

Beyond Rule-Based Mutation Testing: Test-Aware Mutant Generation Using Large Language Models

Nils Kiele 

Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Zainab Saad 

Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Zirui Wang 

Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Steve Drew 

Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Samira Ebrahimi Kahou 

Department of Electrical and Software Engineering, University of Calgary, AB, Canada

Abstract

Background. Mutation testing evaluates test-suite adequacy by injecting synthetic faults into program code. However, traditional rule-based tools often generate large numbers of trivial, redundant, or equivalent mutants that limit their practical use for identifying gaps in a test suite. While recent large language model (LLM)-based approaches generate more realistic faults, most remain test-blind: The model sees only the source code and cannot reason about what existing tests already cover. Ignoring such tests means neglecting context that could help generate higher-quality mutants and thus stronger tests to patch the remaining test suite gaps.

Aims. We propose *test-aware* mutant generation, in which an LLM receives the problem statement, canonical solution and base tests in a single prompt, and must generate a nontrivial mutant that passes the base unit tests.

Method. We evaluate this approach across a set of five LLMs – Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, Qwen3-32B – on the HumanEval and MBPP benchmarks. The extended EvalPlus test suites serve as an automated oracle to verify whether surviving mutants represent genuine bugs.

Results. Test-aware prompting yields verified fault rates of 87.7% (HumanEval) and 79.1% (MBPP), meaning these mutants pass all base tests but are caught by the oracle. This vastly outperforms the matched test-blind prompting (which yields only 12.2% and 23.0%, respectively) and the traditional rule-based tool mutmut (4.4% and 5.7%). While fault subtlety (the fraction of extended tests a mutant fails) remains comparable across all three methods, test-awareness minimizes the computational cost per verified fault, compared to test-blind prompting.

Conclusions. Exposing an LLM to existing unit tests shifts mutant generation from untargeted bug injection toward effective discovery of weaknesses in an existing test suite. Our work establishes a concrete foundation for future research to scale test-aware mutant generation to production-level environments.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Computing methodologies → Natural language generation

Keywords and phrases Mutation testing, large language models

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.49

Category Technical Track Paper



© Nils Kiele, Zainab Saad, Zirui Wang, Steve Drew, and Samira Ebrahimi Kahou; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón; Article No. 49; pp. 49:1–49:20



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supplementary Material *Software*: <https://github.com/codingWhale13/test-aware-mutants>
 archived at `swh:1:dir:753cf1771d9178c5f28653eb3b401645689508de`
Dataset: <https://zenodo.org/records/20195774>

Funding The Machine Intelligence and Reasoning Lab at the University of Calgary is funded by NSERC, CIFAR, DRAC, Alberta Innovates, Schulich Momentum, and Denvr Dataworks.

Acknowledgements We thank Prof. Ahmad Abdel Latif for helpful discussions and feedback.

1 Introduction

Mutation testing is a well-established technique for evaluating and improving the quality of a software test suite by injecting synthetic faults, called *mutants*, into source code and checking whether existing tests detect them [10, 17]. A surviving mutant is one that passes all tests and reveals a gap in the test coverage [12]. Despite the benefits of mutation testing, its practical adoption remains limited. A key challenge is the equivalent mutant problem: traditional rule-based frameworks like the Python tool *mutmut*¹ generate many mutants that are syntactically different but semantically identical to the original code. Distinguishing these from genuine faults is generally undecidable [17]. Empirical studies estimate that 5–20% of generated mutants are equivalent [15]. In industrial settings, the computational cost of executing tests against large mutant pools and the manual effort needed to triage them are the main bottlenecks [19]. For example, calculating the mutation score (ratio of “killed” mutants to total mutant count) on the two billion lines of Google code is generally infeasible [18]. Furthermore, even non-equivalent mutants often provide little value, as many are trivially killed by basic assertions rather than exposing concealed logical flaws. This motivates our search for fewer, higher-value mutants that reveal genuine test-suite weaknesses.

Large language models (LLMs) are a promising tool in efficiently generating mutants. Recent work has shown that LLM-generated mutants are more diverse and behaviorally closer to real bugs than those from rule-based tools, having a $1.8\times$ higher real fault detection rate [29]. However, existing LLM-based approaches such as *LLMorpheus* [27] and *BugFarm* [9] use *test-blind* prompting where the model is provided only the source code and has no knowledge of what the existing tests already cover. Without this context, the LLM cannot explicitly target gaps in test coverage and may instead produce mutants that are redundant, trivially caught, or semantically equivalent. Instead of improving test quality, these low-quality mutants present unnecessary additional cost either in terms of human review or computational resources required for repeated execution at every future code deployment. In contrast, a *test-aware* model can reason about the behavioral constraints already covered by the tests and focus its mutations on what the suite misses. Although these techniques enhance mutation testing, they require iterative prompting and are computationally expensive. To reduce this overhead, we propose a single-prompt, test-aware mutant generation approach.

More concretely, the LLM is provided with both the source code and the existing test suite in one prompt and asked to generate a subtle logical bug that survives the current tests. This approach isolates the impact of test-awareness in prompting LLMs for mutant generation and investigates whether the reasoning of the model alone (without a feedback loop) is enough to efficiently generate mutants. However, there is a methodological challenge: verifying whether a surviving mutant is a genuine, non-obvious fault, rather than an equivalent program, typically requires manual review. To address this, we use the Python-based *EvalPlus*

¹ <https://mutmut.readthedocs.io/en/latest/>

benchmark [14], which extends HumanEval [5] and MBPP [3] with $80\times$ and $35\times$ more tests, respectively. The base test suites (7.7 tests on average per HumanEval problem and 3 for MBPP) serve as the developer-written tests the LLM must evade. The extended suites act as an automated oracle, classifying a mutant that passes the base tests but fails at least one extended test as a verified, non-equivalent fault.

Through extensive experiments with five LLMs – Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, Qwen3-32B – we study various aspects of test-aware prompting for mutant generation and provide a comparison to its matched test-blind and exhaustive rule-based alternatives. Crucially, we show that including the existing test cases in the LLM’s context yields an increase of up to $7\times$ in the rate of verified fault generations. We then measure how many oracle test cases are triggered to evaluate the subtlety of generated mutants; a subtle mutant fails on fewer oracle tests. Finally, our cost analysis demonstrates that while rule-based techniques remain the cheapest overall, test-aware prompting significantly reduces the amortized token cost per verified fault compared to the test-blind baseline.

2 Related Work

Traditional mutation testing tools rely on basic syntactic mutation operations such as altering true/false values or modifying numeric literals. One such tool is the Python library `mutmut`. Although simple and fast, it produces many low-value candidates that do not compile, are trivially caught by basic tests or are semantically equivalent to the initial code. This motivates our work to focus on mutant quality over quantity by generating fewer yet more informative mutants. Our intuition is that a test-aware LLM can reason about the semantic blind spots in the code to produce targeted non-trivial bugs for important edge cases.

Test-Blind LLM Mutation. Several approaches use LLMs to generate mutants from source code without access to existing unit tests. μ BERT [7] generates mutants using the pretrained CodeBERT [8] model to predict masked code replacements. LEAM [25] trains a syntax-guided encoder-decoder on real-world bug-fix patches from GitHub for syntactically correct mutant generation. These methods produce mutants that resemble real faults more closely than semantics-agnostic operators, but they are limited in syntactic accuracy and mutant diversity. LLMorpheus [27] prompts an LLM to fill placeholders in JavaScript code, producing mutants that are more realistic than those from the state-of-the-art (SOTA) rule-based tool StrykerJS². BugFarm [9] uses attention analysis to find the least attended locations in the code and generate hard-to-detect bugs. The main limitation of these approaches is that they are test-blind and do not condition mutant generation on what the current tests already check. LLMut [29] is closest to our work and considers varying levels of contextual information in an LLM prompt. The authors report substantially higher real-bug detection than rule-based mutation, but with a lower compilation rate and higher duplication. Interestingly, while one of their prompts includes existing unit tests, it performs slightly worse than the variant that omits them. We hypothesize that this performance drop occurs because LLMut only appends the tests as passive context. In contrast, our approach succeeds by explicitly structuring the prompt as an adversarial task, instructing the model to bypass the provided tests to produce high-quality, targeted mutants.

² <https://github.com/stryker-mutator/stryker-js>

Adversarial and Iterative LLM Approaches. AdverTest [4] improves the fault detection on the Defects4J dataset [11] by using two LLM agents in an adversarial loop where one generates mutants to target blind spots, and the other generates tests. MutGen [30] is an iterative mutation-guided LLM-based test generation framework that incorporates mutant feedback (whether it survived or was killed by the tests) into the prompt for improved unit tests. Nafis et al. developed an iterative test generation pipeline where mutant score and test coverage are used as feedback signal for test case refinement [16]. These approaches show that feedback-driven workflows can be effective, but they also require repeated compilation, execution and re-prompting. A related iterative approach on the test-generation side lets the LLM form hypotheses about how to kill specific mutants and refine tests using execution feedback [22]. Compared to these iterative approaches, our proposed test-awareness yields strong results using only a single interaction with the LLM, reducing cost.

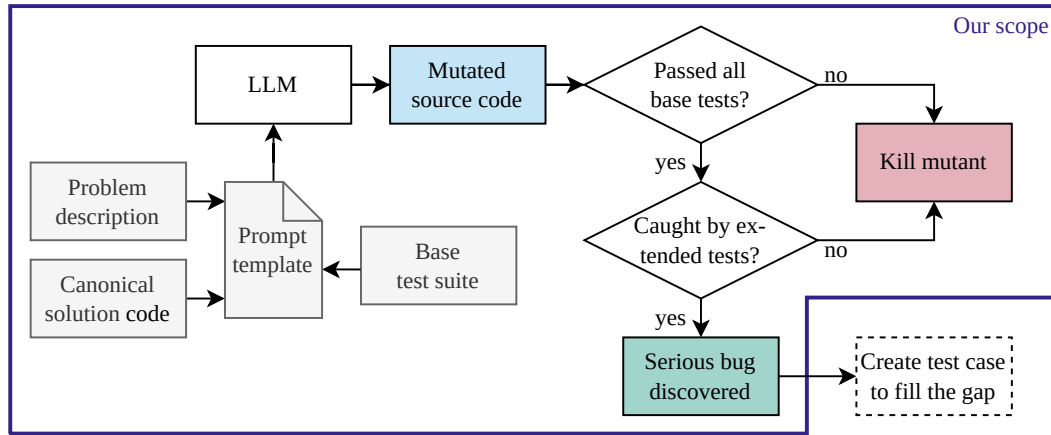
Mutation-Guided and Coverage-Guided Test Generation. A complementary line of work uses mutation or coverage information to improve LLM-generated tests rather than to generate mutants. MuTAP [6] uses rule-based mutants to iteratively improve LLM-generated tests. Meta’s TestGen-LLM provides both source code and existing tests to the model when improving unit tests [1]. CoverUp [2] iteratively targets uncovered code in Python regression test generation, and SymPrompt [21] exploits the multi-step reasoning capability of LLMs and generates test cases in multiple reasoning stages guided by execution path analysis. These approaches do not generate mutants, but they are consistent with our central idea that giving LLMs richer test-related context improves generation quality.

LLMs for Equivalent Mutant Detection. Equivalent mutants remain a fundamental challenge for scaling mutation testing. An empirical study on using LLMs for equivalent mutant detection evaluates several strategies such as fine-tuned code embeddings and prompting strategies on Java mutant pairs [26]. The results show that LLM-based techniques outperform prior equivalent mutant detection methods by an average F1 score of 35.69%. This work uses LLMs to filter equivalent mutants after generation, while we aim to reduce low-value mutant generation by giving the model access to the tests.

To the best of our knowledge, no prior work empirically studies the direct impact of test-awareness on LLM-based mutant generation. Test-blind methods cannot directly find test-suite gaps without iterative methods requiring expensive multi-step feedback. Coverage- and mutation-guided test-generation approaches show that test-related context helps LLM reasoning, but they apply that insight to the test side rather than mutant generation.

3 Method and Experimental Setup

Our objective is to generate mutants that reveal weaknesses in an existing test suite rather than producing arbitrary bugs. We formulate mutant generation as an adversarial task. Given a problem statement P , canonical solution code C , and base tests T_{base} , an LLM must produce a mutant C' that preserves the original function signature, passes T_{base} , and fails at least one test on the hidden oracle T_{oracle} . For every problem, model, and prompt configuration, the LLM generates exactly one mutant without compilation feedback, test-execution feedback, or iterative repair (except in the iterative ablation, see Section 4.4). Our single-shot test-aware mutation pipeline is described below and illustrated in Figure 1. Mutants are assigned an outcome label, as described in Table 1.



■ **Figure 1** Our proposed test-aware mutation pipeline. The LLM generates a **candidate mutant**, which is filtered by the base tests and then validated using the extended EvalPlus oracle, to automatically determine whether the mutant is a **serious gap in the test suite** or **not**.

1. **Prompting.** An LLM receives the problem statement, canonical solution and – in the test-aware setting – the base tests. The prompt asks for a subtle logical bug that is executable and passes the provided base tests.
2. **Extraction.** The raw response is parsed to extract the mutant code and discard the surrounding conversational text or markup. If the mutant omits the essential imports, the canonical solution imports are used instead to compile the mutant.
3. **Base-Test Filtering.** The extracted mutant is executed against the base tests and discarded if it throws an error, times out, or fails. This ensures that only eligible mutants are passed to the oracle, which (1) saves cost in running oracle tests on fewer mutants, (2) provides a controlled setting to compare test-awareness vs. test-blindness, and (3) verifies that the mutant is not a trivial bug, but rather has potential to be a genuine fault.
4. **Oracle Evaluation.** Surviving mutants are executed against the EvalPlus extended tests. A mutant that fails at least one oracle test is classified as a verified fault.

■ **Table 1** Each LLM-generated mutant is assigned one of the four mutually exclusive outcome classes, the meaning of which is explained in this table.

Mutant Type	Definition
INVALID	Mutant does not compile or times out during base-test execution.
REJECTED	Mutant runs without error, but is killed by at least one base test.
SUCCESS	Mutant evades the base tests but is detected by the oracle.
UNDETECTED	Mutant survives both the base tests and the oracle.

Benchmarks and Oracle Design. We evaluate generated mutants on EvalPlus [14], which extends two Python code-generation benchmarks with substantially stronger test suites for 164 HumanEval problems [5] and 378 MBPP problems [3]. While EvalPlus was designed to benchmark code generation quality, we repurpose it to determine the quality of the LLM-generated mutant. We view the few base tests for each problem as high-quality but incomplete developer-written tests that a good mutant must evade, while the much larger

extended test suite acts as an automated oracle. This design lets us automatically determine the distinction between mutants that are arbitrary syntax modifications vs. those that represent real semantic faults, overlooked by the base tests. We selected EvalPlus over traditional mutation testing benchmarks, such as Defects4J [11], because it enables the automated protocol described above to reliably classify mutants at scale without manual review. Furthermore, the authors of the original EvalPlus study [14] empirically confirm that the base tests in HumanEval and MBPP are often insufficient. By exposing flaws in code solutions that would previously have been accepted, their study provides direct evidence that the extended tests are significantly stronger. While these extended test suites are not perfect either, they serve as a useful tool for evaluating our approach.

Models and Baselines. We evaluate five LLMs: Gemini 3.1 Pro, Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini, and Qwen3-32B. These model choices represent a mix of open-source (Qwen3-32B), frontier SOTA models (Gemini 3.1 Pro) and more computationally efficient versions (Gemini 3 Flash, GPT 5.1 Codex Mini, GPT 4.1 Mini). These specific models have been used in recent software engineering literature for tasks such as vibe-coding benchmark [28], software issue resolving [23], code refactoring [24], and multi-requirement programming tasks [20]. All LLM-based experiments use temperature 0 to evaluate the LLMs in deterministic setting; however, this does not guarantee exact reproducibility because LLMs are non-deterministic by nature and produce stochastic outputs. We compare test-aware prompting against matched test-blind prompting and mutmut.

Model Prompts. We use two types of prompts, with two variants each. The design rationale for each variant is described below and summarized in Table 2. The prompt families are dubbed AGENT1 (**A**dversarial **G**eneration of **E**vasive **N**on-equivalent **T**ransformations 1) and AGENT2. Each has a test-aware (TA) and a matched test-blind (TB) variant that differ only in whether the base tests are included in the context. These prompt variants allow for a clean ablation of test-awareness. All prompts (see Table 2 below) share the same task: to generate a single mutant containing a subtle logical bug, given the problem statement and canonical solution. We study two effects: the contribution of test context (TA vs. TB) and the impact of reasoning structure (AGENT1’s free-form style vs. AGENT2’s template).

■ **Table 2** Summary of prompt variants used in our study. TA = test-aware; TB = test-blind.

Prompt	Tests Given?	Reasoning Scaffold
AGENT1-TA	Yes	Free-form
AGENT1-TB	No	Free-form
AGENT2-TA	Yes	Structured template
AGENT2-TB	No	Structured template

We now describe the prompts AGENT1 and AGENT2, followed by their full system and user section definitions. Content that appears *only* in the TA variant is marked with a **yellow background** in the full prompt. Placeholders shown in **red** are filled at runtime with the problem statement, canonical solution, and base tests.

AGENT1 is the primary prompt. The test-aware variant AGENT1-TA has the base tests along with the source code and instructs the model to (1) analyze the tests for blind spots on valid inputs, (2) plan a minimal mutation, and (3) “mentally dry-run” the mutation against each base test. These reasoning steps are elicited by the LLM inside `<thinking>` tags. The test-blind variant AGENT1-TB uses the same system prompt and output format but omits the base tests from the user message.

AGENT1 – Test-Aware (TA) and Test-Blind (TB) Variants**SYSTEM:**

You are an adversarial software tester and quality-assurance agent. Your goal is to introduce a subtle, realistic logic bug (a “mutant”) into the provided Python function.

Your objective is to exploit a semantic blind spot in the Base Tests. The mutated code **MUST** pass all the provided Base Tests, but it should fail on broader edge cases or uncommon inputs.

Constraints:

1. Preserve the original function signature and all required imports.
 2. The modified code must compile successfully without syntax errors.
 3. Make the bug subtle and realistic. Do not completely rewrite the function or introduce obvious crashes.
 4. The mutated code **MUST** pass **EVERY** provided Base Test.
 5. The bug **MUST** manifest on **VALID** inputs that adhere to the problem description.
 6. Do not introduce arbitrary “magic numbers”, epsilon values, or floating-point precision hacks.
- Format your output **EXACTLY** as follows:

<thinking>

1. Analyze the Base Tests to identify a blind spot on **VALID** inputs.
2. Plan a minimal mutation to exploit this.
3. Perform a mental dry-run against the Base Tests to ensure it passes.

</thinking>

<mutant>

[Insert fully modified Python code here, with no markdown formatting]

</mutant>

USER:

Problem statement:

{problem_statement}

Base Tests:

{base_tests}

Canonical Source Code:

{canonical_solution}

Generate the mutant.

AGENT2 describes the LLM reasoning process in a more structured way. The test-aware variant AGENT2-TA requires the LLM to fill the fields in a fixed template: (1) a short summary of the function’s computation, (2) base-test coverage (input types, value ranges, edge cases), (3) a list of two to three identified blind spots, (4) a short mutation plan specifying the exact line and change type, and (5) a per-test trace confirming correctness. This prompt is designed to reduce reasoning shortcuts and force LLM to explicitly analyze gaps in test coverage. Its test-blind counterpart AGENT2-TB only has the function summary and mutation plan fields, because coverage analysis is not possible without tests.

AGENT2 – Test-Aware (TA) and Test-Blind (TB) Variants

SYSTEM:

You are an adversarial mutation tester. Introduce exactly one subtle logic bug into the given Python function. The mutant MUST pass every base test but fail on other valid inputs.

Reason inside `<thinking>` tags using this exact structure, keep each field to one or two sentences maximum:

`<thinking>`

WHAT IT DOES: [One sentence: the function's core computation and return value]

BASE TEST COVERAGE:

- Input types tested: [e.g. positive ints, non-empty lists]
- Value ranges tested: [e.g. n in [1..10], lists of length 2–5]
- Edge cases tested: [e.g. duplicates, empty input, single element]

BLIND SPOTS: [List 2–3 valid input scenarios the base tests do NOT exercise.]

MUTATION PLAN: [One sentence: exactly what line to change and how.]

BASE TEST TRACE: [For each base test input, confirm the mutant output matches expected]

`</thinking>`

Then output the complete modified function:

`<mutant>`

[full function, no markdown]

`</mutant>`

USER:

Problem statement:

`{problem_statement}`

Original code:

`{canonical_solution}`

Base tests to bypass:

`{base_tests}`

Generate the mutant.

Evaluation Metrics. For the quantitative analysis of the efficacy of different prompting approaches, we use “effectiveness” in terms of success rate, “quality” in terms of fault subtlety, and “token cost” for generating a mutant. These metrics are calculated as follows:

- (a) **Effectiveness:** We measure the *success rate* as the fraction of generated mutants classified as **SUCCESS**:

$$M_{\text{eff}} = \frac{|\{\text{mutants classified as SUCCESS}\}|}{|\{\text{all generated mutants}\}|}.$$

A higher M_{eff} means that the LLM generates more verified, non-equivalent faults.

- (b) **Quality:** We measure fault subtlety using the *oracle kill rate*, defined for each **SUCCESS** mutant m as the fraction of extended oracle tests that detect it:

$$M_{\text{sub}}(m) = \frac{|\{t \in T_{\text{oracle}} \mid t \text{ fails on } m\}|}{|T_{\text{oracle}}|}.$$

A lower M_{sub} indicates a narrower fault that is triggered by fewer oracle tests and is therefore harder to detect with a randomly chosen assertion. In addition to the mean, we report the distribution of M_{sub} across all **SUCCESS** mutants, providing rich information about whether a method produces a long tail of very subtle faults.

- (c) **Token Cost:** We report two cost measures, the *raw output tokens* generated by the LLM and *amortized cost per verified fault*. The amortized cost is calculated by normalizing the sum of input, thinking and output tokens by the number of **SUCCESS** mutants:

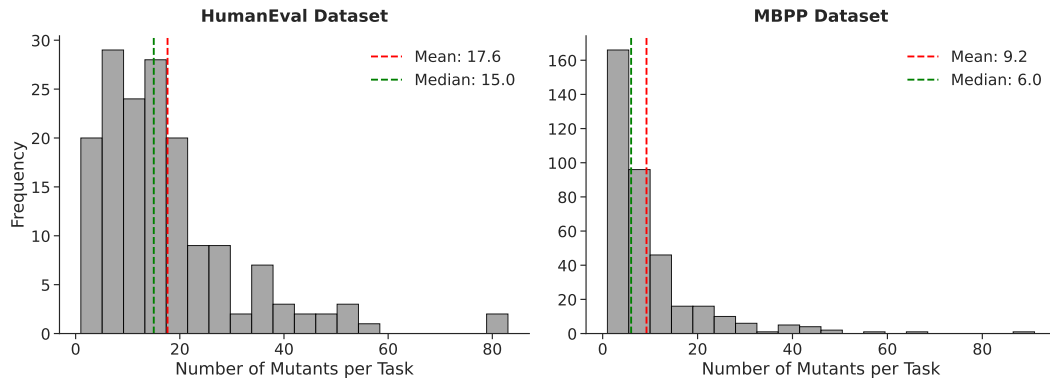
$$M_{\text{cost}} = \frac{\text{total tokens (input + output + thinking)}}{|\{\text{mutants classified as SUCCESS}\}|}.$$

A configuration that uses more tokens per prompt but achieves a higher success rate can still have a lower M_{cost} than a prompt that wastes most of its generations on **INVALID** or **REJECTED** outcomes. For the rule-based baseline mutmut, we do not report token cost, since its computational expense lies in execution time rather than API usage; the comparison with mutmut is therefore limited to M_{eff} and $M_{\text{sub}}(m)$.

4 Results

To maximize reproducibility, we evaluate all LLMs with temperature 0. Because this makes the LLM mostly deterministic, we only generate a single mutant per model-prompt-task combination. In contrast, mutmut’s exhaustive rule-based generation produces a significantly larger mutant count: around 18 mutants per HumanEval task and 9 mutants per MBPP task on average (see Figure 2 for details). Thus, we present the mutmut results separately in Table 3, showing that the vast majority ends up **INVALID** or **REJECTED**.

The LLM-based results in Figure 3 reveal that, as we move from test-blind to test-aware prompting, there is a shift in mutant outcomes from **REJECTED** to **SUCCESS**. For example, AGENT1-TB generates predominantly **REJECTED** mutants: 68% (112) – 87% (143) are executable mutants failing on the base tests for HumanEval and 54% (205) – 78% (295) for MBPP across all LLMs. The proportion of **REJECTED** and **SUCCESS** changes drastically for the test-aware AGENT1-TA. For 164 HumanEval problems, the best-performing model, Gemini 3.1 Pro, generates 144 **SUCCESS** mutants (87.8% **SUCCESS** rate) and only 4 **REJECTED**. Except for GPT 4.1 Mini, every test-aware model outperforms all other models (even smaller ones) in the test-blind scenario in terms of success rate. The ranking of model-prompt combinations regarding success rate is similar for MBPP, establishing a clear trend. This confirms that without visibility of the base tests, LLMs introduce changes that existing assertions already cover. We provide the full results in Table 4.



■ **Figure 2** Mutant counts per task, generated by mutmut. We use the default settings to run mutmut, resulting in varying numbers of mutants per task, depending on the canonical source code.

■ **Table 3** Status distribution of mutants generated by mutmut. Note that the default setting used for mutmut generates multiple mutants per task, making the total mutant count larger than the number of tasks (2841/164 for HumanEval, 3427/378 for MBPP).

Dataset	SUCCESS	UNDETECTED	REJECTED	INVALID	Total
HumanEval	126	179	1473	1063	2841
MBPP	195	197	1519	1516	3427

■ **Table 4** Evaluation of LLM-based mutant generation on 164 HumanEval and 378 MBPP problems under different prompting strategies. For each dataset and model-prompt configuration, we report counts of INVALID (Inv), REJECTED (Rej), SUCCESS (Succ), and UNDETECTED (Und) mutations.

Dataset	Tool	Total	AGENT1-TA				AGENT2-TA				AGENT1-TB				AGENT2-TB			
			Inv	Rej	Succ	Und	Inv	Rej	Succ	Und	Inv	Rej	Succ	Und	Inv	Rej	Succ	Und
HumanEval	gemini-3-flash	164	15	4	117	28	22	12	99	31	2	126	34	2	29	115	11	9
	gemini-3.1-pro	164	7	4	144	9	9	4	130	21	0	143	20	1	8	148	7	1
	gpt-4.1-mini	164	19	45	52	48	23	93	27	21	7	137	9	11	11	129	5	19
	gpt-5.1-codex-mini	164	9	8	133	14	8	7	135	14	5	137	19	3	15	132	13	4
	qwen3-32b	164	46	22	68	28	63	25	55	21	15	112	28	9	9	134	13	8
MBPP	gemini-3-flash	378	28	8	274	68	22	17	225	114	14	232	107	25	44	278	32	24
	gemini-3.1-pro	378	21	2	299	56	25	0	300	53	10	261	87	20	10	322	38	8
	gpt-4.1-mini	378	59	50	147	122	65	175	76	62	13	295	36	34	40	275	22	41
	gpt-5.1-codex-mini	378	19	11	276	72	25	5	270	78	6	261	84	27	22	300	41	15
	qwen3-32b	378	37	29	219	93	91	31	190	66	26	205	108	39	20	300	41	17

We also observe that test-aware prompting produces more UNDETECTED mutants than test-blind (e.g., 9 vs. 1 on HumanEval with AGENT1 for Gemini 3.1 Pro). Given the larger number of REJECTED mutants in the test-blind case, this occurs because test-blind models generate many trivial faults caught by the base tests, leaving fewer to reach the oracle. Also, the number of INVALID mutants depends mostly on the LLM used, with smaller models (Qwen3-32B, Gemini 3 Flash, GPT 4.1 Mini) producing more such errors than larger models (GPT 5.1 Codex Mini, Gemini 3.1 Pro). This is likely due to the smaller models’ limited capacity to understand the source code and task, weakening the reasoning capabilities required to perform a minimal change that remains executable. Generally, we find that the particular prompt has less effect on the mutant status distribution than the model capability.

4.1 Effectiveness of Test-Aware Prompting

Figure 4 compares success rates across the five evaluated models. With mutmut achieving a success rate of only around 4.4% on HumanEval and 5.7% on MBPP (dashed line), this baseline is outperformed by all LLMs, whether prompted in test-aware or test-blind manner, with two exceptions: for AGENT2-TB, Gemini 3.1 Pro has a SUCCESS rate of 4.3% and GPT 4.1 Mini rate is 3.0% on HumanEval. This suggests that even test-blind LLMs are capable of generating higher-value mutants than exhaustive mutmut generation, but test-awareness provides the greatest benefit. In every model, the TA prompt variant outperforms TB. On HumanEval, AGENT1-TA increases the SUCCESS rate for Gemini 3.1 Pro from 12.2% to 87.8% (7.2×) and for GPT 5.1 Codex Mini from 11.6% to 81.1% (7.0×). For AGENT2-TA as well, Gemini 3.1 Pro improves from 4.3% (TB) to 79.3% (TA) and GPT 5.1 Codex Mini improves from 7.9% (TB) to 82.3% (TA). Even for the weakest-performing model, GPT 4.1 Mini, SUCCESS increases from 5.5% to 31.7% (AGENT1) and from 3.0% to 16.5% (AGENT2). On MBPP, the absolute gains are smaller but the trend remains. This consistency across two prompt families and benchmarks for all models shows that test-awareness alone is more crucial for improving mutant generation than the particular prompt phrasing.

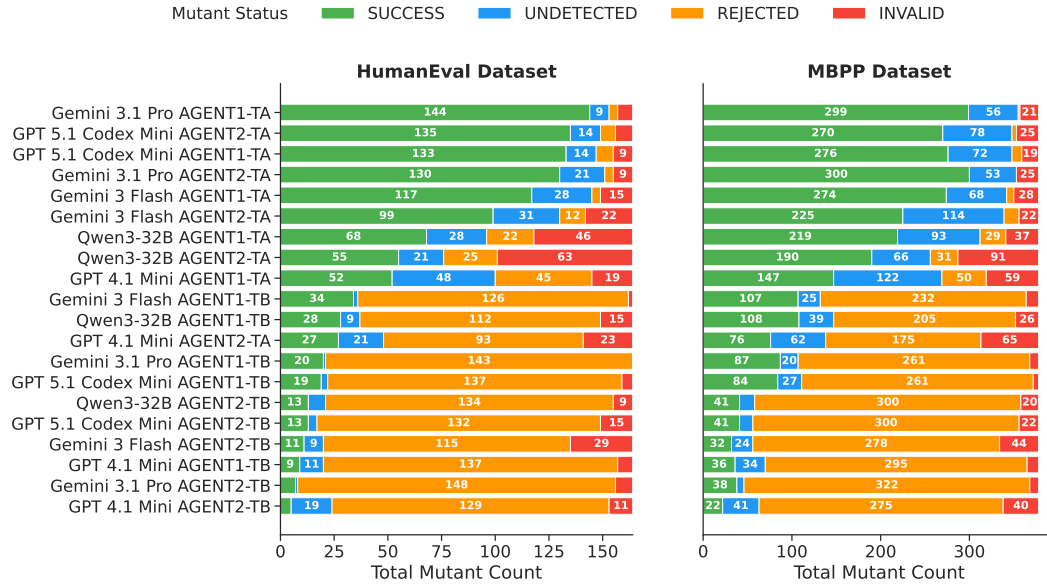


Figure 3 Distribution of mutant classifications across the HumanEval and MBPP datasets.

Comparing across prompts, our results indicate that the free-form reasoning of AGENT1 generally outperforms the structured template of AGENT2. The rigid constraint of AGENT2-TA to explicitly identify two to three blind spots appears to degrade the model’s ability to produce base-test-compliant code, particularly for smaller models. For example, evaluating GPT 4.1 Mini across both prompt variants reveals that AGENT2-TA produces **REJECTED** mutants on 156 tasks (48 in HumanEval, 108 in MBPP) where AGENT1-TA is able to evade the base tests for the same tasks (**SUCCESS** or **UNDETECTED**). This indicates that forcing explicit structural reasoning can hinder the adversarial search space in practice.

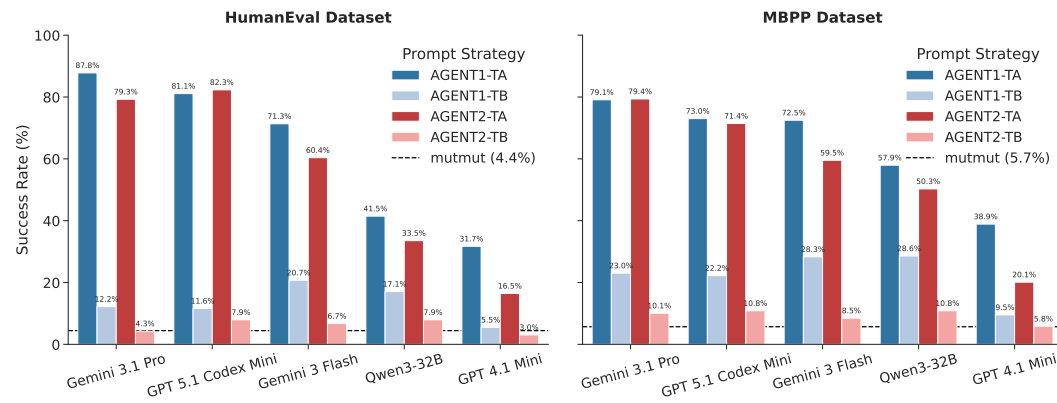


Figure 4 Comparison of mutant success rates across models and prompting strategies. The dashed lines indicate the success rates of the traditional rule-based baseline (mutmut).

4.2 Subtlety and Fault Quality

Figure 5 shows the distribution of oracle kill rates for the generated **SUCCESS** mutants. We choose this visualization to identify whether certain settings produce more subtle bugs, indicated by a smaller number of extended test cases that catch the mutant. We observe that

although test-aware models have a higher number of successful mutants, the median oracle kill rate for test-aware configurations falls between roughly 10-30%, comparable to those produced by test-blind models. While the dense clustering of points in the lower quartile (e.g., see QWEN3-32B) indicates that test-aware prompting is highly capable of generating narrow, subtle edge-cases rather than catastrophic logical breakages, it is not clear from our experiments that LLMs produce significantly more or less subtle bugs than mutmut.

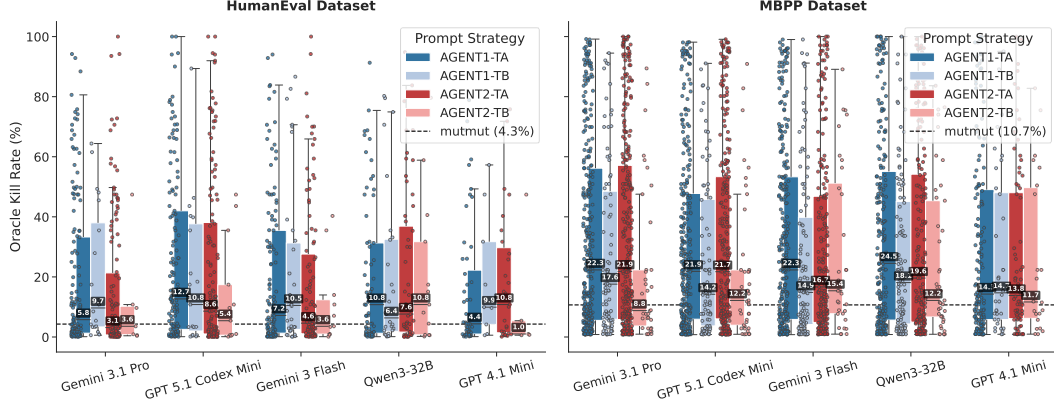


Figure 5 Fault subtlety as measured by the oracle kill rate. A lower kill rate indicates subtler edge-case bugs that are more difficult for test suites to detect. The horizontal lines in each box and the small numbers above show the *median* value. Thus, mutmut (dashed line) is also presented as the *median* (not mean) in this plot. The whiskers represent $1.5\times$ interquartile range (IQR).

To identify the reason for the substantial number of **UNDETECTED** mutants in Figure 3, we manually review each such mutant (9 for HumanEval, 56 for MBPP) produced by the best-performing setting, Gemini 3.1 Pro with the AGENT1-TA prompt. We want to know whether undetected mutants represent failures of the generation pipeline (equivalent mutants) or failures of the extended test oracle (genuine but undetected faults). The classification scheme was derived deductively to cover all common mutation testing failures and successes in a non-overlapping manner. Due to resource constraints, the labeling process was conducted by a single author, and therefore inter-rater agreement was not measured. Nonetheless, this qualitative assessment provides valuable insight into the types of edge-case faults that can bypass even extensive automated oracles. Concretely, we establish a hierarchical classification scheme consisting of four equivalency classes and two non-equivalency classes (see Table 5), where equivalent mutants are programs that behave exactly as the original program on all possible inputs, representing a failed attempt to create a program fault.

The four distinctions among equivalent mutants are: (1) Identical code or cosmetic changes such as removing comments or renaming variables, (2) semantic refactoring, i.e., a change in algorithm procedure, but resulting in the same outputs, (3) out-of-specification faults, for example a mutant that produces incorrect results on negative input values even though the task description explicitly restricts inputs to be positive, and (4) dead code, where modifications only occur in unvisited execution branches. Conversely, non-equivalent mutants introduce real faults that the extended oracle failed to catch, which we again divide into two categories: (5) true oracle gaps, representing realistic bugs completely missed by the tests, such as HumanEval#18 which asks how many times a given substring can be found in the original string, but never tests for an empty substring, and finally (6) ambiguous faults, which are bugs that only manifest under poorly defined task specifications, such as MBPP#87, which asks to merge three dictionaries without defining precedence for duplicate

keys. In summary, **UNDETECTED** mutants of type (1)-(4) represent a failure of the mutant generator, (5) marks a failure in the extended test oracle, and (6) attributes the fault to the underspecified task description.

Out of the 65 **UNDETECTED** mutants, we found that only 9 ($\sim 13.8\%$) are equivalent mutants and a vast majority (49/65, $\sim 75.4\%$) present serious edge-case bugs that slipped through the extended test cases. This shows that even EvalPlus – the more robust version of HumanEval and MBPP – does not provide guaranteed perfect test cases in these instances. The remaining 7 ($\sim 10.8\%$) are classified as ambiguous faults, meaning they exploit underspecified edge cases where neither the task description nor the test suite clearly defines the expected behavior. This manual review confirms that rather than representing a failure of the mutation pipeline, the **UNDETECTED** category highlights the *strength* of test-aware LLMs by showing their ability to act as advanced adversarial testers that even expose flaws in the hidden test oracle, which is significantly stronger than the base tests.

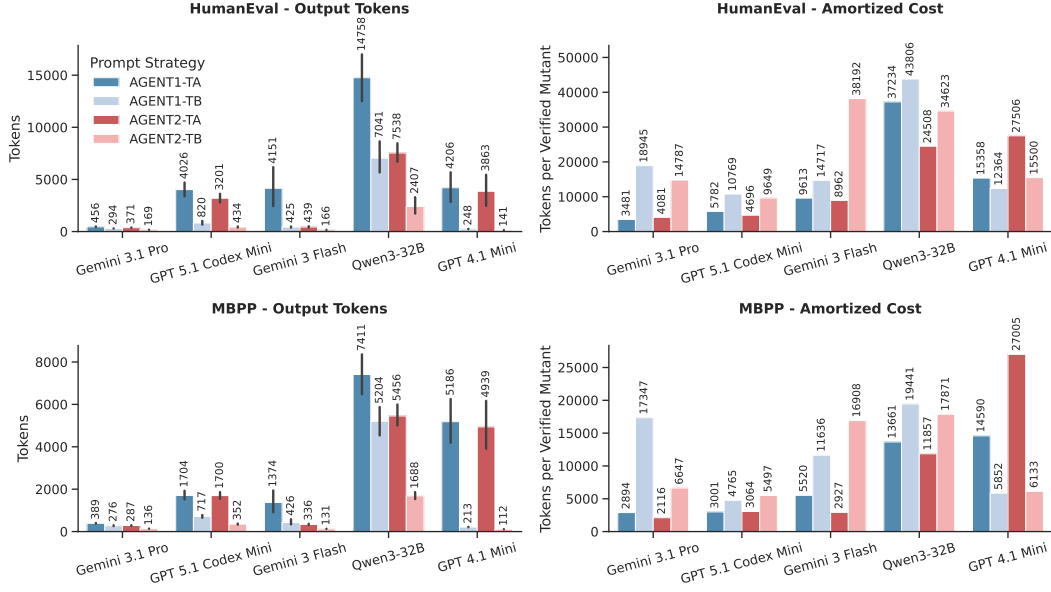
■ **Table 5** Distribution of **UNDETECTED** mutants across equivalent and non-equivalent classes. This qualitative analysis was manually performed on the results from Gemini 3.1 Pro with AGENT1-TA.

Classification	Total	Instances
<i>Equivalent Mutants</i>		
(1) Identical / cosmetic	1	HumanEval #114
(2) Semantic refactoring	5	MBPP #{103, 235, 463, 578, 765}
(3) Out-of-spec fault	3	HumanEval #84, MBPP #{124, 445}
(4) Dead code	0	
<i>Non-Equivalent Mutants</i>		
(5) True oracle gap	49	HumanEval #{16, 18, 36, 125, 132, 153, 162}, MBPP #{9, 12, 58, 61, 77, 96, 123, 129, 130, 138, 139, 166, 168, 239, 245, 285, 294, 308, 410, 415, 421, 428, 479, 557, 566, 573, 577, 631, 635, 637, 730, 734, 737, 751, 755, 760, 772, 787, 793, 800, 804, 807}
(6) Ambiguous fault	7	MBPP #{87, 439, 580, 607, 733, 773, 799}

4.3 Cost Efficiency

Naturally, LLMs are computationally more demanding than a rule-based mutant generation library like mutmut. We argue this price is worth paying for higher-quality results. Among the LLM-based approaches, we perform a fair cost comparison by considering the *amortized* cost, i.e., the average total tokens consumed to produce a single **SUCCESS** mutant. This analysis, shown in Figure 6, reveals that test-aware prompting is more token-efficient per verified mutant than test-blind prompting for all models except GPT 4.1 Mini.

We also report the wall-clock time for running these experiments. For mutmut, the full HumanEval run took 6806.5s and generated 2841 mutants, which is around 2.4s per generated mutant and 54.0s per verified **SUCCESS** mutant. On MBPP, mutmut took 8090.9s for 3427 mutants, around 2.4s per generated mutant and 41.9s per verified mutant. For GPT 5.1 Codex Mini with AGENT1-TA, the total runtime was 29.7s per problem on HumanEval and 16.4s on MBPP. This means that a single LLM-generated mutant is roughly $7\text{--}12\times$ slower than a single mutmut mutant if we compare one generated candidate to one generated candidate. However, the runtime ranking reverses for cost per useful fault. Because test-aware prompting produces a larger proportion of verified faults, GPT 5.1 Codex Mini requires



■ **Figure 6** Distribution of produced output tokens with 95% confidence intervals (left) and amortized computational cost per successful mutant, representing the sum of total tokens (input + thinking + output) used for each mutant, divided by the number of **Success** mutants (right).

only 36.6s per verified mutant on HumanEval and 22.5s on MBPP, compared with 54.0s and 41.9s for mutmut. In other words, LLM-based mutation pays a higher per-candidate premium, but recovers that cost through much higher yield. While LLMs incur higher upfront token and monetary costs, this is offset by the reduction in execution overhead. Executing large suites of trivial mutants is expensive in industrial pipelines; thus, investing compute into high-quality mutants allows for a substantially lower overall cost per verified fault.

4.4 Iterative Prompting

While the single-shot results already show that test-awareness leads to subtle and effective mutant generation, a natural question is whether adding execution feedback can close the remaining gap. The answer is a clear yes: Figure 7 shows results for an iterative variant of our pipeline in which the model follows a simple strategy to retry (up to 2 additional times) when a candidate fails the base tests: If a mutant is **INVALID**, we show the execution error and ask for a syntax fix; if a mutant is **REJECTED**, we include information on which base-test input failed by revealing the mutant output expected output; finally, if a mutant is **UNDETECTED**, we encourage the model to take a different approach to avoid semantic equivalence to the original code. Such iterative prompting narrows the TA–TB gap: Gemini 3.1 Pro AGENT1-TB rises from 12.2% in single-shot to 83.5%, approaching the iterative TA rate of 98.2%. We see gains even on the already high-performing TA prompts (e.g., Gemini 3.1 Pro with AGENT2-TA on HumanEval reaches 163 out of 164 verified faults (99.4%), an increase from 130 in the single-shot setting). While the test-blind prompts use around twice the generation cost in the iterative setting, the test-aware iterative configurations require far fewer retries: Gemini 3.1 Pro averages 1.19–1.24 attempts per problem under TA versus 2.27–2.46 under TB. This reinforces the cost-efficiency advantage of test-awareness. These

results suggest that single-shot test-aware prompting captures the majority of the achievable gain and iteration offers diminishing but still meaningful returns, particularly for converting the remaining 10–15% of problems into verified faults.

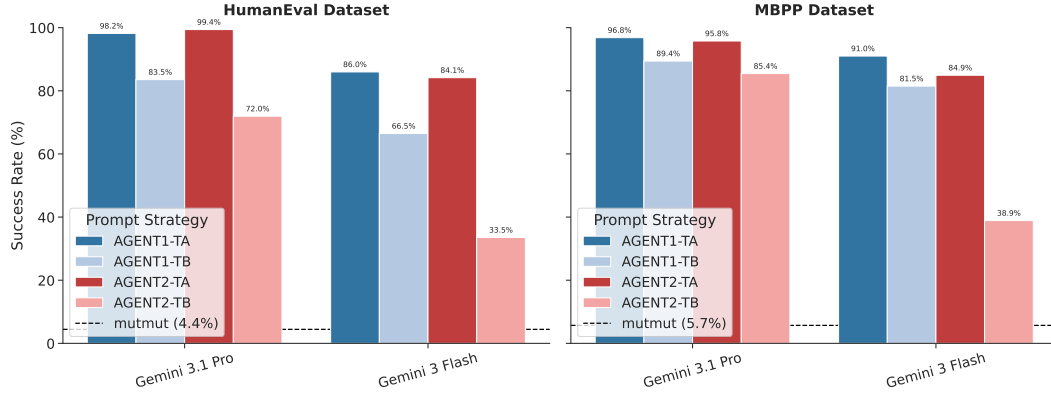


Figure 7 Success rates under iterative prompting. The TA–TB gap narrows compared with the single-shot results in Figure 4, as repeated execution feedback partially compensates for missing test context (see Table 6).

Table 6 Iterative prompting results for Gemini 3.1 Pro and Gemini 3 Flash on HumanEval and MBPP. Unlike the single-shot setting, the pipeline retries until the maximum attempt limit (in our case, 3) is reached. For each configuration, we report total generated mutants (Gen), counts of **INVALID** (Inv), **REJECTED** (Rej), **SUCCESS** (Succ), and **UNDETECTED** (Und) mutants, and the average number of attempts per problem (Att).

Dataset	Model	AGENT1-TA						AGENT2-TA						AGENT1-TB						AGENT2-TB					
		Gen	Inv	Rej	Succ	Und	Att	Gen	Inv	Rej	Succ	Und	Att	Gen	Inv	Rej	Succ	Und	Att	Gen	Inv	Rej	Succ	Und	Att
HumanEval	gemini-3.1-pro	195	0	0	161	3	1.19	203	0	0	163	1	1.24	373	0	9	137	18	2.27	404	1	18	118	27	2.46
	gemini-3-flash	247	0	0	141	23	1.51	270	1	0	138	25	1.65	379	0	16	109	39	2.31	445	1	37	55	71	2.71
MBPP	gemini-3.1-pro	469	0	0	366	12	1.24	490	0	0	362	16	1.30	735	0	4	338	36	1.94	846	0	5	323	50	2.24
	gemini-3-flash	533	1	0	344	33	1.41	612	5	1	321	51	1.62	773	0	13	308	57	2.04	1011	3	46	147	182	2.67

5 Discussion

Our experiments demonstrate that providing an LLM with test context pushes it to act as an adversarial agent that targets semantic blind spots rather than blindly generating equivalent faults. We further analyze these findings by examining the trade-offs between computational cost and human effort required to find a verified fault, why test-awareness works even in a single non-iterative prompt, how the results improve with an iterative feedback loop, and the limitations of deploying this approach in a real development environment.

Human Time vs. Compute Time Trade-off. Generating a few “smart” mutants is more valuable than a large number of low-quality (redundant, equivalent, trivial) mutants. Although the computational cost in generating mutants using mutmut is far lower than LLM-based mutation testing, the human effort required to review these mutants often surpasses the computational efficiency. Petrović et al. [19] perform a cost-benefit analysis of mutation testing in industrial software development applications. They argue that the goal of mutation testing is to provide developers with “productive” mutants (that indicate genuine bugs) rather than achieving mutation adequacy by strengthening the test suite against all “unproductive”

redundant mutants. This supports the intuition behind our *test-aware* approach that a mutant generator becomes more useful when it produces fewer low-value candidates and more mutants that expose meaningful testing gaps. Our results fit this picture well. Test-aware LLM mutation does not win by being cheaper per generated candidate; it wins when cost is measured per useful, verified fault. In this regime, “smart” mutants become practically attractive because less human effort is required in the process.

Model Selection. The choice of LLM is an important practical consideration, because the effectiveness of our approach depends on it. For example, one test-aware instance (GPT 4.1 Mini AGENT2-TA) underperforms compared to the test-blind alternative (AGENT1-TB with Gemini 3 Flash and Qwen3-32B). Besides success rate, the fault subtlety and token cost are also affected. Our results suggest practical guidelines for deploying test-aware mutation: When the computational budget is strict, smaller models like GPT 5.1 Codex Mini save token costs. However, to maximize the discovery of subtle edge-case faults, frontier models like Gemini 3.1 Pro are recommended despite the higher API costs.

Why Single-Prompt Test-Awareness Helps. The most clear finding in this study is that simply providing the base tests in the prompt only produces more effective, subtle mutants over test-blind generation. This improvement is visible not only in overall success rates, but also in the status distribution: Moving from test-blind to test-aware prompting shifts a substantial portion of outputs from **REJECTED** to **SUCCESS**. The interpretation for such a trend is that once the model can see which behaviors are already checked, it becomes better at avoiding trivial mutations that are immediately killed by the base suite. In that sense, the main value of test-awareness is not only that it helps the model generate bugs, but also that it helps it generate bugs specifically aligned with existing test gaps.

The Dependency on Base Test Quality. Our results show that even few test cases can provide useful context for LLM-based mutant generation. In EvalPlus, where each problem has only few (7.7 tests on average per HumanEval problem and 3 for MBPP) base tests, access to those tests still substantially improves effectiveness over the test-blind setting. At the same time, this also highlights a limitation of the approach: An LLM can only exploit shortcomings that are actually expressed in the provided tests. Because we did not systematically vary base-test quality or coverage, this study does not establish exactly how performance changes as the initial suite becomes stronger or weaker. That dependency remains an important direction for future work.

No Oracle in the Real World. Our evaluation relies on the extended EvalPlus tests as an oracle for determining whether a mutant that survives the base tests is actually non-equivalent. This is useful for controlled benchmarking, but it is not representative of most real-world development settings, where no such oracle is available. In practice, a surviving mutant would not automatically come with a ground-truth label; instead, it would signal that the current test suite may be missing a behavior worth checking. The main value of our setup is therefore methodological: It lets us verify that adding tests to the prompt helps LLMs find faults that the original suite misses. In real workflows, these mutants should be interpreted primarily as concrete suggestions for stronger tests and developer inspection rather than automatically validated defects. Identifying high-quality mutants makes it generally easy to specify an additional high-quality unit test to strengthen the test suite.

Practical Implications. Our findings indicate that test-aware LLMs can aid as review assistants in continuous integration (CI), directly improving software engineering workflows. During pull requests, models could analyze modified code and tests to generate targeted mutants, providing developers actionable suggestions for stronger unit tests.

6 Threats to Validity

We identify and address the following potential threats to validity in our work:

- (a) **Internal Validity: Sparse Base Tests.** Bypassing only a handful of human-written base tests may appear to be a low bar for modern LLMs. However, we view this circumstance as a plausible representation of real-world codebases, where developers often have only basic assertions in place, before moving on to subsequent tasks. The aim of this study is not to show that LLMs are better than humans at generating mutants, but rather that they can achieve what a human developer with abundant time would do: uncover gaps in a test suite to make it stronger.
- (b) **Construct Validity: Lack of Human Review.** Our evaluation pipeline is fully autonomous, which raises the question how reliable the mutant classification is. We highlight our careful experimental setup: The four classes are designed to be mutually exclusive and cover all distinct scenarios, with **INVALID**, **REJECTED**, and **SUCCESS** being clearly defined and automatically verifiable. Only the last category leaves room for error, considering that **UNDETECTED** mutants can either be fully valid code or contain a bug sneaky enough to go unnoticed by the strong extended test suite. To mitigate this threat, we perform a qualitative analysis, see Section 4.2. We find that most of such undetected cases in the best model-prompt combination are in fact serious faults, revealing that if we had a perfect oracle, this would only further improve the success rates. While EvalPlus serves as a robust automated oracle, we emphasize that it is a proxy rather than a definitive ground-truth measure of mutant quality in industrial settings.
- (c) **External Validity: Benchmark Limitations.** Our study evaluates self-contained Python benchmarks rather than complex production systems. We consider this sandbox setting a crucial first step, deliberately using EvalPlus as a controlled pipeline for reliable and transparent results. However, industrial codebases introduce technical complexities such as larger test suites, stateful functions, and multi-file dependencies. This could exceed an LLM’s context window and dilute its focus. Scaling to production-level datasets like Defects4J [11] may require iterative prompting to navigate stateful executions and retrieval-augmented generation (RAG) [13] to fetch relevant unit tests.
- (d) **External Validity: Model Constraints.** LLM performance inherently varies across models and provider-side policies. We mitigate this threat by evaluating five diverse LLMs, confirming that the benefits of test-aware prompting generally hold across different models. A discussion on the practical trade-offs of model selection is provided in Section 5.

7 Conclusion & Future Work

This paper explores whether a test-aware prompting strategy can turn LLMs into targeted mutant generators rather than generic bug injectors. Across HumanEval and MBPP, the answer is yes: Giving the model access to the existing tests substantially improves verified fault generation over both matched test-blind prompting and the rule-based baseline mutmut. Two broader findings stand out. First, the largest gain from test-awareness is not a universal reduction in oracle kill rate, but a significant increase in the number of verified, useful

mutants produced at comparable subtlety. Second, although test-blind prompting can reduce tokens per generation due to less input and thinking tokens, test-aware prompting is more cost-effective per verified mutant because it wastes less computation on outputs that never become useful faults. Overall, our results suggest that test-aware prompting is a practical and lightweight alternative to both exhaustive rule-based mutation and heavier iterative multi-agent systems. More broadly, they indicate that when existing tests are available, LLMs can use them not only as a constraint, but also as a signal for where the test suite is still weak. This finding could substantially improve automated mutant generation, and ultimately lead to stronger test suites.

Our empirical findings pave the way for future work. For example, researchers with access to well-tested industrial code repositories could evaluate our pipeline by splitting their test suite into a handful of base tests and leaving the rest as oracle tests, pretending these are not yet part of the test suite. We hypothesize that this would replicate our finding that test-awareness leads to an increase in success rate. Additionally, as LLMs scale, further research is needed to quantify the computational and environmental trade-offs of this approach. This includes comparing the energy consumption and CO₂ emissions of local versus cloud-hosted inference, and analyzing whether next-generation models with extended reasoning capabilities can further optimize the amortized cost per fault.

Data Availability Statement

We make all code and data publicly available as open artifacts. Our code repository is available at <https://github.com/codingWhale13/test-aware-mutants>. The generated data is available at <https://zenodo.org/records/20195774> under a CC-BY 4.0 license. To reproduce the plots in this paper, rename the unzipped folder to “result”, place it at the root of the code repository, and follow the instructions in the repository’s README file.

References

- 1 Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 185–196, 2024. doi:10.1145/3663529.3663839.
- 2 Juan Altmayer Pizzorno and Emery D. Berger. CoverUp: Effective high coverage test generation for Python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025. doi:10.1145/3729398.
- 3 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. doi:10.48550/arXiv.2108.07732.
- 4 Pengyu Chang, Yixiong Fang, Silin Chen, Yuling Shi, Beijun Shen, and Xiaodong Gu. Test vs mutant: Adversarial LLM agents for robust unit test generation. *arXiv preprint arXiv:2602.08146*, 2026. doi:10.48550/arXiv.2602.08146.
- 5 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. doi:10.48550/arXiv.2107.03374.
- 6 Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171, 2024. doi:10.1016/j.infsof.2024.107468.

- 7 Renzo Degiovanni and Mike Papadakis. μ BERT: Mutation testing using pre-trained language models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 160–169, 2022. doi:10.1109/ICSTW553395.2022.00039.
- 8 Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020. doi:10.18653/v1/2020.findings-emnlp.139.
- 9 Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. Challenging bug prediction and repair models with synthetic bugs. In *2025 IEEE International Conference on Source Code Analysis & Manipulation (SCAM)*. IEEE, 2025. doi:10.1109/SCAM67354.2025.00021.
- 10 Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2010. doi:10.1109/TSE.2010.62.
- 11 René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014. doi:10.1145/2610384.2628055.
- 12 René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014. doi:10.1145/2635868.2635929.
- 13 Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.
- 14 Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572, 2023. doi:10.52202/075280-0943.
- 15 Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1), 2013. doi:10.1109/TSE.2013.44.
- 16 Sadnan Nafis, Abdullah Al Walid, and Amatus Subhan Anuja. *An LLM-based framework for automated Python test generation and mutation testing evaluation*. PhD thesis, Brac University, 2025. URL: <http://hdl.handle.net/10361/26750>.
- 17 Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*, volume 112. Elsevier, 2019. doi:10.1016/bs.adcom.2018.03.015.
- 18 Goran Petrović and Marko Ivanković. State of mutation testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018. doi:10.1145/3183519.3183521.
- 19 Goran Petrović, Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018. doi:10.1109/ICSTW.2018.00027.
- 20 Dimitrios Rontogiannis, Maxime Peyrard, Nicolas Baldwin, Martin Josifoski, Robert West, and Dimitrios Gunopulos. Interactive evaluation of large language models for multi-requirement software engineering tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 32843–32850, 2026. doi:10.1609/aaai.v40i39.40564.

- 21 Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi:10.1145/3643769.
- 22 Philipp Straubinger, Marvin Kreis, Stephan Lukasczyk, and Gordon Fraser. Mutation testing via iterative large language model-driven scientific debugging. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2025. doi:10.1109/ICSTW64639.2025.10962485.
- 23 Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, et al. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026. doi:10.48550/arXiv.2601.01426.
- 24 Alex Thillen, Niels Mündler, Veselin Raychev, and Martin Vechev. CodeTaste: Can LLMs generate human-level code refactorings? *arXiv preprint arXiv:2603.04177*, 2026. doi:10.48550/arXiv.2603.04177.
- 25 Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022. doi:10.1145/3551349.3556949.
- 26 Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi:10.1145/3650212.3680395.
- 27 Frank Tip, Jonathan Bell, and Max Schäfer. LLMorpheus: Mutation testing using large language models. *IEEE Transactions on Software Engineering*, 2025. doi:10.1109/TSE.2025.3562025.
- 28 Hung Tran, Langston Nashold, Rayan Krishnan, Antoine Bigeard, and Alex Gu. Vibe code bench: Evaluating AI models on end-to-end web application development. *arXiv preprint arXiv:2603.04601*, 2026. doi:10.48550/arXiv.2603.04601.
- 29 Bo Wang, Mingda Chen, Ming Deng, Youfang Lin, Mark Harman, Mike Papadakis, and Jie M. Zhang. A comprehensive study on large language models for mutation testing. *ACM Transactions on Software Engineering and Methodology*, 2026. doi:10.1145/3805038.
- 30 Guancheng Wang, Qinghua Xu, Lionel Briand, and Kui Liu. Mutation-guided unit test generation with a large language model. *IEEE Transactions on Software Engineering*, 2026. doi:10.1109/TSE.2026.3682975.