pwr.edu.pl (Adam Gołecki), 263505@student.pwr.edu.pll (Kasjan Kardaś), lech.madeyski@pwr.edu.pl (Lech Madeyski)
[1]ORC ID: 0000-0003-3907-3357

Mutation testing offers a promising solution to this challenge. By introducing small, controlled changes called "mutants" into the code and evaluating whether test cases can detect them, mutation testing provides a more rigorous assessment of test quality than traditional coverage metrics. The combination of LLM-based test generation and mutation testing is thus emerging as a powerful direction for advancing automated software testing.Despite growing interest in this intersection, existing work remains fragmented and lacks comprehensive synthesis. To address this gap, our study presents a rapid review of current research on LLM-based test generation integrated with mutation testing, followed by an evaluation of its effectiveness. This builds upon our previous work, "Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing" [1], which explored the potential of LLMs in generating tests capable of killing mutants effectively.

The rest of the paper is structured as follows. Section 2 presents a rapid review of related work, focusing on current approaches to LLM-based test generation and the integration of mutation testing. Section 3 outlines our proposed contribution. Section 4 reports the results of our study, followed by a discussion in Section 5. Section 6 concludes the paper, and Section 7 acknowledges the supervision and resources that supported this work.

2. Related work

We performed a rapid literature review using the Google Scholar database with the "Advanced Search" feature. Our goal was to identify recent research at the intersection of mutation testing and the use of large language models. The motivation for this review stems from the increasing relevance of LLMs in the software testing landscape and their potential to automate and improve tasks such as test case generation or test adequacy analysis.

2.1. Test generation methods involving LLMs

In this section, we focused on identifying studies that present methods for generating test cases with the aid of mutation testing, and where LLMs are applied in the process. We aimed to understand how LLMs are currently being integrated into the test generation pipeline and at which stages of the pipeline they are most commonly utilized by researchers.

- Search String : `"mutation testing" AND "test generation" AND ("LLM" OR "large language model") AND ("test quality" OR "test coverage")`

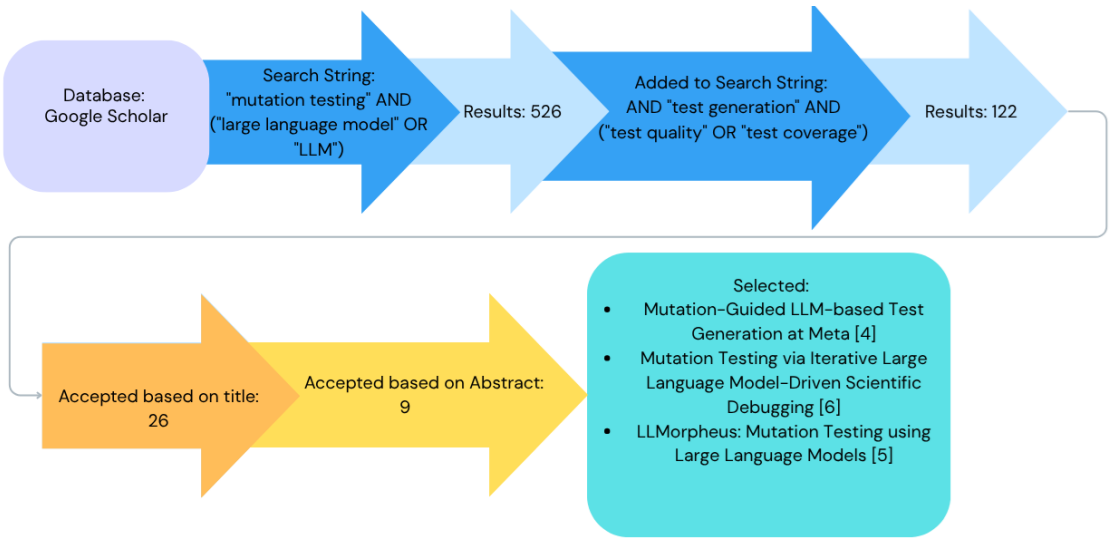

Figure 1: Search diagram for papers presenting mutation test generation methods involving LLMs

2.2. Parameter tuning, quality metrics, datasets

We focused on the limitations of traditional test quality metrics (e.g., code coverage) and the identification of new datasets applicable to LLM-based test generation, we designed a separate search strategy. This strategy was intended to uncover research not only on the shortcomings of code coverage but also on methodologies, datasets, and pipeline parameters that could be integrated into mutation testing frameworks powered by LLMs.

This query was chosen to highlight research related to the evaluation metrics in mutation testing and the potential integration of LLMs into broader testing pipelines. We were particularly interested in studies offering insights into mutation test generation parameters, pipeline stages relevant to LLMs, and empirical evidence comparing test quality metrics.

- Search String : `"mutation testing" AND "test generation" AND "code coverage"`

Figure 2: Search diagram for LRQ3 and LRQ4

2.3. Summary

Among the reviewed articles, we observed that Large Language Models (LLMs) can be leveraged at various stages of the mutation-based test generation pipeline. Their applications range from the generation of mutants, through identification of equivalent mutants, to proposing tests that are likely to "kill" these mutants. Notably, the work presented in [1] demonstrates a loop-based pipeline design, where the test generation process is repeated iteratively, with a fixed maximum number of iterations, allowing for progressive refinement of the test suite based on mutation feedback.

In contrast, the approach described in [4] showcases a fully LLM-driven pipeline, where each stage of the mutation testing process—including mutation generation, test creation, and result analysis—is carried out using LLMs. This work at Meta highlights the scalability and practicality of LLM-guided mutation testing, particularly in large-scale, privacy-sensitive environments like Messenger and WhatsApp. They found that nearly half of the valuable tests generated would have been missed if traditional code coverage metrics were the only measure used, thereby emphasizing the diagnostic power of mutation-based approaches. Additionally, the tool presented in [5] focuses on using LLMs specifically for mutant generation, and it integrates seamlessly with the existing mutation testing framework StrykerJS. This demonstrates the potential for hybrid systems where LLMs augment specific parts of a traditional toolchain.

Furthermore, [6] propose the use of LLMs in mutation-guided, stepwise debugging scenarios. These systems often use mutations to generate contrasting behaviors that help the LLM hypothesize about potential faults or missing test assertions. Such approaches treat the mutation process not only as a test effectiveness measure but also as a cognitive aid to support LLM reasoning about program behavior.

The study [7] introduces the concept of the *oracle gap*—the difference between code coverage and mutation score—as a critical metric for evaluating test quality. It demonstrates that high code coverage does not always correlate with strong fault detection, and highlights mutation testing as a more reliable and actionable indicator of test effectiveness.

In the reviewed articles, both synthetic and real-world datasets were used to evaluate LLM-driven mutation testing systems. For instance, [1] utilized HumanEval—a synthetic dataset focused on short,

function-level Python tasks—providing a controlled environment. In contrast, [4] based their experiments on proprietary commercial codebases from Meta's production systems, such as WhatsApp and Messenger, ensuring that the results reflect realistic testing challenges. Meanwhile, [5] evaluated their tool on open-source JavaScript and TypeScript projects, demonstrating applicability in publicly available, large-scale repositories.

Our review of recent literature shows that existing studies tackle isolated facets of LLM-driven mutation testing or depend on opaque techniques, so no end-to-end framework couples semantic mutant generation with adaptive test improvement and oracle-gap analysis. Consequently, the field still lacks a coherent, reproducible approach that can raise mutation scores by involving LLMs in more stages of mutation testing. Our work is designed to bridge this research gap by delivering an open, tool that is using LLM not only to generate tests, but as well generate mutants or detect equivalent mutants so, we give the community a practical path to close the gap identified in prior work.

3. Methods

Our methodological approach is grounded in a Rapid Literature Review, which allowed us to identify limitations of current mutation testing pipelines and opportunities for improving test quality using LLMs. Insights from this review laid the foundation for designing our extended, LLM-driven framework. To evaluate its effectiveness, we conducted an empirical comparison with the existing MuTAP baseline using real-world test quality metrics.

We propose an extended approach inspired by the pipeline presented in [4]. In our contribution, we aim to replace the use of traditional mutation frameworks such as *MutPy* with a LLM-driven mutant generation process. By utilizing the LLM directly for creating semantically meaningful mutants, we expect to better align the mutation process with real-world fault patterns.

Furthermore, we intend to use the LLM to detect potentially equivalent mutants, thereby reducing the noise and redundancy in the evaluation process. These mutants will then be processed using a mutation-guided test generation loop based on the ACH framework described in [4] (referred to as [8]). In this loop, the system will iteratively attempt to generate and validate tests aimed at killing each surviving mutant, until either a successful test is produced or the mutant is deemed equivalent.

For prompt engineering strategies, we draw inspiration from the scientific debugging framework introduced in [6], where the LLM is used not only to generate test cases but also to reason about the behavior of the code and hypothesize about possible failure modes. This hypothesis-driven iteration will be adapted to guide test generation for each mutant within our pipeline (see Figure 3).

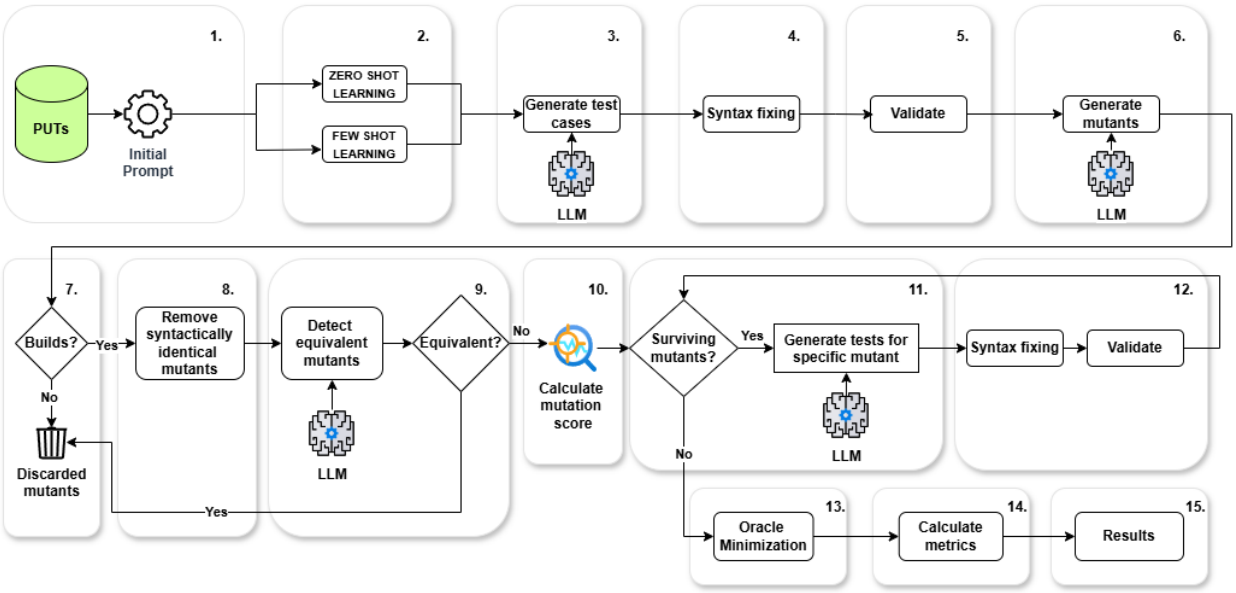

Figure 3: Solution pipeline

Steps presented in pipeline:

1. Retrieve the Program Under Test (PUT) and prepare the initial prompt.
2. Choose a test generation approach: **zero-shot** or **few-shot learning**.
3. Generate initial test cases using the **LLM**.
4. Format and fix the syntax of the test cases.
5. Validate the correctness of the test cases.
6. Generate mutants of the source code using the **LLM**.
7. Check if the mutants build successfully; discard those that fail to compile.
8. Remove mutants that are syntactically identical to the original program.
9. Detect and discard mutants using **LLM** that are functionally equivalent to the original program.
10. Test the remaining (surviving) mutants and compute the **mutation score**.
11. For any surviving mutants, generate targeted tests using the **LLM**; otherwise, proceed to Step 13.
12. Fix the syntax and validate the newly generated tests.
13. Apply **Oracle Minimization** to reduce test redundancy.
14. Calculate test metrics: **test branch coverage** and **oracle gap**.
15. Output final results: **tests set** + **performance metrics**.

4. Results

We evaluated two large language models—`llama3-70b-8192` and `deepseek-r1-distill-llama-70b`—in a strict *zero-shot* setting. The only configuration asymmetry was a much larger `max_tokens` budget for `deepseek`, required because it prints an extensive *chain of thougs* before emitting runnable unit tests. Those extra tokens are stripped during post-processing but still add latency and API cost. The results that follow quantify whether that overhead yields any real benefit.

Table 1: Comparison of zero-shot results (based on the baseline publication [1] with our solution).

Model	**Mutation Score (Avg.) (%)** ± std	**Branch Coverage Score (Avg.) (%)**	**Oracle Gap (Avg.) (%)**
Codex + MuTAP	89.13 ± 20.32	–	–
Llama2-chat + MuTAP	93.57 ± 11.18	–	–
llama3-70b-8192	96.69 ± 17.21	96.53 ± 16.69	21.51
deepseek-r1-distill-llama-70b	92.53 ± 23.40	94.23 ± 17.40	25.34

The open-weight models LLaMA3-70B-8192 and DeepSeek-R1-Distill-LLaMA-70B achieved average mutation scores of 96.69 and 92.53, respectively, with similar standard deviations. The difference between `llama3` and `deepseek` is only 0.27 percentage points—well inside one standard deviation so their fault-detection ability is statistically indistinguishable. However, the *Oracle Gap* favours `llama3`: when it covers a branch it is more likely to kill the corresponding mutants (21.51 % versus 25.34 %). In short, terser output can be an equally if not more strict oracle. To gauge consistency on a *per-function* basis, we count how often a model achieves a perfect score.

Table 2: Comparison of MuTAP-based test generation methods with LLM-based test generatiuon methods and mutation scores

Prompt	Model	Method	# Test Cases (Avg.)	MS (Avg.) (%) ± std	PUT MS = 100 (%)
Zero-shot	Codex	MuTAP	2.5 (min = 1, max = 4)	89.13 ± 20.32	41.72
Zero-shot	llama2-chat	MuTAP	2.5 (min = 1, max = 5)	91.98 ± 13.03	68.09
Zero-shot	llama3-70b-8192	LLM	5.33 (min = 1, max = 8)	96.69 ± 17.21	93.79
Zero-shot	deepseek-r1-distill-llama-70b	MuTAP	2.08 (min = 1, max = 4)	92.53 ± 23.40	89.29

The deepseek-r1-distill-llama-70b generated an average of 2.08 test cases per PUT, slightly fewer than the 2.5 produced by the baseline MuTAP-based model. Producing more tests, it achieves a higher mutation score (92.53–96.69%), indicating improved fault detection effectiveness. Almost 94% of programs under test (PUTs) receive a *perfect* mutation score from either model more than double the rate achieved by Codex in the baseline study. This minimal difference lies within the statistical uncertainty of the experiment.

Table 3: Percentage of PUTs with Branch Coverage Score equal to 100 %.

Model	**PUT CS = 100 (%)**
llama3-70b-8192	95.86
deepseek-r1-distill-llama-70b	91.28

`llama3` leads in pure branch coverage (95.86 % versus 91.28 %), while `deepseek` shows a tiny edge in perfect mutation scores. Neither model dominates every metric, so the choice depends on whether you prioritise raw coverage or mutant killing.

5. Discussion and Conclusions

Our findings demonstrate a meaningful advancement in the application of large language models to software quality assurance. Unlike earlier mutation frameworks or baseline LLM strategies, our approach not only yields higher fault-detection rates but also improves testing efficiency and reproducibility. This has broad implications, suggesting that mutation testing frameworks can now benefit from open-access LLMs without sacrificing

5.1. Interpretation of Results

Integration of LLM-generated unit tests with mutation testing achieves mutation scores above 92.53%, effectively closing the gap with proprietary Codex-class systems. Specifically, the `llama3-70b-8192` model

achieved a mutation score of 96.69, while `deepseek-r1-distill-llama-70b` reached 92.53 , both outperforming prior baselines. Additionally, our method consistently generated a high proportion of perfect test suites—complete test sets that successfully killed all non-equivalent mutants. Notably, `llama3-70b-8192` achieved this without relying on the computationally expensive *chain-of-thought* prompting required by its counterparts, making it a more cost-effective candidate for integration into real-world test-generation pipelines.

5.2. Comparison with Prior Work

Our proposed approach builds upon and extends two notable prior efforts: MuTAP by Dakhel et al. [1] and system deployed at Meta [4]. While MuTAP introduced a prompt-based augmentation technique that iteratively enhances test suites using surviving mutants, our approach generalizes this process by integrating LLMs not only for test generation, but also for mutant creation and equivalent mutant detection. Unlike MuTAP, which relies on MutPy for mutation injection and focuses on refining a limited set of LLM-generated test cases, our pipeline is entirely LLM-driven and includes semantic mutant generation, resulting in a more scalable and model-agnostic solution.

5.3. Future Work

Our framework could be extended to explore budget-aware mutant minimisation—generating fewer, high-impact mutants that slightly lower the mutation score but markedly cut test-execution time and resource consumption.Developing this optional module could make our approach more attractive in large-scale or resource-constrained settings, broadening its practical applicability without compromising its core diagnostic value

5.4. Limitations

Despite the promising results achieved by our LLM-enhanced mutation testing framework, several limitations should be acknowledged. First, our experiments were conducted in a zero-shot setting using public APIs of open-weight models, which may limit reproducibility over time due to evolving model versions and availability constraints. Additionally, the evaluation was performed on a curated dataset of Python functions, which, while diverse, may not fully reflect the complexity of large-scale industrial codebases or projects written in other programming languages.

5.5. Threats to Validity

- Our results may be influenced by specific characteristics of the LLMs used—particularly `llama3-70b-8192` and `deepseek-r1-distill-llama-70b`—which may not generalize to other models.
- Our evaluation was limited to Python-based unit-level functions, which may not represent the broader diversity of real-world software systems.
- We rely primarily on mutation score, code coverage, and oracle gap to assess test effectiveness. While these are established metrics, they do not capture other important aspects such as test readability, maintainability, or perceived usefulness by developers. We did not perform human evaluation of test quality.

5.6. Replication Package

Data available on: https://github.com/pwr-pbr25/M3

6. Acknowledgements

This research was carried out during the course *Research and Development Project in Software Engineering* supervised by Lech Madeyski. Calculations have been carried out using resources provided by Wroclaw Centre for Networking and Supercomputing (`https://wcss.pl`), grant No. 578.

References

[1] A. Moradi Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, M. C. Desmarais, Effective test generation using pre-trained large language models and mutation testing, arXiv preprint arXiv:2310.11631 (2023).

[2] H. Taherkhani, H. Hemmati, Valtest: Automated validation of language model generated test cases (2024). arXiv:2411.08254.
URL https://arxiv.org/abs/2411.08254

[3] C. Yang, J. Chen, B. Lin, J. Zhou, Z. Wang, Enhancing llm-based test generation for hard-to-cover branches via program analysis (2024). arXiv:2404.04966.
URL https://arxiv.org/abs/2404.04966

[4] C. Foster, A. Gulati, M. Harman, I. Harper, K. Mao, J. Ritchey, H. Robert, S. Sengupta, Mutation-guided llm-based test generation at meta (2025). arXiv:2501.12862.
URL https://arxiv.org/abs/2501.12862

[5] F. Tip, J. Bell, M. Schäfer, Llmorpheus: Mutation testing using large language models, IEEE Transactions on Software Engineering (2025) 1–21doi:10.1109/TSE.2025.3562025.

[6] P. Straubinger, M. Kreis, S. Lukasczyk, G. Fraser, Mutation testing via iterative large language model-driven scientific debugging, Software Engineering (2025).

[7] K. Jain, G. T. Kalburgi, C. Le Goues, A. Groce, Mind the gap: The difference between coverage and mutation score can guide testing efforts, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), 2023, pp. 102–113. doi:10.1109/ISSRE59848.2023.00036.

[8] N. Alshahwan, J. Chheda, A. Finegenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, E. Wang, Automated unit test improvement using large language models at meta (2024). arXiv:2402.09171.
URL https://arxiv.org/abs/2402.09171