GEM: A Framework for Strengthening LLM-Generated Unit Tests Using Mutation Feedback

Arda Celik and Qusay H. Mahmoud

Department of Electrical, Computer and Software Engineering
Ontario Tech University
Oshawa, ON, L1G 0C5 Canada

{arda.celik, qusay.mahmoud}@ontariotechu.net

Abstract. *Large Language Models (LLMs) are increasingly used for automated unit test generation and can produce executable tests with substantial structural coverage. However, recent empirical studies indicate that such tests often rely on weak or superficial assertions, leading to limited fault-detection capability despite extensive code coverage. This paper introduces GEM (Generate-Execute-Mutate), an automated framework that systematically strengthens test oracles to improve the mutation-based adequacy of LLM-generated unit tests. GEM integrates three stages into a unified pipeline: LLM-based test synthesis, execution-driven self-repair of failing tests, and mutation-guided oracle refinement. The framework follows a modular hexagonal architecture and supports multiple programming languages through pluggable adapters for test execution, coverage analysis, and mutation testing. GEM was evaluated on three established benchmarks across Python, Java, and C++, using multiple state-of-the-art LLMs, and was compared with the automated testing tool Pynguin. Experimental results reveal a persistent gap between coverage and mutation score in baseline LLM-generated tests. Under the evaluated setup, mutation-guided strengthening improved mutation scores on Python and yielded smaller gains on Java, while execution-driven self-repair improved executability across several model and dataset settings.*

1. Introduction

Large Language Models (LLMs) are a rapidly evolving class of artificial intelligence systems designed to understand and generate natural language at scale. Beyond their original focus on language tasks, LLMs have demonstrated strong capabilities in software engineering, including code generation, bug detection, and documentation. Their ability to translate natural language descriptions into executable code has attracted significant attention, as it lowers barriers between design intent and implementation and broadens participation in software development [Rasnayaka et al. 2024].

The adoption of LLMs in software engineering research and practice has accelerated markedly in recent years. Surveys report their use across tasks, including code generation, summarization, translation, vulnerability detection, and question answering over codebases [Sallou et al. 2024]. This trend is reinforced by industrial tools that integrate LLM-based code generation into development environments, positioning LLMs as a transformative element of modern software engineering workflows.

Despite these advances, software testing remains a critical yet under-resourced activity. Unit testing plays a foundational role in validating functional correctness and supporting long-term maintainability [Schäfer et al. 2024], but developers often perceive test creation as tedious and insufficiently rewarded, leading to chronic underinvestment [Schäfer et al. 2024, Straubinger and Fraser 2023] . Industrial studies further highlight persistent constraints related to time, staffing, and organizational prioritization [Garousi et al. 2020]. These challenges have motivated research into automated test generation. Traditional techniques such as search-based testing and symbolic execution can achieve substantial structural coverage, but they often face scalability issues and struggle to generate meaningful test oracles [Daka and Fraser 2014]. Empirical studies show that developers' main difficulties lie not only in selecting code to test, but also in determining appropriate assertions and validation criteria, a challenge widely known as the oracle problem [Daka and Fraser 2014]. This gap underscores the limitation of focusing on execution alone without ensuring behavioral verification.

As unit testing requires diverse inputs and behaviors to expose faults, syntactic correctness and executability alone are insufficient indicators of test adequacy [Wang et al. 2024]. Multiple reviews report that high coverage achieved by LLM-generated tests does not necessarily translate to effective fault detection [Zhang et al. 2025, Facundo Molina et al. 2025]. Empirical analyses reveal that generated assertions are often superficial or semantically weak, leading to poor fault detection despite extensive code execution [Facundo Molina et al. 2025, Fan et al. 2023].

This paper presents GEM, a framework that integrates LLM-based test generation, execution-driven self-repair, and mutation-guided oracle strengthening in a single iterative workflow. GEM treats mutation analysis not only as an evaluation metric but as active feedback for refining assertions and addressing weaknesses in LLM-generated test oracles. The framework supports multiple languages through a modular architecture and integrates with existing testing and mutation tools in a way intended to support practical experimentation and future extension. To evaluate the proposed framework, this study addresses the following research questions:

- **RQ1:** How effective is GEM's integrated pipeline in improving executability and mutation-based adequacy?
- **RQ2:** How does GEM compare with existing automated testing tools under mutation- and coverage-based evaluation?
- **RQ3:** How does GEM perform across different programming languages, datasets, and LLMs?

GEM is evaluated on Python, Java, and C++ benchmarks using multiple LLMs. The results reveal a persistent gap between coverage and mutation score in LLM-generated tests, while showing measurable improvements in mutation-based adequacy across several evaluated settings, with the clearest gains observed in the Python evaluations.

The contributions of this work are:

- A unified workflow combining test synthesis, self-repair, and mutation-guided oracle strengthening.
- A practical implementation supporting Python, Java, and C++ via modular adapters.

- Evidence demonstrating persistent coverage-mutation gaps and mutation-score improvements through oracle strengthening in several evaluated settings.

The remainder of this paper is organized as follows. Section 2 presents related work on unit test generation with LLMs. Section 3 describes the GEM framework architecture. Section 4 presents the experimental design, including the setup and evaluation protocol. Section 5 reports the results organized around the research questions. Section 6 provides a discussion. Section 7 outlines threats to validity, and Section 8 concludes with a summary and directions for future research.

2. Related Work

2.1. LLM-Based Unit Test Generation

Research on using LLMs for test case generation has largely developed along four main directions, namely prompt engineering, feedback-driven refinement, pre-training or fine-tuning, and hybrid [Celik and Mahmoud 2025]. Early work showed that Codex- and GPT-style models can generate executable, readable, and framework-compliant unit tests from source code, often achieving substantial structural coverage [Lemieux et al. 2023]. Later studies found that effectiveness depends heavily on model choice and prompting, with results varying across projects and benchmarks [Yang et al. 2024]. Comparisons with SBST tools, such as EvoSuite, indicate that LLMs can match or exceed coverage and improve readability in some cases, but may lag behind in fault detection and assertion quality [Tang et al. 2024]. Hybrid approaches embed LLMs within SBST workflows to overcome coverage plateaus, yielding statistically significant gains over SBST or LLM-only baselines [Lemieux et al. 2023]. Prompt engineering is critical as structured prompts, added context, and decomposition techniques such as method slicing improve coverage for complex methods [Yang et al. 2024]. Beyond prompting, fine-tuning and specialization further enhance performance, including improved assertion generation and consistent gains over prompt-only methods across tasks and metrics [Tufano et al. 2022]. Overall, LLMs are effective generators of executable, high-coverage tests, but prior work largely emphasizes coverage metrics, motivating research on improving behavioral adequacy and oracle strength.

2.2. Weak Oracles in LLM-Generated Tests

Recent empirical studies identify oracle weakness, rather than insufficient execution coverage, as the primary limitation of LLM-generated tests. Although LLMs can produce compilable assertions, these often fail to capture precise functional intent or to expose real faults, and oracle accuracy is highly sensitive to prompt design and contextual information [Bodicoat et al. 2026]. Multiple works report a clear disconnect between coverage and fault-detection capability, where tests achieve high coverage but rely on shallow assertions that result in low mutation scores and limited bug detection [Facundo Molina et al. 2025, Hossain and Dwyer 2025]. Qualitative analyses reveal recurring patterns of trivial or tautological assertions, such as non-null checks or assertions that mirror implementation logic, which offer little discriminatory power against faulty behavior and persist unless enhanced through fine-tuning, external tool support, or iterative and multi-agent refinement [Hayet et al. 2025]. Large-scale studies further show that while LLM-generated oracles can approach human-written tests in average mutation

score, they continue to struggle with complex behaviors, underscoring the need for techniques and evaluation criteria that explicitly target oracle strength rather than coverage alone [Molinelli et al. 2025].

2.3. Mutation-Guided Test Improvement

Several approaches integrate mutation feedback into LLM-based workflows, using surviving mutants to reveal behavioral weaknesses. Methods such as MuTAP and MUTGEN incorporate mutation information into prompts and iteratively refine or extend tests, achieving higher mutation scores than coverage-driven tools or vanilla LLM prompting [Dakhel et al. 2023, Wang et al. 2025]. Industrial evidence from Meta's ACH system further shows that mutation-guided LLM workflows can generate regression tests that expose previously uncaught faults, even without coverage increases [Foster et al. 2025]. However, these systems are typically presented as approach-specific pipelines often tied to particular languages/toolchains. In contrast, GEM operationalizes mutation feedback specifically for oracle strengthening within a unified, language-agnostic generate–execute–repair–mutate workflow, to make mutation-driven refinement deployable across ecosystems. Comparative evaluations confirm that mutation metrics uncover quality differences not captured by coverage or executability alone, motivating mutation-aware, feedback-driven generation pipelines [Cabral et al. 2025].

2.4. Self-Repair and Iterative Refinement

LLM-based unit test generation frequently yields non-executable tests due to compilation errors, missing dependencies, or incorrect API usage. To mitigate this, many approaches adopt execution-feedback-based self-repair, where compiler or runtime errors are iteratively fed back to the LLM. ChatTester shows that validate-and-fix loops driven by compilation diagnostics substantially increase the number of compilable tests [Yuan et al. 2024], while ChatUniTest combines rule-based and LLM-guided repair in a generation-validation-repair workflow to improve executability and coverage [Chen et al. 2024].

More recent work extends self-repair with richer feedback and iterative refinement. TestART interleaves generation and repair using template-based fixes and coverage feedback to improve pass rates and coverage [Gu et al. 2025], and LLMLOOP generalizes this idea through multiple automated loops, including compilation repair, test failure fixing, and mutation-guided improvement [Ravi et al. 2025].

Despite these advances, self-repair techniques primarily address executability and structural adequacy. Strengthening assertions to improve fault detection remains substantially harder, leaving the oracle problem largely unresolved.

3. The GEM Framework

This section presents GEM, an automated test generation framework that uses LLMs to synthesize unit tests with integrated self-repair and mutation driven oracle strengthening. GEM combines initial test generation, execution-based repair, and mutation-guided refinement in a single workflow to improve mutation based test adequacy. In its current form, the framework follows an implementation driven approach, using feedback derived from the program under test rather than an external specification such as contracts, requirements, or reference outputs. The system is implemented as a TypeScript command

line application and follows a hexagonal architecture with ports and adapters that separate orchestration logic from language-specific tooling and LLM integrations, enabling extensibility across testing ecosystems. Figure 1 illustrates this architecture.

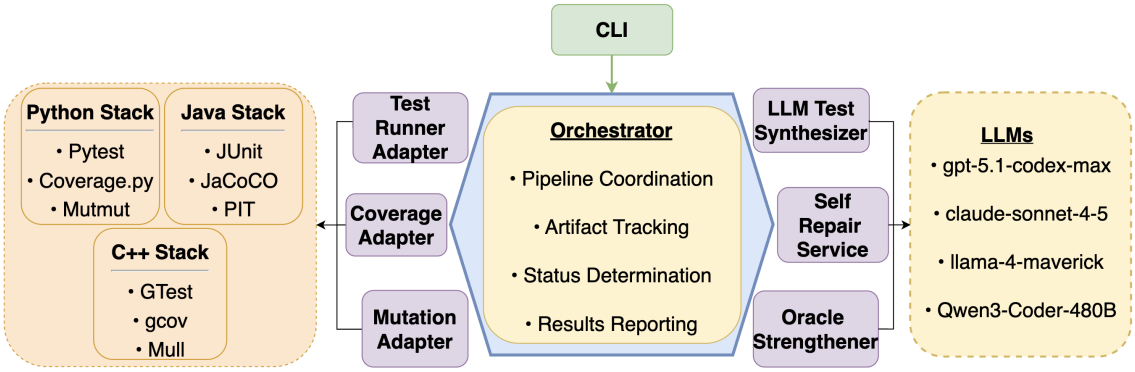

Figure 1. System Architecture.

At a high level, GEM proceeds through five connected stages: initial test synthesis, test execution, self-repair, mutation testing, and oracle strengthening. A central orchestrator manages the full lifecycle and coordinates specialized components responsible for test synthesis, repair, execution, coverage parsing, mutation analysis, oracle strengthening, artifact tracking, and structured reporting. This separation allows the core workflow to remain stable while language-specific adapters handle framework dependent operations.

Given a target source file, GEM first prompts the LLM to generate an initial test file and writes the result to a workspace. The generated tests are then executed and diagnostics are collected. If compilation or runtime failures occur, the framework enters a self-repair loop in which the LLM receives error feedback and attempts to correct the tests. Once the test suite executes successfully, GEM performs mutation testing to evaluate fault detection strength. If surviving mutants remain and strengthening is enabled, the framework prompts the LLM to revise the suite by adding or modifying assertions that target those mutants. Final coverage, mutation, and execution outcomes are then collected in a structured JSON report.

Oracle strengthening is implemented as a batch-oriented, whole-suite refinement step guided by surviving mutants. When a test suite executes successfully but does not reach the target mutation score, GEM extracts structured information about the surviving mutants, including the mutation operator, code diff, source location, and related coverage context. To control cost and prompt size, the framework randomly samples up to N surviving mutants from the full survivor set, where N is 10 by default, using Fisher-Yates shuffling without further prioritization. The sampled mutants, together with the source code, current test suite, and mutation results, are incorporated into a strengthening prompt that instructs the LLM to generate stronger tests with more discriminative assertions. The revised suite is then re-executed, repaired again if necessary up to the configured limit, and re-evaluated using mutation testing. GEM records mutation scores before and after strengthening, runs for up to a configurable number of iterations, which is 1 by default, and stops early if all mutants are killed or no improvement is observed.

The framework achieves language independence by defining common interfaces for test execution, coverage analysis, and mutation testing. It implements separate

language-specific adapters for each stack to execute tests, collect coverage, and run mutation analysis. In the evaluated implementation, Python relied on pytest, coverage.py, and mutmut, while Java used JUnit, JaCoCo, and PIT, and C++ used GTest, Gcov, and Mull. This architecture allows the same orchestration logic to be reused across languages while integrating with established testing ecosystems.

4. Experimental Design

4.1. Experimental Setup

GEM was evaluated across three datasets containing Python, Java, and C++ programs, utilizing four different LLMs. The datasets for evaluation included ULT (UnLeakedTestBench) [Huang et al. 2025], HumanEval-C++, and HumanEval-Java [Zheng et al. 2023]. The LLMs used in this evaluation were gpt-5.1-codex-max (OpenAI), claude-sonnet-4-5-20250929 (Anthropic), meta-llama/llama-4-maverick-17b-128e-instruct (Meta), and Qwen/Qwen3-Coder-480B-A35B-Instruct-FP8 (Alibaba).

For the gpt-5.1-codex-max model, the reasoning effort was set to Low through the Responses API, with a maximum allowance of 10,000 output tokens. The "extended thinking" variable was enabled for the claude-sonnet-4-5-20250929 model, which was configured with a token budget of 1024, a temperature setting of 1, and a maximum output token limit of 10,000. Both the Qwen/Qwen3-Coder-480B-A35B-Instruct-FP8 and meta-llama/llama-4-maverick-17b-128e-instruct models had their temperature set to 0.5, with maximum output tokens capped at 10,000 and 8192, respectively.

Static prompts were defined for all three evaluation stages, which involved interactions with the LLMs: initial test synthesis, self-repair, and oracle strengthening. Each of these steps consisted of system and user prompts, along with specific interactions tailored for each programming language. Three key configuration variables guided the evaluations: `maxSelfRepairAttempts`, `maxStrengtheningIterations`, and `maxSurvivingMutantsPerIteration`. These were set to 3, 1, and 10, respectively, to limit self-repair attempts to three, allow a single strengthening effort, and restrict processing to a maximum of ten surviving mutants. `maxStrengtheningIterations` was set to 1 to keep evaluation cost and latency bounded and to assess whether a single conservative refinement step could still produce measurable gains. This setting was intended as a lower-bound evaluation of the framework's strengthening effect rather than an estimate of its maximum achievable improvement, and future work will explore the impact of allowing multiple strengthening iterations.

A test generation attempt was deemed successful if no error-level diagnostics were recorded during the pipeline execution. Conversely, a test generation attempt was classified as failed under several conditions such as the generated tests failing to compile or execute after all repair attempts and test execution timeouts. The framework also monitored 11 additional statuses, such as "self-repair failed for initial tests" and "mutation score stayed the same after oracle strengthening," to obtain finer-grained diagnostic information from the evaluation..

For the evaluation, all 163 programs from HumanEval-Java and all 164 programs from HumanEval-C++ were assessed. In contrast, due to its larger size, the ULT dataset

was sampled using three randomly selected subsets of 100, 200, and 300 programs. Each subset was evaluated once across all four LLMs, except for the subset of 100 samples, which was run three times for each model.

4.2. Evaluation Protocol

For each program in the evaluated datasets, GEM executed the following evaluation pipeline. First, an initial set of unit tests was generated using the selected LLM and language-specific prompts. The generated tests were then executed, and compilation or runtime errors were addressed through up to three rounds of execution-driven self-repair. Programs that failed to produce executable tests within this limit were classified as unsuccessful.

For successfully executed test suites, statement and branch coverage and initial mutation scores were computed. Mutation analysis was then used to identify surviving mutants, from which up to 10 were randomly sampled and incorporated into a single oracle-strengthening iteration. The strengthened test suites were subsequently re-evaluated to obtain final coverage and mutation scores.

Evaluation outcomes were quantified using success rate, average number of self-repair attempts, branch coverage, and mutation score, with statement coverage and surviving mutants used as supplementary diagnostic metrics. Vanilla LLM-generated test suites prior to strengthening served as the baseline for assessing mutation-guided improvements. This protocol was designed to support analysis of oracle quality, executability, self-repair effectiveness, and performance variation across models, languages, and datasets.

5. Results

5.1. Effectiveness of GEM's Integrated Pipeline for Executability and Mutation-Based Adequacy

Table 1 summarizes the main quantitative outcomes of the evaluation and serves as the primary reference for success rate, average self-repair attempts per sample, and pre- and post-strengthening mutation and branch coverage results. For ULT (100), the reported values are averages over three runs on the same randomly selected 100-program sample, whereas ULT (200), ULT (300), HumanEval-Java, and HumanEval-C++ were each evaluated once. The Self Repair column reports the average number of self-repair attempts per sample. Mutation Score Initial and Branch Cov. Initial denote the metrics measured before oracle strengthening, while the Final columns denote the post-strengthening values.

The results show a consistent relationship between model capability, executability, and the effectiveness of mutation-guided refinement. Across the ULT splits, Codex and Sonnet achieved the highest success rates while requiring fewer repair attempts on average, whereas Qwen and Llama required substantially more repair attempts and still produced much lower success rates. This pattern also appeared on HumanEval-Java and HumanEval-C++, where Codex and Sonnet remained the most reliable models overall. A further caveat is that a non-trivial portion of ULT failures was caused by unresolved import errors in the dataset rather than purely by model behavior, which means the reported executability for the stronger models is somewhat conservative.

Table 1. Evaluation Results.

Model/ Tool	Dataset	Success Rate	Self Repair	Mutation Score		Branch Cov.	
				Initial	Final	Initial	Final
Codex	ULT (100)	73.00	0.456	72.71	81.85	84.30	88.81
	ULT (200)	81.50	0.465	72.41	81.95	84.56	88.25
	ULT (300)	79.67	0.476	73.25	81.89	84.53	89.63
	HumanEval-Java	99.39	0.447	90.14	91.10	92.79	93.02
	HumanEval-C++	92.07	0.451	86.20	86.20	78.03	79.17
Sonnet	ULT (100)	70.33	0.91	81.09	86.08	90.98	91.80
	ULT (200)	74.50	0.875	80.20	87.48	90.35	91.98
	ULT (300)	72.33	0.976	79.40	84.17	90.98	92.46
	HumanEval-Java	98.16	0.527	90.10	90.63	93.70	93.75
	HumanEval-C++	95.73	0.536	87.40	87.40	79.25	80.12
Qwen	ULT (100)	30.33	1.59	80.69	84.64	86.31	86.67
	ULT (200)	33.00	1.66	78.24	84.29	88.68	89.48
	ULT (300)	28.33	1.776	74.66	81.31	83.89	85.24
	HumanEval-Java	75.46	1.644	89.90	90.94	92.99	93.23
	HumanEval-C++	78.05	0.859	87.32	87.32	79.75	79.98
Llama	ULT (100)	22.00	1.83	73.29	80.67	87.40	89.70
	ULT (200)	23.50	1.995	74.85	82.47	86.58	87.94
	ULT (300)	21.67	1.94	70.94	78.49	88.28	89.80
	HumanEval-Java	73.01	1.466	89.63	90.85	92.24	92.61
	HumanEval-C++	50.00	1.46	86.71	86.71	76.71	77.20
Pynguin	ULT (100)	67.00	N/A	N/A	23.97	N/A	46.83
	ULT (200)	70.00	N/A	N/A	31.16	N/A	51.27
	ULT (300)	71.33	N/A	N/A	26.92	N/A	52.71

With respect to oracle strengthening, the clearest gains were observed on the ULT benchmark. Across all ULT sample sizes, mutation score increased after strengthening for all evaluated LLMs, with the largest improvements generally observed for Codex and Sonnet. In contrast, branch coverage changed only modestly. The central pattern visible

in Table 1 is that high initial structural coverage did not imply equally strong mutation performance, and mutation-guided refinement helped narrow that gap.

Answering RQ1: Under the evaluated setup, GEM's integrated pipeline improved executability and mutation-based adequacy most clearly on ULT, with improvements varying by model and dataset. Self-repair was most effective for stronger models, which required fewer repair attempts and achieved higher success rates, while mutation-guided oracle strengthening improved mutation score across the evaluated ULT settings even when initial branch coverage was already high.

5.2. Comparison with Existing Automated Testing Tools

The comparison with Pynguin shows that the main advantage of GEM is not merely that tests execute, but how strong those tests are once produced. Pynguin achieved moderate success rates on the ULT splits, but its final mutation scores remained much lower than those of the strengthened GEM pipelines. Its final branch coverage was also substantially below the branch coverage obtained by the stronger GEM configurations. In practical terms, this means that Pynguin could often generate runnable tests, but those tests were much less effective at killing mutants than the tests produced and strengthened by GEM. Among the GEM configurations, Codex provided the strongest overall balance across success rate, mutation score, and branch coverage, while Sonnet was also consistently strong, especially on mutation score.

An approximate view of computational overhead can be obtained from the recorded total runtimes for the ULT (100) evaluations. Codex completed in approximately 33 minutes, Sonnet in 2 hours and 28 minutes, Llama in 17 minutes, Qwen in 1 hour and 43 minutes, and Pynguin in 1 hour and 21 minutes. As expected, evaluations with larger sample sizes generally required more time. Among the tested configurations, Sonnet was the slowest and Llama was the fastest. Besides architectural and model size differences across configurations, Llama's lower runtime is likely influenced in part by the specialized inference hardware provided through the Groq platform.

Answering RQ2: On the evaluated ULT splits and under the mutation- and coverage-based evaluation used in this study, the stronger GEM configurations outperformed Pynguin in mutation score and branch coverage. However, computational overhead varies substantially across model configurations and serving infrastructure, meaning GEM's runtime can be either lower or higher than Pynguin depending on the selected model.

5.3. Cross-Language, Cross-Dataset, and Cross-Model Performance of GEM

The performance of GEM varied by language and dataset. The largest improvements were observed on the ULT dataset, where the gap between initial mutation score and final mutation score was the widest. On HumanEval-Java, initial mutation scores were already around 90 percent, so oracle strengthening produced only modest gains. On HumanEval-C++, mutation scores remained unchanged after strengthening across all evaluated LLMs. This suggests that the combination of already high baseline adequacy and the constraints of the Mull-based mutation pipeline limited the opportunity for additional assertion-level improvement. Across models, stronger models not only achieved better executability, but also converted the strengthening step into more reliable end-to-end gains. Weaker models

still benefited on successful samples, but their low success rates limited the overall impact of refinement.

Answering RQ3: GEM improved mutation-based adequacy on the evaluated Python and Java benchmarks, while no post-strengthening mutation-score gains were observed on the evaluated C++ benchmark. The magnitude of improvement depended strongly on dataset difficulty, language ecosystem, and baseline model performance. The clearest gains occurred on the ULT dataset, while Java showed smaller gains overall and C++ showed no additional gains under the current setup due to mutation-tool constraints and high initial scores.

6. Discussion

6.1. Challenges in Tool Integration and Language-Specific Mutation Frameworks

Implementing GEM across multiple languages exposed substantial integration challenges arising from heterogeneous testing and mutation ecosystems. Although GEM's core principles are language-agnostic, their realization required extensive tool-specific adaptations due to the lack of standardized mutation testing interfaces and reporting formats.

Mutation frameworks differ in execution models, supported operators, result formats, and failure reporting mechanisms. These inconsistencies required significant adapter code and custom parsing logic to normalize outputs, making tool integration a major source of system complexity. Much of GEM's implementation effort was therefore devoted to bridging tools designed for human inspection with automated workflows.

In Python, mutmut reports results through console logs and HTML pages, requiring fragile text-based parsing. This approach is sensitive to format changes and increases maintenance overhead. In contrast, Java's PIT provides structured XML and CSV reports, substantially reducing integration complexity and improving reliability.

Overall, the absence of unified mutation testing APIs and reporting schemas limits the scalability of language-agnostic test-strengthening systems. Reliance on ad-hoc parsing and tool-specific logic introduces fragility and maintenance burdens, representing a practical obstacle to deploying mutation-driven workflows in production environments.

6.2. Practical Implications for AI-Assisted Development Tools

The findings have direct implications for AI-assisted development tools. Although GEM operates as an external pipeline, its architecture aligns with the execution-edit-feedback loops used by modern coding assistants and agentic systems.

Tools such as GitHub Copilot, Codex CLI, Claude Code, Cursor, and autonomous agents already support code editing, command execution, and result interpretation. GEM can be integrated as an execution oracle via a stable CLI interface. In this model, tools generate or modify tests, invoke GEM for analysis, and use its structured outputs such as surviving mutants and quality metrics to guide refinement.

This integration requires no tight coupling, provided that GEM exposes stable interfaces and machine-readable artifacts. Self-repair can reduce manual debugging by resolving common errors before user exposure, improving usability and trust while mutation-guided refinement can address pervasive oracle weakness without increasing developer burden.

6.3. Toward AI-Native Testing Infrastructure

While GEM demonstrates the feasibility of mutation-driven workflows, its implementation also reveals deeper limitations in current testing infrastructure. Most test runners, coverage tools, and mutation frameworks are designed for human-centered use, emphasizing textual logs and HTML reports. These choices hinder their use in automated reasoning loops.

This motivates the development of AI-native testing infrastructure. Such tools would prioritize structured, semantically rich outputs, stable programmatic interfaces, and incremental execution models suitable for rapid feedback. Rather than reporting only aggregate metrics, they would expose fine-grained information about uncovered behaviors, surviving mutants, and oracle deficiencies in formats directly consumable by LLMs and agents.

AI-native tooling does not require replacing existing frameworks. Lightweight wrappers and abstraction layers could provide standardized interfaces across languages, reducing integration complexity and improving robustness. Unified mutation representations and reporting protocols would further support scalable agentic workflows.

From this perspective, GEM serves as a transitional system that demonstrates the value of mutation-driven feedback while exposing shortcomings in current ecosystems. By identifying sources of integration complexity and illustrating the benefits of structured feedback, this work clarifies requirements for next-generation testing infrastructure. Advances in AI-native tools can substantially improve automated test quality and developer experience in AI-assisted and agent-driven environments.

7. Threats to Validity

7.1. External Threats

Several factors limit the generalizability of these results. First, the evaluation focused on isolated functions and classes from benchmark datasets rather than real production systems. Real-world codebases involve complex dependencies, architectural constraints, concurrency, and distributed behavior that were not represented, potentially introducing challenges absent from this study.

Second, the analysis relies heavily on the ULT dataset, as HumanEval-derived datasets were too simple to challenge frontier models. High baseline mutation scores on Java and C++ HumanEval indicate that mutation feedback was often unnecessary for simpler tasks. Although ULT is more challenging, missing import statements caused 18.50% - 23.00% of samples to fail, reducing effective sample size and potentially biasing results toward simpler programs.

Third, cost and execution time measurements depend on the specific infrastructure and API configurations used. Alternative deployment environments may yield substantially different performance and cost profiles.

Finally, evaluations were conducted on specific model versions. Given rapid model evolution, absolute performance and relative rankings may change over time, limiting temporal generalizability.

7.2. Internal Threats

Several factors affect attribution of improvements to the mutation-driven approach. The strengthening process was limited to a single iteration targeting at most ten surviving mutants. While chosen for efficiency, this conservative setup likely understates the method's potential. Allowing additional iterations would probably yield larger gains at higher computational cost. The observed 6-10 percentage point improvements should therefore be viewed as lower bounds on mutation-score gains under the conservative configuration used in this study, rather than as lower bounds on broader fault-detection effectiveness.

Mutation scores were computed using framework-specific operators (mutmut, PIT, and Mull). Differences in operator design and fault coverage make results framework-dependent and may influence scores independently of test quality.

Another mutation-specific threat concerns equivalent mutants, which the current workflow does not explicitly detect. Consequently, some of the surviving mutants selected for strengthening may be non-actionable. Because the process was limited to a single iteration over at most ten surviving mutants, the inclusion of equivalent mutants may reduce strengthening efficiency and cause the reported mutation-score improvements to understate the achievable gains within the current mutation-based setting.

An additional internal threat arises from the coupling between GEM's optimization signal and its evaluation signal. GEM strengthens tests using surviving mutants derived from the same mutation-analysis process later used to assess post-strengthening effectiveness. Moreover, the refinement process is implementation-driven and does not incorporate an independent specification source such as contracts, requirements, or reference outputs. As a result, improvements in mutation score and reductions in surviving mutants may partially reflect success on the mutation-based objective used during refinement, rather than fully independent evidence of broader defect-detection capability or semantic correctness with respect to an external specification. Accordingly, the post-strengthening results should be interpreted primarily as evidence that GEM improves mutation-based adequacy with respect to implementation-derived mutants. Independent evaluation on held-out mutants, real faults, or other fault-based benchmarks would be needed to establish stronger claims about general fault detection.

Model performance may also depend on evaluation-time hyperparameters such as temperature and output length. Although held constant for comparability, alternative settings could produce different outcomes.

Despite these limitations, the results showed a recurring coverage-mutation gap across the evaluated settings, and mutation-guided strengthening improved mutation-based adequacy in several of them. These findings support the conclusion that mutation-guided strengthening can improve mutation-based adequacy under the evaluated setup. However, the extent to which these gains translate to independent fault-detection effectiveness remains an open question for future evaluation.

8. Conclusion and Future Work

This paper presented GEM, an automated test-generation framework that integrates LLM-based test synthesis, automatic self-repair, and mutation-driven oracle strengthening in a unified pipeline. It adopts a hexagonal architecture with pluggable adapters for Python,

Java, and C++ across language-specific testing, coverage, and mutation frameworks. The results indicate that mutation testing complements coverage metrics when assessing LLM-generated test suites under mutation-based evaluation. The observed gains, particularly on the ULT dataset, suggest that mutation-driven feedback can strengthen assertions beyond what coverage alone captures. However, since GEM uses mutation feedback for both refinement and evaluation, the results should be interpreted as evidence of improved mutation-based adequacy rather than definitive proof of broader fault-detection effectiveness.

Future work should first examine the effects of additional strengthening iterations and larger mutant samples, as the conservative configuration used in this study likely underestimates the benefits achievable within the current mutation-based setting. At the same time, the lack of standardized interfaces across mutation testing frameworks remains a major integration challenge, underscoring the need for unified APIs to better support language-agnostic workflows. Broader evaluation is also needed across a wider range of unit test generation benchmarks, with particular care taken to minimize data contamination, and against a larger set of existing tools. An especially important next step is to assess GEM on real-world open-source and industrial systems with multi-file structures, external dependencies, and realistic build and test pipelines, in order to better understand its practical applicability beyond benchmark-style tasks. Future studies should also include independent evaluation on held-out mutants, real faults, or other fault-based benchmarks to determine whether the observed gains generalize beyond the mutation-guided setting used here. In parallel, a more systematic analysis of computational overhead is needed, including phase-wise runtime breakdowns, normalized cost comparisons, and token and API usage, in order to better characterize efficiency and cost trade-offs. Another promising avenue is to explore metamorphic testing as a complementary mechanism for oracle improvement, particularly to assess whether LLMs can infer candidate metamorphic relations or translate known relations into executable follow-up tests. Finally, integrating mutation-driven strengthening into mainstream development environments appears to be a promising direction for making automated test generation more practical in production settings.

References

Bodicoat, A., Jahangirova, G., and Terragni, V. (2026). Understanding LLM-Driven Test Oracle Generation. arXiv:2601.05542 [cs].

Cabral, P., Arruda, J., Souza, C., and Pinto, V. (2025). Evaluating llm-generated unit tests with mutation testing: Chatgpt vs deepseek. In *Anais do XXIV Simpósio Brasileiro de Qualidade de Software*, pages 120–130, Porto Alegre, RS, Brasil. SBC.

Celik, A. and Mahmoud, Q. H. (2025). A Review of Large Language Models for Automated Test Case Generation. *Machine Learning and Knowledge Extraction*, 7(3).

Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., and Yin, J. (2024). ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 572–576, Porto de Galinhas Brazil. ACM.

Daka, E. and Fraser, G. (2014). A Survey on Unit Testing Practices and Problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, Naples, Italy. IEEE.

Dakhel, A. M., Nikanjam, A., Majdinasab, V., Khomh, F., and Desmarais, M. C. (2023). Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing. arXiv:2308.16557 [cs].

Facundo Molina, Gorla, A., and d'Amorim, M. (2025). Test Oracle Automation in the Era of LLMs. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–24.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., and Zhang, J. M. (2023). Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, Melbourne, Australia. IEEE.

Foster, C., Gulati, A., Harman, M., Harper, I., Mao, K., Ritchey, J., Robert, H., and Sengupta, S. (2025). Mutation-Guided LLM-based Test Generation at Meta. arXiv:2501.12862 [cs].

Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K., and Eldh, S. (2020). Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process*, 32(8):e2251.

Gu, S., Zhang, Q., Li, K., Fang, C., Tian, F., Zhu, L., Zhou, J., and Chen, Z. (2025). TestART: Improving LLM-based Unit Testing via Co-evolution of Automated Generation and Repair Iteration. arXiv:2408.03095 [cs].

Hayet, I., Scott, A., and d'Amorim, M. (2025). ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance. *IEEE Transactions on Software Engineering*, 51(1):305–319.

Hossain, S. B. and Dwyer, M. B. (2025). TOGLL: Correct and Strong Test Oracle Generation with LLMS. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1475–1487, Ottawa, ON, Canada. IEEE.

Huang, D., Zhang, J. M., Harman, M., Zhang, Q., Du, M., and Ng, S.-K. (2025). Benchmarking llms for unit test generation from real-world functions.

Lemieux, C., Inala, J. P., Lahiri, S. K., and Sen, S. (2023). CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, Melbourne, Australia. IEEE.

Molinelli, D., Di Grazia, L., Martin-Lopez, A., Ernst, M. D., and Pezzè, M. (2025). Do LLMs generate useful test oracles? an empirical study with an unbiased dataset. In *ASE 2025: Proceedings of the 39th Annual International Conference on Automated Software Engineering*, Seoul, South Korea.

Rasnayaka, S., Wang, G., Shariffdeen, R., and Iyer, G. N. (2024). An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 111–118, Lisbon Portugal. ACM.

Ravi, R., Bradshaw, D., Ruberto, S., Jahangirova, G., and Terragni, V. (2025). LLM-LOOP: Improving LLM-Generated Code and Tests Through Automated Iterative Feedback Loops. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 930–934, Auckland, New Zealand. IEEE.

Sallou, J., Durieux, T., and Panichella, A. (2024). Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106, Lisbon Portugal. ACM.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2024). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1):85–105.

Straubinger, P. and Fraser, G. (2023). A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 80–90, Florence, Italy. IEEE.

Tang, Y., Liu, Z., Zhou, Z., and Luo, X. (2024). ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation. *IEEE Transactions on Software Engineering*, 50(6):1340–1359.

Tufano, M., Drain, D., Svyatkovskiy, A., and Sundaresan, N. (2022). Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 54–64, Pittsburgh Pennsylvania. ACM.

Wang, G., Xu, Q., Briand, L. C., and Liu, K. (2025). Mutation-Guided Unit Test Generation with a Large Language Model. arXiv:2506.02954 [cs].

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q. (2024). Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering*, 50(4):911–936.

Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q., Chu, X., Zhou, J., Liang, G., Wang, Q., and Chen, J. (2024). On the Evaluation of Large Language Models in Unit Test Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619, Sacramento CA USA. ACM.

Yuan, Z., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., and Lou, Y. (2024). Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726.

Zhang, Q., Fang, C., Gu, S., Shang, Y., Chen, Z., and Xiao, L. (2025). Large Language Models for Unit Testing: A Systematic Literature Review. arXiv:2506.15227 [cs].

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. (2023). Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.