

SpecMutate: Coverage-Guided Specification Diagnosis and Repair for Property-Based Tests

Aditya Singh • IIT Kharagpur

Apart Research Secure Program Synthesis Hackathon 2026 • Track 2: Specification Validation

Abstract

SpecMutate achieves 15/15 diagnostic accuracy and 10/10 repair convergence on SpecMutate-15, a balanced 15-task benchmark spanning underconstrained, overconstrained, and correct Hypothesis specification classes. As large language models generate increasing volumes of test specifications, unverified specs become a silent failure mode: underconstrained specs allow buggy implementations to pass verification, while overconstrained specs reject correct implementations. SpecMutate diagnoses whether a Hypothesis specification is underconstrained, overconstrained, or correct using a 3-tier cascade of deterministic gates and four independent signals, then repairs broken specs through a feedback-guided LLM loop grounded in AST-level mutation evidence, coverage delta quantification, and concrete counterexample inputs. Ablation reveals the cascade is load-bearing asymmetrically: removing the primary overconstrained gate drops accuracy by 33.3%, while removing the underconstrained guard costs zero accuracy because the weighted fusion formula independently handles those cases. SpecMutate is the first Python-native, Hypothesis-native system to close the diagnosis–repair loop with AST-level constraint localization.

1. Introduction

Property-based testing (PBT) with Python’s Hypothesis framework allows developers to express executable specifications as universally-quantified properties verified against thousands of randomly generated inputs. As large language models generate increasing volumes of code and test specifications, the trust bottleneck shifts from writing implementations to verifying that the specifications themselves are correct.

Broken specifications fail in one of two directions. An underconstrained specification is too weak: it accepts incorrect implementations, creating false confidence that code is correct when it is not. For AI-assisted biosecurity screening tools — classifying dual-use biology research, detecting hazardous compound synthesis, or flagging dangerous protocol requests — an underconstrained spec for a critical classifier could allow a buggy implementation to pass all tests while misclassifying specific compound classes or request patterns. An overconstrained specification is too strict: it rejects correct implementations, producing brittle tests that block valid code and are eventually deleted by frustrated developers. Both failure modes are dangerous; neither produces a visible error signal.

Mutation testing [1] has long been used to evaluate test suite quality by measuring how many synthetic bugs a test suite detects. SpecMutate inverts this: it mutates the specification to diagnose which constraint is broken, scoring each mutation by branch coverage delta on the correct implementation. Coverage-guided exploration, pioneered in fuzz testing by AFL [2], is applied here to specification space rather than input space: instead of measuring which inputs expose new branches, we measure which spec mutations allow new branches to be covered.

Our main contributions are:

- A **3-tier diagnostic cascade** achieving 15/15 accuracy on a balanced 3-class benchmark by combining two deterministic gates with four-signal weighted fusion. Ablation reveals the architecture is load-bearing asymmetrically: Gate 1 (overconstrained detector) is critical; Gate 2 (underconstrained guard) is a fast-path optimisation that the fusion formula handles redundantly.
- A **coverage-guided mutation engine** that applies three AST-level operators (FlipComparison, RemovePrecondition, RemovePostcondition) scored by branch coverage delta, localizing the specific bad AST constraint rather than merely detecting that a spec is wrong.
- A **feedback-guided repair loop** that provides each LLM repair call with the specific bad AST node, its coverage delta, and a concrete counterexample input — achieving single-iteration convergence in 9 of 10 repair cases and 100% convergence within 3 iterations.
- SpecMutate-15: a purpose-built 15-task benchmark with balanced 5/5/5 distribution across verdict classes, hand-crafted to represent realistic LLM specification failure modes.

2. Related Work

Mutation-based testing [1] evaluates test suite quality by mutating implementations and measuring detection rates. SpecMutate inverts this: it mutates the *spec* and uses coverage delta on a correct implementation to diagnose which constraint is broken.

Property-based testing with QuickCheck [3] and Hypothesis [7] enables property specification over random input distributions. EvoSuite [5] and Pynguin [6] generate test suites from scratch; SpecMutate instead diagnoses and repairs existing specifications.

Counterexample-Guided Inductive Synthesis (CEGIS) [4] provides the conceptual model: generate a candidate, verify it, extract a counterexample on failure, use the counterexample to constrain the next candidate. The SpecMutate repair loop is inspired by this structure, using an LLM as the synthesizer and a Hypothesis runner as the verifier. Unlike formal CEGIS, the loop does not formally guarantee the synthesizer excludes the counterexample from its search space; it provides it as context and relies on the LLM to incorporate it.

CoverUp [8] uses coverage-guided LLM test generation for Python, demonstrating that coverage feedback materially improves test quality. SpecMutate applies coverage guidance at the specification level: instead of generating new tests to cover lines, it identifies which spec constraint is preventing coverage of the correct implementation’s branches.

VeriAct [9] is the closest prior work. It implements a diagnosis-and-repair loop for JML/Java specifications using OpenJML symbolic verification. VeriAct identifies that a spec is bad but cannot localize which specific constraint is wrong — its repair prompt is generic rather than grounded in a specific AST node. VeriAct also does not detect overconstrained specifications; it only addresses underconstrained ones. SpecMutate establishes the Python/Hypothesis equivalent with constraint localization and bidirectional diagnosis.

CodeSpecBench [10] confirms that underconstrained specifications dominate LLM-generated outputs, motivating our focus on both under- and over-constrained cases as independent failure modes.

3. Methods

3.1 Problem Formulation

Given a Python function f with reference implementation $f_{\square_e}f$, a set of known-buggy implementations B , and a Hypothesis property-based test specification S , determine the label of S :

- Underconstrained: $\exists f' \in B$ such that $S(f')$ passes — the spec is too weak to reject a known-buggy implementation.
- Overconstrained: $S(f_{\square_e}f)$ fails — the spec rejects the correct implementation.
- Correct: $f_{\square_e}f$ passes S and S rejects $\geq 70\%$ of B .

If the label is not correct, produce a repaired specification S^* satisfying the correctness criteria. All Hypothesis tests run in subprocess isolation using `####PLACEHOLDER####` string substitution rather than f-strings, because Python source code containing curly braces corrupts string formatting.

3.2 Benchmark: SpecMutate-15

SpecMutate-15 is a 15-task benchmark with a balanced 5/5/5 split across verdict classes. The balanced distribution ensures precision and recall are comparable — a biased classifier gains no advantage from distribution asymmetry.

Underconstrained tasks (T01–T05): planted specs with weak or missing postconditions. T01 (`merge_sorted_lists`) checks only `assert len(result) == len(a)+len(b)` without verifying sortedness, allowing any permutation to pass. T02 (`binary_search`) only bounds the return value without checking it points to the target.

Overconstrained tasks (T06–T10): planted specs with overly restrictive preconditions or postconditions. T06 (`factorial`) uses `assume(n >= 1)`, excluding the valid case `factorial(0)=1`. T09 (`clamp`) uses `assume(lo <= x <= hi); assert clamp(x, lo, hi) == x`, restricting testing to the identity region only. No buggy implementations are needed — the correct implementation itself reveals the fault.

Correct tasks (T11–T15): well-formed specifications that should not be repaired. These measure the system’s false positive rate.

Oracle disclosure. The reference implementation is used at inference time for Gate 1, Signal S4, the mutation engine, and repair convergence. This is an inherent requirement of specification evaluation: overconstrained

detection requires knowing what correct behaviour looks like. Ground-truth labels, correct specs, and repair actions are used only for post-hoc evaluation.

3.3 The Four Diagnostic Signals

S1 — Completeness (weight 0.35): An LLM generates 5 adversarial implementations designed to exploit a weak spec. S1 = fraction rejected. Low S1 indicates the spec permits buggy behavior to pass.

S2 — Discrimination (weight 0.35): All available implementations are run against the spec. S2 = fraction of pairwise combinations where one passes and one fails. S2=0.0 is the most informative single data point: the spec cannot distinguish any implementation pair. Note: with 5 buggy + 1 correct implementation, the theoretical maximum S2 is $5/15 \approx 0.33$ if only the correct implementation passes. Higher observed S2 values (e.g., 0.6 for correct tasks) indicate some adversarial implementations also pass the spec — expected, as the 70% rejection threshold permits this.

S3 — CrossHair symbolic refutation (weight 0.20): CrossHair runs with `--analysis_kind=hypothesis` on the correct implementation and spec. Health-check gated: S3=0.5 (neutral) if unavailable, S3=1.0 if a counterexample is found. Note: the current implementation adds S3 to the underconstrained signal ($+0.20 \cdot S3$). Since S3=1.0 represents overconstrained evidence, it should semantically subtract from the underconstrained signal. With S3=0.5 constant across all benchmark tasks, the sign does not affect current results, but represents a latent bug that would produce incorrect diagnoses if CrossHair were to return S3=1.0.

S4 — Stability (weight 0.10): Three independent specs are generated at temperature=0.7 with caching disabled. S4=1.0 if all agree on pass/fail for the reference implementation; S4=0.0 if there is a split. Low stability indicates an ambiguous task description.

The weighted fusion formula:

$$S_{\text{under}} = 0.35 \cdot (1-S1) + 0.35 \cdot (1-S2) + 0.20 \cdot S3 + 0.10 \cdot (1-S4)$$

3.4 The 3-Tier Diagnostic Cascade

Weighted fusion alone cannot resolve the central ambiguity: both underconstrained specs (all implementations pass, S2=0) and overconstrained specs (even the correct implementation fails, S2=0) produce identical aggregate signal profiles. Deterministic gates fire before fusion to resolve this.

Gate	Condition	Verdict	Tasks Resolved
Gate 1	<code>correct_impl_passes == False</code>	Overconstrained (deterministic)	T06–T10 (5 tasks)
Gate 1.5	<code>RemovePrecondition</code> has max <code>coverage_delta > 0</code>	Overconstrained (mutation-based)	Not triggered — defence-in-depth
Gate 2	<code>S2 == 0.0 AND S1 < 0.9</code>	Underconstrained (signal-based)	T01–T05 (5 tasks)*
Fusion	<code>score < 0.45 → correct; score > 0.65 → underconstrained</code>	Correct	T11–T15 (5 tasks)

Table 1: 3-tier diagnostic cascade. *Gate 2 is a fast-path optimisation: T01–T05 have S1=0.0, S2=0.0, giving fusion score 0.80 > 0.65, so the fusion formula classifies them correctly even without Gate 2 (ablation: 0% accuracy drop).

3.5 Coverage-Guided Mutation Engine

Three AST-level mutation operators are applied to the spec and scored by coverage delta — the number of branches in the reference implementation newly covered after mutation, measured via `coverage.py` in a subprocess. The mutation with the highest delta is the bad constraint.

- **FlipComparison:** flips comparison operators in assert statements ($= \Leftrightarrow !=$, $< \Leftrightarrow \leq$, $> \Leftrightarrow \geq$).
- **RemovePrecondition:** removes `assume()` calls or relaxes `min_value/min_size` strategy constraints to 0.
- **RemovePostcondition:** removes individual assert statements.

Example (T09, `clamp`): the planted spec `assume(lo ≤ x ≤ hi)` restricts inputs to the identity region. `RemovePrecondition` on this `assume()` yields `coverage_delta=4` (exposes clamp-down and clamp-up branches). The mutation engine identifies this as the bad constraint and passes it to both the diagnosis cascade (Gate 1.5) and the repair prompt.

3.6 Feedback-Guided Repair Loop

For each non-correct diagnosis, the repair loop runs up to 3 LLM iterations at temperature=0.2. Each iteration constructs a grounded prompt containing: (1) the specific bad AST node identified by the mutation engine; (2) the mutation operator and coverage delta; (3) a concrete counterexample input from the Hypothesis runner; (4) the correct and buggy implementation outputs on that input; (5) current signal scores S1, S2, S4.

Convergence criteria: (a) the repaired spec passes the reference implementation, AND (b) the repaired spec rejects at least 70% of available buggy implementations. The 70% threshold accommodates implementations so close to correct that no reasonable spec can distinguish them without oracle-level postconditions. For overconstrained tasks (T06–T10), where no buggy implementations exist by design, criterion (a) alone suffices — this means convergence for those tasks does not verify the repaired spec is not now underconstrained, a known limitation.

The repair loop uses LLM response caching (use_cache=True). Benchmark results are produced from cached LLM calls; fresh runs will produce equivalent but not bit-identical results.

4. Results

4.1 Headline Results

15/15	100%	1.1	10/10
Diagnostic Accuracy	Repair Convergence	Avg. Repair Iterations	Repairs Converged

4.2 Confusion Matrix

Predicted \ Actual	Underconstrained	Overconstrained	Correct	Precision
Underconstrained	5 ✓ (T01–T05)	0	0	100%
Overconstrained	0	5 ✓ (T06–T10)	0	100%
Correct	0	0	5 ✓ (T11–T15)	100%
Recall	100%	100%	100%	—

Table 2: Confusion matrix: 15/15 correct classifications. Zero misclassifications across all three verdict classes.

4.3 Per-Task Signal Profile

ID	Function	Label	S1	S2	S3	S4	Score	Path	Repair
T01	merge_sorted_lists	under	0.0	0.0	0.5	1.0	1.0	Gate 2	1 iter
T02	binary_search	under	0.0	0.0	0.5	1.0	1.0	Gate 2	1 iter
T03	remove_duplicates	under	0.0	0.0	0.5	1.0	1.0	Gate 2	1 iter
T04	rotate_list	under	0.0	0.0	0.5	0.0	1.0	Gate 2	1 iter
T05	flatten_nested	under	0.0	0.0	0.5	1.0	1.0	Gate 2	1 iter
T06	factorial	over	0.0	0.0	0.5	1.0	0.0	Gate 1	1 iter
T07	gcd	over	0.0	0.0	0.5	1.0	0.0	Gate 1	1 iter
T08	is_palindrome	over	1.0	0.0	0.5	1.0	0.0	Gate 1	2 iter
T09	clamp	over	0.6	0.6	0.5	1.0	0.0	Gate 1	1 iter
T10	string_palindrome_check	over	0.4	0.0	0.5	1.0	0.0	Gate 1	1 iter
T11	sum_list	correct	1.0	0.6	0.5	1.0	0.24	Fusion	—
T12	max_of_list	correct	1.0	0.6	0.5	1.0	0.24	Fusion	—
T13	has_duplicates	correct	1.0	0.6	0.5	1.0	0.24	Fusion	—
T14	is_sorted	correct	1.0	0.6	0.5	1.0	0.24	Fusion	—

ID	Function	Label	S1	S2	S3	S4	Score	Path	Repair
T15	absolute_value	correct	1.0	0.4	0.5	1.0	0.31	Fusion	—

Table 3: Per-task signal values from `benchmark_results.json`. Score = fusion value (0.0 = gate-classified deterministically; 1.0 = Gate 2 definitive). S3=0.5 constant (CrossHair unavailable in test environment).

4.4 Ablation Study

Configuration	Accuracy	Drop	Notes
Full system (3-tier cascade)	15/15 (100%)	—	Baseline
Remove Gate 1	10/15 (66.7%)	−33.3%	All 5 overconstrained tasks lost (T06–T10)
Remove Gate 2	15/15 (100%)	0%	Fusion score 0.80–0.90 > 0.65 threshold for T01–T05*
Remove both gates	5/15 (33.3%)	−66.7%	Only correct-class tasks survive
S3 (CrossHair) only	—	—	S3=0.5 constant; zero discriminative power
S4 (Stability) only	~8/15	~−47%	S4=1.0 for 14/15 tasks; near-constant

Table 4: Ablation study (analytically derived from gate architecture). *Gate 2 removal: T01–T05 have S1=0.0, S2=0.0, S3=0.5, yielding fusion score $0.35+0.35+0.10+0=0.80$ (S4=1) or 0.90 (S4=0), both above the 0.65 unconstrained threshold. Gate 1 is the load-bearing component; Gate 2 is a fast-path optimisation.

4.5 Repair Case Studies

T09 — clamp (1-iteration repair). Planted spec: `assume(lo ≤ x ≤ hi); assert clamp(x, lo, hi) == x`. Restricts testing to the identity region, leaving clamp-down and clamp-up branches unreachable. Mutation engine: RemovePrecondition on `assume(lo ≤ x ≤ hi)` yields `coverage_delta=4`. Gate 1 fires (correct implementation fails on $x < lo$). Repaired spec: `assume(lo ≤ hi); result=clamp(x, lo, hi); assert lo ≤ result ≤ hi; if x<lo: assert result==lo; elif x>hi: assert result==hi; else: assert result==x`. Converged in 1 iteration.

T08 — is_palindrome (2-iteration repair). Planted spec restricts inputs to lowercase alpha and asserts `is_palindrome(s) is True` always. Iteration 1 counterexample `s='aab'`: repair produces `assert is_palindrome(s) == (s == s[::-1])`, which fails for case-insensitive palindromes. Iteration 2 counterexample `s='0:'`: repair produces `cleaned=''.join(c.lower() for c in s if c.isalnum())`; `assert is_palindrome(s) == (cleaned == cleaned[::-1])`. Converged in 2 iterations.

Two-tier repair disclosure. Underconstrained repairs (T01–T05) require the repaired spec to both pass the reference implementation AND reject ≥70% of buggy implementations. Overconstrained repairs (T06–T10) require only that the reference implementation passes — no buggy implementations exist for overconstrained tasks by design. This means convergence for T06–T10 does not verify the repaired spec is not now underconstrained.

5. Discussion and Limitations

5.1 Architectural Implications

The ablation result that Gate 1 removal costs 33.3% while Gate 2 removal costs 0% is the most architecturally significant finding. The cascade is load-bearing in an asymmetric, non-obvious way: the overconstrained detector is entirely deterministic (run the correct implementation, observe pass/fail), while the underconstrained detector is actually handled redundantly by both Gate 2 and the fusion formula. This asymmetry reflects a fundamental property of the problem: overconstrained detection requires a known-correct oracle, while underconstrained detection can be approximated by signal aggregation alone.

The grounded repair prompt design is responsible for the 9/10 single-iteration convergence rate. Generic repair prompts (“fix this spec”) produce generic, often incorrect repairs. Providing the LLM with the specific bad AST node, its coverage delta, and a concrete counterexample directs repair to the right constraint on the first attempt in most cases.

5.2 Limitations

- **100% accuracy on a self-constructed benchmark.** The 15-task benchmark was designed by the same author as the system. Signal values have clean profiles because the tasks were designed to produce them. Statistical significance at 95% confidence for 3-class classification requires 100+ tasks with independent labeling.
- **S3 (CrossHair) inactive.** $S3=0.5$ constant for all 15 tasks; zero discriminative power. Additionally, the code adds S3 to the underconstrained signal when semantically it should subtract it ($S3=1.0$ is overconstrained evidence). With $S3=0.5$ constant this does not affect current results, but will produce incorrect diagnoses if CrossHair becomes functional.
- **S4 (Stability) near-constant.** $S4=1.0$ for 14/15 tasks. LLMs generate consistent specs for well-defined functions. S4 contributes minimal discriminative signal in the current benchmark.
- **Oracle dependency.** The reference implementation is used at inference time. This is an inherent requirement — detecting overconstrained specs requires knowing what correct behavior looks like — but limits applicability to settings where a trusted reference implementation exists.
- **Overconstrained repair validation gap.** T06–T10 repairs are validated only by confirming the correct implementation passes. There is no check that the repaired spec is not now underconstrained, since no buggy implementations are available for these tasks.
- **LLM caching.** The repair loop uses `use_cache=True`. Benchmark results were produced from cached calls; fresh runs produce equivalent but not bit-identical results.

5.3 Future Work

Month 1: Scale to 100+ tasks using CodeSpecBench with independent labeling, establishing statistical significance. Month 2: Fix CrossHair integration and the S3 sign direction in the fusion formula; diversify S4 test implementation sets. Month 3: Add a coverage-based precondition detector to address the overconstrained repair validation gap. Month 4: Evaluate on real GitHub Python repositories with known bugs, replacing the hand-authored benchmark with production specifications.

6. Conclusion

SpecMutate demonstrates that automated specification diagnosis and repair for Python/Hypothesis property-based tests is tractable and effective within hackathon constraints. The 3-tier cascade achieves 15/15 diagnostic accuracy on a balanced benchmark; the feedback-guided repair loop achieves 10/10 convergence averaging 1.1 iterations. The central insight is architectural: reliable overconstrained detection requires a deterministic implementation-testing gate that no combination of aggregate signal scores can replace. As LLM-generated code and specifications scale, tools that can automatically identify and repair specification defects will become a critical component of trustworthy software synthesis pipelines — particularly for safety-critical AI systems where a wrong spec silently certifies incorrect behavior as correct.

Code and Data

- Code: <https://github.com/Aditya26189/specmutate-coverage-guided-spec-repair>
- Benchmark: `benchmark.json` (15 tasks with planted specs, reference implementations, buggy implementations)
- Results: `results/benchmark_results.json` (full per-task signal values and repair histories)
- Reproducibility: `.llm_cache.json` committed to repository. Repair loop uses `use_cache=True`; reruns return cached responses rather than fresh LLM calls.
- Demo: `streamlit run app.py` — interactive dashboard showing verdict, signal scores, bad constraint, and repair history
- Model: `gemini-2.5-flash` (intended primary; exact model unconfirmable — cache stores prompt hashes without model metadata; API may have fallen back from `gemini-3.5-flash`)

References

- [1] DeMillo, R.A., Lipton, R.J., Sayward, F.G. (1978). Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4), 34–41.
- [2] Zalewski, M. (2014). American Fuzzy Lop (AFL). <https://lcamtuf.coredump.cx/afl/>
- [3] Claessen, K., Hughes, J. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ICFP* 2000.
- [4] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V. (2006). Combinatorial Sketching for Finite Programs. *ASPLOS* 2006.
- [5] Fraser, G., Arcuri, A. (2011). EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. *ESEC/FSE* 2011.
- [6] Lukasczyk, S., Fraser, G. (2022). Pynguin: Automated Unit Test Generation for Python. *ICSE* 2022.
- [7] MacIver, D., Hatfield-Dodds, Z., et al. (2019). Hypothesis: A New Approach to Property-Based Testing. *JOSS* 4(43).
- [8] Pizzorno, J., Berger, E. (2024). CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv:2403.16218*.
- [9] VeriAct: Iterative Specification Repair for JVM via OpenJML. *arXiv:2604.00280*.
- [10] CodeSpecBench: A Benchmark for Executable Specification Generation. *arXiv:2604.12268*.
- [11] CrossHair: An Analysis Tool for Python. github.com/pschaneley/CrossHair.

Appendix: Limitations and Dual-Use Considerations

A.1 False Positives and False Negatives

False positive risk (correct spec misdiagnosed as broken): S1 relies on LLM-generated adversarial implementations. If the LLM generates near-correct implementations that happen to pass a correct spec, S1=0.0 can trigger Gate 2 and produce a false underconstrained verdict. Mitigated by generating 5 diverse implementations and by the deterministic Gate 1 check, but not eliminated.

False negative risk (overconstrained spec not detected): If a spec's precondition is so restrictive that it only tests the "easy case" where the correct implementation happens to satisfy the weakened assertion, Gate 1 will not fire. Gate 1.5 (RemovePrecondition + coverage delta) provides a secondary signal, but this path was not triggered by any task in the benchmark and remains untested.

Near-threshold case: T12 (max_of_list) scores 0.24 on the fusion formula with the current S3=0.5. If S3 were activated and returned S3=1.0 for this task, the score would increase to 0.34 (still correct classification). The system is not fragile on this task, but it should be noted that T15 (absolute_value) at fusion score 0.31 is the task with the smallest margin to the 0.45 correct threshold (0.14 points) — comfortably below, with no misclassification risk.

A.2 Scalability Constraints

- Each task requires 8–15 LLM API calls (S1 generation: 1; S4: 3; repair loop: up to 6). The 15-task benchmark runs in approximately 20–30 minutes with API rate limiting.
- The mutation engine runs one subprocess per mutation candidate. With approximately 10 mutations per spec at 3 seconds each, coverage measurement takes approximately 30 seconds per task. In-process measurement with proper isolation would be required at larger scale.
- API quota exhaustion is mitigated by up to 10 Gemini API keys with round-robin rotation and automatic model fallback. Full quota exhaustion triggers a 5-minute cooldown rather than a crash.

A.3 Dual-Use Considerations

SpecMutate is designed to strengthen specifications, not weaken them. Two dual-use scenarios warrant consideration:

Adversarial spec weakening: The mutation operators that identify bad constraints could theoretically be applied in reverse — deliberately identifying which assertions to remove to allow a backdoored implementation to pass specification review. Mitigations: (1) SpecMutate requires a reference implementation; an adversary who controls the reference already controls the ground truth and does not need this tool. (2) The coverage delta is visible in every output, making systematic weakening auditable. (3) SpecMutate produces a *repaired* (stronger) specification, not a weakened one.

Over-reliance: Developers who treat SpecMutate’s convergence signal as a guarantee of spec correctness may reduce human review of generated specifications. Convergence means the repaired spec passes the reference implementation and rejects the provided buggy implementations — it does not guarantee the spec captures all intended behaviour. SpecMutate is a diagnostic and repair assistant, not a formal verification system.

LLM Usage Statement

Claude (Anthropic) was used for code review, architecture analysis, and documentation drafting during development. All benchmark results, signal values, and ablation numbers were produced by running the actual SpecMutate system and were independently verified against `benchmark_results.json` before inclusion in this report. Core research claims, system architecture, and experimental design are the work of the author. Note: `llm.py` sets `_current_model_index=2` pointing to `gemini-3.5-flash`, which is likely not a valid Google API model name; the API almost certainly fell back to `gemini-2.5-flash` (index 0), but this cannot be confirmed from the cache, which stores responses keyed by prompt hash without model metadata.