Beyond Superficial Tests: Adversarial Refinement for Reliable Property-Based Testing

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated remarkable proficiency in code generation, yet their application to Property-Based Testing (PBT) remains fraught with a "superficiality gap". While LLMs can readily generate syntactically correct tests, they often struggle to bridge the semantic gap between code implementation and its intended invariant logic, resulting in weak properties that provide a false sense of security. To address this, we introduce PROBE, an agentic framework that hardens software properties through *Adversarial Refinement*. Unlike traditional generation approaches, PROBE treats test generation as a game of "semantic asymmetry": it employs a Validator agent to actively generate counter-implementations, which are semantically incorrect codes that satisfy the generated property, to expose loopholes in the specification. Furthermore, PROBE constructs a cross-functional semantic graph to capture deep dependencies often missed by local analysis. Extensive evaluation reveals that PROBE increases mutation scores by 9.79% over baselines. In real-world deployment, PROBE identified 45 previously unknown bugs in top-tier libraries that have been confirmed by developers, demonstrating its ability to uncover deep semantic defects.

1 Introduction

Property-Based Testing (PBT) has emerged as a rigorous paradigm for ensuring software correctness (Fink and Bishop, 1997). Unlike Example-Based Testing (EBT) which verifies discrete input-output pairs (Daka and Fraser, 2014), PBT validates programs against executable *properties*—universal invariants that must hold across the entire input domain. Since its inception with QuickCheck (Claessen and Hughes, 2000), this approach has proven uniquely capable of uncovering deep corner-case defects. However, despite its theoretical superiority, PBT remains underutilized in practice due to a substantial **cognitive barrier**: defining non-trivial properties and implementing effective input generators are notoriously difficult and time-consuming tasks for human developers (Goldstein et al., 2024).

The advent of Large Language Models (LLMs) promised to automate this labor-intensive process, given their proficiency in code synthesis (Islam et al., 2024; Liu et al., 2024). Yet, current approaches to automated PBT generation face a critical **"superficiality gap"**. While LLMs can readily generate syntactically correct test scripts, they struggle to bridge the semantic distance between implementation details and high-level behavioral contracts. Recent studies reveal two distinct failure modes: (1) **Low Executability:** Even with optimized prompting, over 50% of generated tests fail to compile or execute due to hallucinations (Vikram et al., 2024). (2) **Semantic Triviality:** More insidiously, even valid tests often enforce weak properties (e.g., generic type checks) that provide a false sense of security without exercising the target function's core logic. This limitation stems from the lack of **semantic grounding** — general-purpose models treat PBT as a localized translation task, ignoring the broader logical dependencies (e.g., cross-function contracts) required for rigorous specification.

To bridge this gap, we propose PROBE, the first open-source agentic framework explicitly designed to **harden** properties via **Adversarial Refinement**. Unlike traditional methods that treat PBT as a one-shot translation, PROBE frames it as a game of *semantic asymmetry*. We leverage the insight that **construction is harder than falsification**: while synthesizing a comprehensive property requires complex *logic abstraction* (a high-difficulty task), identifying a loophole via a specific *counter-implementation* is a much simpler *concrete validation* task. PROBE exploits this asymmetry by employing a Validator agent that actively seeks to

"break" the generated properties by constructing degenerate implementations—semantically incorrect code but satisfies the generated property—to expose loopholes in the generated weak property. These discovered *weaknesses* are then fed back to a Generator agent to iteratively refine and strengthen the property.

Furthermore, PROBE addresses the limitations of context-isolation through two key innovations: (1) **Semantic Planning:** It constructs a *Cross-function Semantic Graph* to retrieve logical dependencies often missed by local analysis, ensuring properties adhere to global function contracts (e.g., round-trip consistency between encode/decode). (2) **Contextual Grounding:** It utilizes AST-based static analysis to derive physical input constraints, providing a "reasoning scaffolding" that prevents the agent from exploring invalid search domains.

Extensive evaluations demonstrate that PROBE consistently outperforms state-of-the-art baselines, improving mutation scores (Just et al., 2014) by up to **9.79%** while maintaining robust performance gains across diverse LLM backbones. Manual analysis reveals that PROBE not only bridges the semantic gap with a **95% property correctness rate** (compared to 65% for standalone models) but also exhibits superior discriminative power, uniquely uncovering subtle logic flaws overlooked by other approaches. Most notably, in a large-scale deployment on 21 high-impact repositories, PROBE successfully identified **45 previously unknown bugs** in foundation-level libraries such as `CPython`, `scipy`, and `cryptography`, with 32 already fixed by developers. To the best of our knowledge, PROBE is the first open-source framework that treats property strength as an explicit optimization objective through adversarial agentic games.

Our contributions are summarized as follows:

- We present PROBE, a novel agentic framework to automate whole PBT lifecycle, from constraint grounding to adversarial property hardening.
- We introduce *Adversarial Refinement* and *Semantic Planning*, mechanisms that exploit semantic asymmetry and cross-function dependencies to synthesize high-strength properties.
- We provide an extensive empirical evaluation and demonstrate practical impact by detecting 45 previously unknown bugs (with 32 merged fixes) in widely-used Python libraries.

2 Background and Related Work

This section outlines the theoretical foundations of Property-Based Testing and surveys recent advancements in Large Language Model (LLM) driven test generation.

2.1 Property-Based Testing

Property-Based Testing (PBT) is a dynamic verification framework that validates programs against executable specifications, known as *properties* (Claessen and Hughes, 2000; Goldstein et al., 2024). Unlike example-based testing, which verifies specific input-output pairs, PBT employs strategy-based generators to sample diverse inputs x from a domain $\mathcal{X}$. A property φ is a predicate over the execution trace:

$$\varphi : (x, f(x)) \to \{\texttt{true}, \texttt{false}\} \quad (1)$$

We say a function f satisfies φ if :

$$\forall x \in \mathcal{X} : \varphi(x, f(x)) = \texttt{true} \quad (2)$$

When a violation is detected, PBT frameworks typically perform *shrinking* to minimize the failing input into a concise counterexample (MacIver and Donaldson, 2020).

A property represents a *necessary condition* for correctness. For example, a correct `sort` function must satisfy both monotonicity (φ_{ord}) and permutation (φ_{perm}). As shown in Figure 2, each property alone is insufficient. A function returning a constant list satisfies φ_{ord} but violates φ_{perm}, while reversing the input satisfies φ_{perm} but violates φ_{ord}. Only their conjunction $\varphi_{ord} \wedge \varphi_{perm}$ adequately constrains the behavior.

To formalize property effectiveness, we introduce the notion of *behavior sets*. A *behavior* is an input-output pair $(x, f(x))$ representing one possible execution. Let $\mathcal{S}$ denote the set of correct behaviors and $\mathcal{A}_\varphi$ denote the behaviors accepted by φ. A valid property requires $\mathcal{S} \subseteq \mathcal{A}_\varphi$. Our framework leverages the principle that conjunction strengthens properties: $\mathcal{A}_{\varphi_1 \wedge \varphi_2} = \mathcal{A}_{\varphi_1} \cap \mathcal{A}_{\varphi_2}$. By synthesizing and combining multiple properties, we progressively narrow the accepted behavior set $\mathcal{A}_\varphi$ towards $\mathcal{S}$, leaving minimal room for incorrect implementations to pass. In this work, we utilize Hypothesis (MacIver et al., 2019), a mature Python PBT framework, as our execution backend. Figure 3 illustrates how to test `sort` function by using Hypothesis.

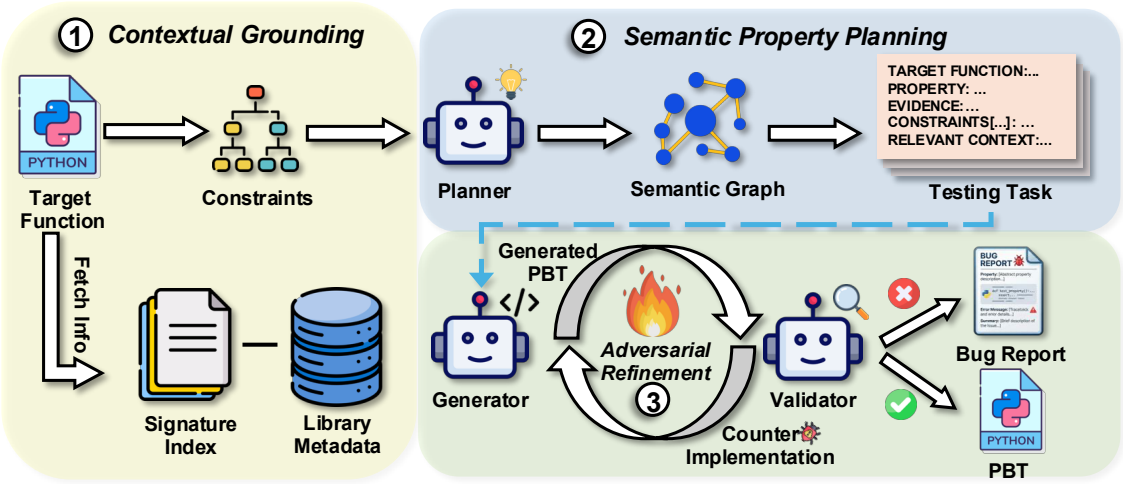


Figure 1: Overall structure of PROBE.

```python
def sort(x: list) -> list:
  return [i for i in range(len(x))]

def sort(x: list) -> list:
  return x[::-1]
```

Figure 2: Incorrect implementations of `sort`.

```python
from collections import Counter
from hypothesis import given, strategies as st
@given(st.lists(st.integers()))
def test_sort(arr):
  sorted_arr = sorted(arr)
  for i in range(1, len(arr)):
    if sorted_arr[i-1] > sorted_arr[i]:
      assert False
  assert Counter(sorted_arr) == Counter(arr)
```

Figure 3: An example of a property-based test using Hypothesis framework.

2.2 LLM-based Test Generation

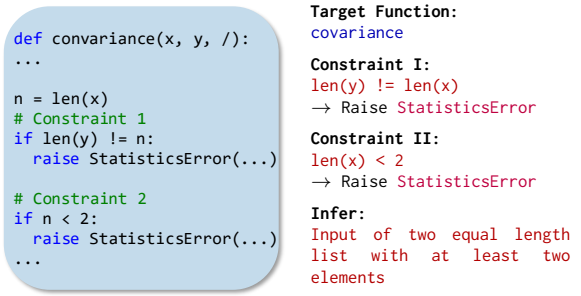
Recent research has explored leveraging LLMs for automated software testing (Chen et al., 2024; Yang et al., 2024; Zhang et al., 2025). Approaches such as MuTAP (Dakhel et al., 2024) utilize mutation testing to guide LLM generation, while Test4Py (Liu et al., 2025) infers parameter types to construct valid inputs. Frameworks like AEGIS (Wang et al., 2024) further employ agent-based architectures with multi-feedback optimization to generate reproduction scripts.

Despite these successes in unit testing, automating property-based testing remains challenging due to the complexity of specifying semantic invariants. Heuristic tools like Hypothesis `ghostwriter` (Hypothesis team) provide syntactic skeletons but still necessitate significant manual effort to fill in semantic logic. Recent works have attempted to leverage LLMs to reduce this burden: Vikram et al. (2024) explored directly prompting LLMs to translate documentation into PBTs, while Maaz et al. (2025) applied general-purpose commercial agents to existing frameworks. However, these approaches largely rely on the intrinsic coding capabilities of LLMs without task-specific optimization. In contrast, PROBE introduces an approach that combines global context with adversarial refinement to generate more rigorous PBTs.

3 Methodology

In this section, we present PROBE, an agentic framework for automatically generating high quality PBTs. Unlike single-pass generation methods, PROBE orchestrates a collaborative workflow among specialized agents to produce PBTs that are both syntactically valid and semantically robust.

As illustrated in Figure 1, the PROBE comprises three stages: (1) **Contextual Grounding**, employing AST-based static analysis to derive physical constraints defining valid input spaces; (2) **Semantic Property Planning**, which contextualizes the target function within a repository-wide semantic graph to infer implicit properties; and (3) **Adversarial Refinement**, where the Generator and Validator engage in a minimax game to iteratively harden PBT strength by synthesizing and falsifying counter-implementations. This workflow produces either a semantically robust PBT or a bug report grounded in rigorous evidence.



(a) Source Code **(b) Extracted Constraints**

Figure 4: Static constraints extraction of `covariance` from `statistics` package.

3.1 Contextual Grounding

To avoid trivial inputs that merely trigger intentional error handlers (e.g., `ValueError`), PROBE implements **Contextual Grounding** to derive physical constraints directly from the source code. The system first analyzes the Abstract Syntax Tree (AST) to identify *guard clauses* (e.g., `raise` or `assert`). Instead of treating these as isolated code snippets, PROBE resolves them into **Physical Constraints** through backward data-flow analysis. As illustrated in Figure 4, it maps local variables back to inputs (e.g., converting `if n < 2` to `len(x) >= 2`). These constraints prune the input search space, establishing a "logical safety zone" that allows the agent to focus on inferring implicit semantic guarantees rather than satisfying basic preconditions.

3.2 Semantic Property Planning

The primary challenge in automated PBT generation lies in the *semantic isolation* of LLMs, which often struggle to infer invariant logic by analyzing a single function in a vacuum. To bridge this gap, PROBE treats PBT generation not as a localized coding task, but as a systematic planning process over a **Cross-function Semantic Graph** $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Here, nodes $\mathcal{V}$ represent functions within the repository, and edges $\mathcal{E}$ encode implicit logical contracts and behavioral dependencies.

The Planner agent orchestrates a three-stage workflow to construct this graph and synthesize high-strength properties:

❶ Relational Neighborhood Identification Rather than ingesting the entire repository, which introduces excessive noise, the Planner first isolates the *logical neighborhood* of the target function f_{target}. It traverses the import dependencies and class hierarchies to isolate a subset of modules $\mathcal{M}_{rel}$. This minimizes noise and establishes strict boundaries for relevant context retrieval.

❷ Contractual Edge Discovery The core innovation of the Planner lies in its ability to identify **Implicit Contracts** — behavioral consistencies that span multiple functions. Using the signature index $\mathcal{I}$, the Planner iteratively scans $\mathcal{M}_{rel}$ to identify functions that share relational signatures with f_{target}. Specifically, it searches for three types of semantic edges:

- **Inverse Operations:** Identifying pairs such as `encode`/`decode`, implying the round-trip consistency ($f_{inv}(f_{target}(x)) = x$).
- **State Invariants:** Identifying operations that should preserve a global state, such as `push`/`pop` in a stack or `add`/`remove` in a container.
- **Relational Identities:** Identifying logical properties between distinct functions that share underlying semantics. For example, the standard deviation function should yield a result consistent with the square root of the variance function.

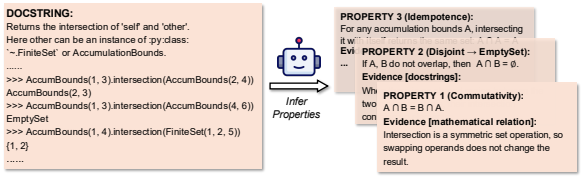


Figure 5: Example of evidence-grounded property.

To mitigate hallucination, PROBE enforces **Evidence-Grounded Inference**. As demonstrated in Figure 5, the Planner is strictly required to map every synthesized property φ to explicit evidence e (e.g., docstrings or algebraic identities) extracted during the discovery phase.

The result is consolidated into a formal **Testing Task** $t = \langle f_{target}, ic, \varphi, c_\varphi, e \rangle$, serving as a traceable blueprint for Algorithm 1. Here, ic denotes the physical input constraints derived in Section§ 3.1 and c_φ the enriched context.

3.3 Adversarial Refinement

Given a testing task t, the Generator synthesizes a PBT that encodes the specified property φ. Initially, the generated PBT is executed in an isolated environment. If the execution fails, the Validator performs root cause analysis based on the execution logs and t, classifying failures into three categories:

- *Test Code Defects:* Syntax errors or incorrect function usage.
- *Property Design Defects:* semantic misalignment between the property and evidence, or discrepancies between the evidence and docstrings.

Algorithm 1 Generator-Validator Workflow

Require: Testing Task $t = \langle f_{target}, ic, \varphi, c_\varphi, e \rangle$
Ensure: Hardened PBT **or** Bug Report

$PBT \leftarrow$ GENERATOR.GENPBT(t)
$Env \leftarrow$ ENV-CREATE(t)
$cnt_{fix}, cnt_{ref} \leftarrow 0, 0$ ▷ Counters for T_{fix} and T_{ref}
while $cnt_{fix} \leq T_{fix}$ **do**
  $res \leftarrow$ RUN(PBT, f_{target}, Env)
  if res.status == PASS **then**
    if $cnt_{ref} == T_{ref}$ **then**
      return PBT
    $f' \leftarrow$ VALIDATOR.GENADVERSARY(PBT, t)
    if $f' \neq \varnothing$ **then**
      $res' \leftarrow$ RUN(PBT, f', Env)
      if res'.status == PASS **then**
        $t \leftarrow$ PLANNER.REFINE(PBT, f', t)
        $PBT \leftarrow$ GENERATOR.GENPBT(t)
        $cnt_{ref} \leftarrow cnt_{ref} + 1$; $cnt_{fix} \leftarrow 0$
      else
        return PBT
    else
      return PBT
  else
    $cause, info \leftarrow$ VALIDATOR.DIAGNOSE(res, PBT, t)
    if $cause$ == CODE_DEFECT **then**
      $PBT \leftarrow$ FIXCODE($PBT, info$)
    else if $cause$ == LIB_DEFECT **then**
      return CREATEBUGREPORT($t, info$)
    else if $cause$ == PROP_DEFECT **then**
      $t \leftarrow$ PLANNER.REPLAN($t, info$)
      $PBT \leftarrow$ GENERATOR.GENPBT(t)
    $cnt_{fix} \leftarrow cnt_{fix} + 1$
return PBT

- *Library Defects:* The property is valid and the test code is correctly implemented, yet execution fails. This triggers an immediate bug report.

The diagnostic loop is bounded by a limit T_{fix}. However, achieving a passing state in this loop merely guarantees executability. It does not ensure the property's rigor.

Transitioning from correctness to quality, we address a fundamental limitation in PBT generation known as the *propensity for triviality*: LLMs often converge on weak properties that are syntactically valid but semantically vacuous. To address this, PROBE frames property refinement as a minimax game predicated on **Difficulty Asymmetry** — the principle that verifying or falsifying a solution is often significantly easier than generating it (Wei, 2024). We posit that while synthesizing a globally optimal property is a high-entropy inductive challenge, identifying a local loophole via a *counter-implementation* is a significantly more tractable deductive task.

The Adversarial Objective Upon receiving a passing PBT, the Validator assumes the role of a *Semantic Adversary* to challenge the sufficiency of φ. Instead of verifying φ, it actively seeks to construct a *counter-implementation* f'—a code variant that is *syntactically plausible* and satisfies the current property φ, yet is *semantically incorrect* (e.g., dropping the last element of the output list for a sort function). This process exploits the inherent asymmetry between *construction* and *breaking*. Constructing f' provides the system with a *concrete falsification witness*. When f' successfully "cheats" the current φ, it yields a high-signal feedback message for the Generator: "The current property is too weak to distinguish intended behavior from this specific degenerate f'." This evidence-driven feedback is more effective as it anchors the LLM's reasoning in a tangible failure case.

As illustrated in Algorithm 1, this adversarial process is iterative. Each successful f' acts as a high-signal feedback, forcing the Generator to tighten the PBT with more rigorous properties (e.g., moving from "output is sorted" to "output is a permutation of the input and is sorted"). The loop continues until the Validator fails to synthesize a property-passing f' within T_{ref} attempts, effectively collapsing the accepted behavior set toward the ground truth intended behaviors.

4 Experiment

To comprehensively evaluate the performance and practical value of PROBE, we design experiments to answer the following research questions:

- **RQ-1 Effectiveness:** How effectively can PROBE generate non-trivial PBTs that distinguish correct implementations from erroneous ones compared to baselines.
- **RQ-2 Validity & Distinctiveness:** What proportion of generated PBTs are syntactically and semantically correct, and how distinctive are the properties discovered by each tool?
- **RQ-3 Bug Finding:** Can PROBE discover previously unknown bugs in widely-used libraries?
- **RQ-4 Ablation Study:** How does each component contribute to the overall performance of PROBE?

4.1 Experiment Setup

Baseline Methods To our knowledge, current methods for automated PBT generation primarily utilize LLM through two main paradigms: employing an LLM via direct prompting (Vikram et al., 2024) or leveraging a commercial agent (Maaz

et al., 2025). To thoroughly evaluate PROBE, we categorize our baselines into three configurations:

- **LLM:** We select several leading LLMs to serve as our primary baseline, including DeepSeek-V3.2 (DeepSeek-AI et al., 2025), GPT-5 (OpenAI, 2025) and Qwen3-Next-80B-A3B-Thinking (QwenTeam, 2025a). This category assesses the inherent reasoning and code generation capabilities of LLMs.
- **Retrieval-Augmented LLM (LLM+RAG):** We implement the second baseline by equipping the LLM with Retrieval-Augmented Generation (RAG). This setup provides the model with a more precise context of the repository.
- **LLM-Based Agent:** Finally, we include an LLM-based agent to represent a method optimized for complex software tasks. Claude Code (Anthropic, 2025a) has been explored to generate PBTs. We select it with claude sonnet 4.5 (Anthropic, 2025b) as our third baseline, which features an state-of-art agent designed for code generation beyond simple prompting.

Baselines details are provided in Appendix B.4.

Datasets We construct two datasets for evaluation. *Dataset I* serves as the primary testbed for RQ1, RQ2, and RQ4. We derived 256 non-trivial functions from four diverse PyPI packages: `sympy` (mathematics), `sortedcontainers` (data structures), `more-itertools` (utilities), and `simplejson` (serialization). These libraries span various domains, reflecting the diversity of properties encountered in practice. To ensure meaningful evaluation, we apply a systematic filtering pipeline based on code complexity metrics (detailed in Appendix B.2). *Dataset II* is curated for RQ3, which encompasses 21 high-impact repositories, including 8 Python Standard Library modules and 13 prominent PyPI packages. Within each package, we manually identified functions characterized by intricate semantic logic.

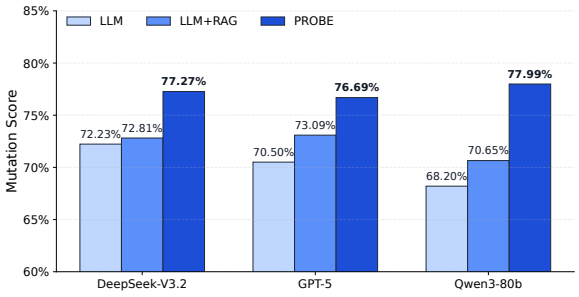


Figure 6: Performance comparison of PROBE against baselines across different backbone models.

4.2 RQ-1: Effectiveness

To quantify PBT strength, we employ mutation testing (DeMillo et al., 1978), a standard method for evaluating fault-detection capability of a test suite. The core principle of mutation testing is to systematically inject artificial faults into the original program using predefined mutation operators, producing modified variants known as *mutants*. Previous studies have demonstrated a statistically significant correlation between mutant detection and real fault detection (Just et al., 2014; Ravi and Coblenz, 2025). A mutant is considered *killed* if the PBT passes on the original implementation but fails on the mutated version. Higher mutation scores indicate stronger discriminative power of tests.

To ensure evaluation independence, we utilized the MUTMUT tool (Mutmut contributors, 2025) to generate up to 10 mutants per function. We compute mutation score exclusively on the *intersection* of functions where all evaluated tools produce PBTs that pass on the original implementation. This intersection-based protocol eliminates potential bias arising from tools that generate valid PBTs on different subsets of functions. For completeness, we report the mutation scores per-tool on the full dataset in Appendix B.1.

Results. Table 1 presents the mutation scores on a set of 82 functions (695 mutants). The best performance in each category is highlighted in bold. Among all evaluated configurations, PROBE (Qwen3-80b) achieves the best overall performance, attaining the highest mutation score of **77.99%** by killing 542 out of 695 mutants. This represents a 9.79 percentage point improvement over the vanilla Qwen3-80b baseline.

This performance advantage is consistent across architectures. As illustrated in Figure 6, when applied to DeepSeek-V3.2 and GPT-5, PROBE boosts mutation scores to **77.27%** and **76.69%**, respectively. Crucially, PROBE outperforms Claude Code (Sonnet 4.5) by margins of 5.90~7.20 percentage points. These results suggests that our domain-specific structured approach is more effective than general-purpose methods for PBT generation.

4.3 RQ-2: Validity & Distinctiveness

To evaluate the semantic validity of generated PBTs, we manually analyzed PBTs in representative top-performers in RQ-1.

We conduct a systematic manual analysis to evaluate the **validity** of PBTs. Specifically, three senior

Table 1: Comparison of Mutation Score and Cost across evaluation subjects. Columns *SymPy*, *M-Iter* (more-itertools), *Sorted* (sortedcontainers), and *JSON* (simplejson) report the killed mutants for each package. *Mut. Score* represent the mutation score. *Cost* denotes the total expenditure in USD.

Method	Killed Mutate per Package				Overall Performance		
	SymPy	M-Iter	Sorted	JSON	Total Killed	Mut. Score	Cost
LLM							
Qwen3-80b	83	178	207	6	474	68.20%	$12.23
DeepSeek-V3.2	87	186	215	14	502	72.23%	**$0.41**
GPT-5	91	176	217	6	490	70.50%	$18.89
LLM + RAG							
Qwen3-80b + RAG	83	180	209	19	491	70.65%	$20.47
DeepSeek-V3.2 + RAG	92	190	210	14	506	72.81%	$1.75
GPT-5 + RAG	98	184	214	12	508	73.09%	$53.95
LLM-Based Agent							
Claude Code (Sonnet 4.5)	88	192	205	7	492	70.79%	$53.57
Ours							
PROBE (Qwen3-80b)	99	**207**	**229**	7	**542**	**77.99%** (↑4.90~9.79%)	$35.26
PROBE (DeepSeek-V3.2)	**101**	195	227	14	537	77.27% (↑4.18~5.04%)	$6.07
PROBE (GPT-5)	86	201	224	**22**	533	76.69% (↑3.60~6.19%)	$86.10

PhD students independently reviewed 20 randomly sampled PBTs from each tool. We adopt a *strict binary criterion* for validity, classifying a PBT as **False** if it exhibits implementation defects (e.g., syntax errors) or contains *any* incorrect property. Conversely, a PBT is considered **True** only if it is both syntactically executable and *all* asserted properties are semantically sound. Inter-annotator agreement is measured via Fleiss' κ.

To further evaluate the uniqueness of the valid properties, we analyze mutant detection at the individual mutant level. A property is considered as *distinctive* if it kills mutants that remain undetected by the correct properties from other tools.

Results. Among these samples, the annotators reached a complete consensus on 73 cases, resulting Fleiss' $\kappa = 0.83$, indicating a substantial level of agreement. Table 2 presents the validity results. DeepSeek-V3.2 and GPT-5 + RAG achieve validity rates of 65.0% and 70.0%, respectively. In contrast, both Claude Code and PROBE demonstrate substantially higher correctness, achieving 100% and 95.0%. Notably, while Claude Code achieves perfect correctness on these samples, it attains a lower mutation score compared to PROBE. This suggests a potential trade-off between property conservativeness and fault-detection capability. General purpose models may generate semantically valid but overly general properties that lack the specificity required to detect subtle behavioral deviations. PROBE strikes a favorable balance. We detailed root cause analysis of the single failed case in Appendix A.1.

Figure 7 illustrates the overlap of killed mutants across tools. PROBE uniquely killed 6 mutants, demonstrating its ability to discover stronger properties that capture subtle semantics overlooked by other approaches.

Table 2: Evaluation of PBT validity.

Method	True	False
DeepSeek-V3.2	13	7
GPT-5 + RAG	14	6
Claude Code	20	0
PROBE (Qwen3-80b)	19	1

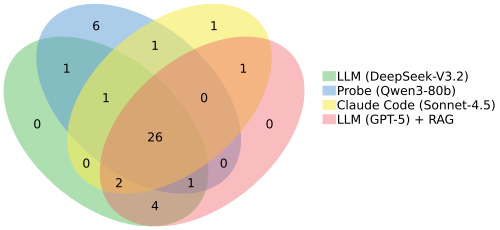


Figure 7: Overlap of killed mutants across evaluated tools. Each region indicates the number of mutants detected by the corresponding tool.

4.4 RQ-3: Bug Finding

We manually verified each bug report and submitted issues to the corresponding GitHub repositories. To balance performance with computational cost, we employ Qwen3-30B-A3B-Thinking (Qwen-Team, 2025b) as the backbone LLM of PROBE. To minimize noise for maintainers, we only submitted reports that were reproducible and interpretably erroneous based on our manual inspection

Table 3: Summary of defects discovered by PROBE. *Rep.* denotes the total issues reported. *Conf.* indicates bugs confirmed by developers. *Fixed* denotes issues resolved with merged patches. *Doc.* refers to documentation updates instead of code changes.

Repository (Star Count)	Rep.	Conf.	Fixed	Doc.
Python Standard Library (70.5k)				
json / html / re	5	4	3	1
functools	1	1	1	1
collections	2	1	1	0
statistics	2	1	1	0
shlex	1	1	0	0
types	1	1	0	0
PyPI Packages				
boto / boto3 (9.6k)	1	0	0	0
marshmallow (7.2k)	3	2	1	0
scipy / scipy (14.3k)	4	4	2	0
sympy / sympy (14.2k)	13	13	9	0
pyproj4 / pyproj (1.2k)	2	1	0	1
tobgu / pyrsistent (2.2k)	1	0	0	0
cpburnz / pathspec (205)	3	1	1	0
pyca / cryptography (7.4k)	2	2	2	0
scikit-hep / awkward (929)	8	6	4	0
networkx / networkx (16.4k)	1	1	1	0
aws-lambda-powertools (3.2k)	3	3	3	2
google-deepmind / optax (2.1k)	2	2	2	0
huggingface / tokenizers (10.3k)	1	1	1	0
Total	**56**	**45**	**32**	**5**

Results. As summarized in Table 3, we have submitted 56 issues, of which 45 were confirmed by the developers with 32 fixes. Notably, all confirmed defects were previously unknown, demonstrating PROBE's ability to detect subtle semantic bugs. Among the confirmed issues, 5 were resolved through documentation amendments rather than code changes, as developers attempted to preserve backward compatibility for existing users. Details of all reported defects are provided in Appendix C. The remaining raw reports, which involve highly specialized domain logic beyond our immediate expertise, will be released on our repository to invite community adjudication.

Table 4: *w/o SeGra.* and *w/o Adv.* exclude the Semantic Graph and Adversarial Refinement, respectively. Red percentages denote performance drops.

Backbone Model	Full System	w/o SeGra.	w/o Adv.
DeepSeek-V3.2 *(680B)*	**83.57**	73.43 (-10.1%)	76.78 (-6.8%)
Qwen3-80B *(80B)*	**81.93**	70.36 (-11.6%)	77.69 (-4.2%)
GPT-5 *(Closed)*	**82.23**	74.78 (-7.5%)	77.01 (-5.2%)

Table 5: Performance of PROBE across different LLM backbones. (Δ) denotes the improvement compared to the corresponding vanilla LLM.

Backbone LLM	Size	Mut. Score	Δ
Qwen3-80b	80B	77.99%	+9.79%
DeepSeek-V3.2	685B	77.27%	+5.04%
GPT-5	-	76.69%	+6.19%

4.5 RQ-4: Ablation Study

To quantify the impact of PROBE's core components, we performed an ablation study by disabling semantic graph and adversarial refinement.

Results. Results in Table 4 are reported on the intersection of functions where all variants successfully generated passing PBTs. Semantic graph proves to be a critical module. Its removal precipitates the performance decline with drops ranging from 7.5% to 11.6% across models. This underscores that retrieving cross-function dependencies is essential for strengthening PBTs. Adversarial refinement also plays an indispensable role in hardening properties. Disabling it reduces mutation scores by 4.2% to 6.8%, which confirms that the adversarial loop effectively filters out trivial properties that fail to detect subtle bugs.

Crucially, the relative impact of these components remains consistent across different LLMs. As shown in Table 5, the full PROBE system consistently outperforms its vanilla LLM regardless of the backbone LLMs. This suggests that the performance gains are intrinsic to our structured framework design rather than being an artifact of a specific underlying LLM's capability.

5 Conclusion

In this paper, we introduce PROBE, the first open-source agentic framework designed to automate the lifecycle of Property-Based Testing. PROBE addresses the critical limitations of existing methods. Our approach ensures that generated PBTs are not only executable but also able to capture complex semantic logic with cross-function properties. Extensive evaluations demonstrate that PROBE significantly outperforms the baselines in mutation scores, which achieve up to 9.79%. Notably, PROBE discovered 45 previously unknown bugs from top-tier Python libraries, with 32 already fixed and merged. In addition, the ablation study shows that each component of PROBE has contribution to the overall performance.

Limitations

We acknowledge several limitations inherent to PROBE. First, agentic architectures that prioritize test quality through iterative refinement inherently involve more computation than pure LLM generation methods, which is a trade-off between test robustness and inference efficiency common to all refinement-based approaches. We mitigate this overhead through static analysis to constrain the search space and modular agent design to minimize context sizes, though further optimization remains an avenue for future work. Second, the effectiveness of LLM-based testing frameworks is partially bounded by the reasoning capabilities of the underlying language models. We address it by designing targeted prompts and decomposing complex tasks across specialized agents to reduce the burden on any single inference call. Finally, a fundamental challenge in automated PBT generation—shared across existing methods—is distinguishing genuine specification violations from intentional design. Properties derived from docstrings may not always align with developer's intent, occasionally yielding false positives. Fully resolving this challenge remains an open problem.

References

Anthropic. 2025a. Claude code. https://www.anthropic.com/claude-code. Powered by Claude Sonnet 4.5.

Anthropic. 2025b. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet4-5.

G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation. In *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, page 57–58, New York, NY, USA. Association for Computing Machinery.

Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 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.

Koen Claessen and John Hughes. 2000. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA. Association for Computing Machinery.

Ermira Daka and Gordon Fraser. 2014. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211.

Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology*, 171:107468.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

George Fink and Matt Bishop. 1997. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80.

Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Hypothesis team. Integrations Reference: Ghostwriter (Hypothesis 6.148.9 Documentation). https://hypothesis.readthedocs.io/en/latest/reference/integrations.html#ghostwriter. Accessed: 2025-11-01.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. MapCoder: Multi-agent code generation for competitive problem solving. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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*, FSE 2014, page 654–665, New York, NY, USA. Association for Computing Machinery.

Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. 2024. Llm-based java concurrent program to arkts converter. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2403–2406, New York, NY, USA. Association for Computing Machinery.

Runlin Liu, Zhe Zhang, Yunge Hu, Yuhang Lin, Xiang Gao, and Hailong Sun. 2025. Llm-based unit test

generation for dynamically-typed programs. *arXiv preprint arXiv:2503.14000*.

Muhammad Maaz, Liam DeVoe, Zac Hatfield-Dodds, and Nicholas Carlini. 2025. Agentic property-based testing: Finding bugs across the python ecosystem. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

David MacIver, Zac Hatfield-Dodds, and Many Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4:1891.

David R. MacIver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:27, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Thomas J. McCabe. 1976. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 407, Washington, DC, USA. IEEE Computer Society Press.

Mutmut contributors. 2025. mutmut: Mutation testing for python. Accessed: 2025-12-03.

OpenAI. 2025. Gpt-5 system card. Accessed: 2025-11-29.

QwenTeam. 2025a. Qwen3-next: Towards ultimate training & inference efficiency. Accessed: 2025-11-29.

QwenTeam. 2025b. Qwen3: Think deeper, act faster. Accessed: 2025-11-29.

Savitha Ravi and Michael Coblenz. 2025. An empirical evaluation of property-based testing in python. *Proc. ACM Program. Lang.*, 9(OOPSLA2).

Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2024. Can large language models write good property-based tests? *Preprint*, arXiv:2307.04346.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. 2024. Aegis: An agent-based framework for general bug reproduction from issue descriptions. *arXiv preprint arXiv:2411.18015*.

Jason Wei. 2024. Asymmetry of verification and verifier's rule. Accessed: 2025-12-03.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and 1 others. 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.

Zhe Zhang, Xingyu Liu, Yuanzhang Lin, Xiang Gao, Hailong Sun, and Yuan Yuan. 2025. Reference-based retrieval-augmented unit test generation. *ACM Trans. Softw. Eng. Methodol.* Just Accepted.

A Case Analysis

A.1 Failed Case Analysis

To better understand the validity results in RQ-2, we analyze the single failed case generated by PROBE. The target function is `sympy.series.gruntz.compare(a, b, x)`, a core utility within the Gruntz limit algorithm. The docstring for `compare` succinctly states: *"returns '<' if $a < b$, '=' if $a == b$, and '>' if $a > b$."* In isolation, this wording strongly resembles a conventional total order, which can lead a general-purpose LLM to interpret `compare` as comparing numeric values. Guided by this interpretation, PROBE synthesized the following PBT, asserting that distinct powers of x should be strictly ordered:

$$\forall m, n \in \mathbb{Z}^+, \quad m < n \implies \texttt{compare}(x^m, x^n, x) = \texttt{"<"}$$

This property is *invalid* under the intended semantics of `compare`. In the Gruntz framework, the comparison operators correspond to a coarse asymptotic classification used for limit computation, rather than a fine-grained ordering over polynomial degrees. In particular, polynomial powers typically fall into the same comparability class (e.g., $x^m \asymp x^n$ for $m, n > 0$), so `compare` may legitimately return `"="` even when $m < n$. Consequently, the generated PBT fails not because of a defect in SymPy, but due to a *semantic gap* between an underspecified docstring and a domain-specific interpretation.

This failure case highlights a practical limitation of LLM-driven test generation: PROBE relies on docstrings being sufficiently explicit to disambiguate overloaded mathematical terminology. When docstrings use common relational language to denote specialized notions, even rigorous evidence-grounded inference can default to the standard interpretation and produce invalid PBTs.

A.2 Bug Cases Analysis

To provide qualitative insights into PROBE's efficacy, we present a comparative analysis of PBT generation in Figure 8 and a showcase of discovered real-world defects in Figure 9.

Figure 8 illustrates that using the mark_ends function, where a logic bug was injected. The LLM

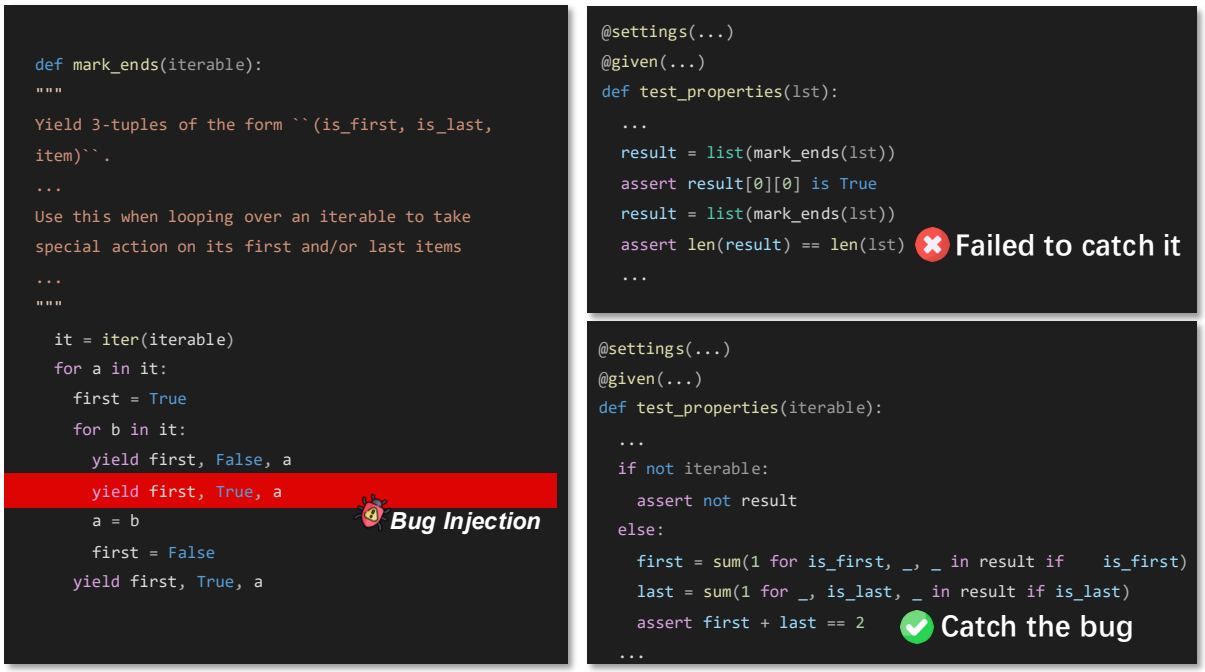
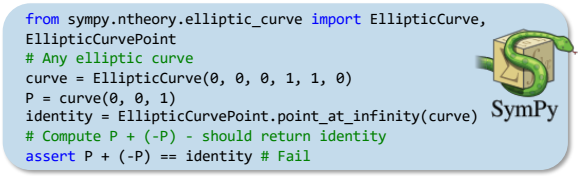
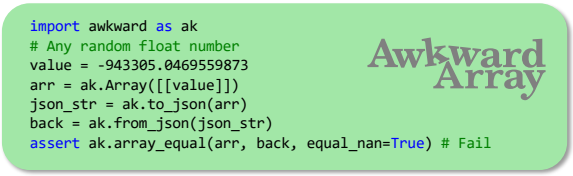


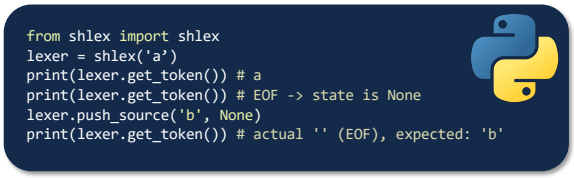
Figure 8: Comparative analysis of generated PBTs between PROBE and LLM.



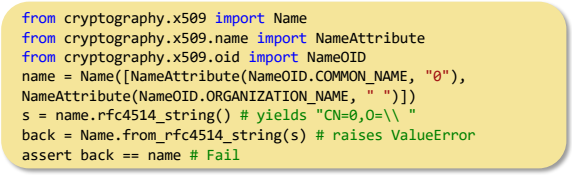
(a) Sympy: Violation of group property of elliptic curves.



(b) Awkward: Data loss through the round-trip.



(c) CPython: Shlex failed to reset the lexer state.



(d) Cryptography: Non-parseable output for trailing spaces.

Figure 9: Exemplary round-trip like bugs from reported issues.

generates superficial assertions that pass despite the bug, exemplifying the weak property problem inherent in standard prompting. In contrast, PROBE synthesizes a precise invariant verifying that the sum of is_first and is_last flags must equal 2 for the sequence, successfully catching the bug.

Figure 9 shows 4 minimal reproduced codes for exemplary round-trip-like bugs from our reported issues. In Sympy, PROBE identified the fundamental group property $P + (-P) = O$ (Identity) violation. The library computed an incorrect sum for a point and its inverse on a specific elliptic curve, violating the mathematical axioms the library claims to support. In package Awkard, PROBE detected a precision loss when converting floating-point arrays to JSON and back. The deserialized value differed from the original input, indicating a failure in preserving data fidelity during the serialization process. In CPython, the shlex lexer failed to correctly reset its internal EOF state when a new input source was pushed after the previous one was exhausted. This caused the lexer to prematurely stop processing valid subsequent inputs. Finally, PROBE detected that the Cryptography library successfully serialized but failed to parse that the same string back into an object, raising a ValueError. These bugs are detected through cross-function properties, which are typically unreachable by trivial properties.

B Experiment Details

All experiments were conducted on a Ubuntu 20.04 server with dual Intel(R) Xeon(R) Gold 5218 @2.30GHz CPU with 128 GB of RAM, 2 NVIDIA A100-SXM4-80GB GPUs. We set both the diagnostic retry limit T_{fix} and the adversarial refinement limit T_{ref} to 3 across all experiments.

B.1 Mutation Score with full dataset

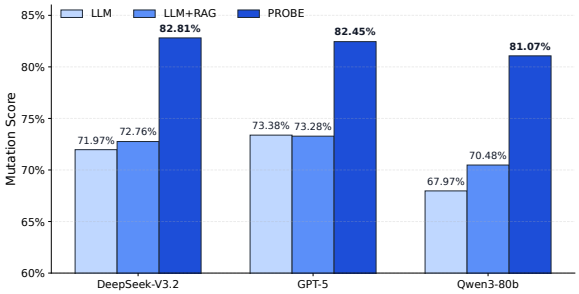


Figure 10: Performance comparison of PROBE against baselines across different backbone models (full dataset).

Method	Pass%	Killed / Total	Score
LLM			
Qwen3-80b	73.04%	1193 / 1608	67.97%
DeepSeek-V3.2	58.99%	937 / 1302	71.97%
GPT-5	84.77%	1384 / 1886	73.38%
LLM + RAG			
Qwen3-80b + RAG	52.73%	814 / 1155	70.48%
DeepSeek-V3.2 + RAG	47.27%	708 / 973	72.76%
GPT-5 + RAG	77.34%	1245 / 1699	73.28%
LLM-Based Agent			
Claude-Code (sonnet 4.5)	**97.66%**	1521 / 2186	69.58%
Ours			
PROBE (Qwen3-80b)	76.20%	1366 / 1685	81.07%
PROBE (DeepSeek-V3.2)	74.22%	**1363 / 1646**	**82.81%**
PROBE (GPT-5)	90.60%	1668 / 2029	82.45%

Table 6: *Pass%* denotes the percentage of functions for which the tool generated a passing PBT. *Killed / Total* denotes the number of killed mutants / total mutants. *Score* denotes the mutation score.

Table 6 and Figure 10 present mutation testing results on the complete Dataset I (256 functions), complementing the intersection-based analysis in Section§ 4.2. Unlike the main evaluation, which controls for function coverage by restricting to the intersection where all tools produce passing PBTs, this analysis evaluates each tool on all functions for which it successfully generates a passing PBT.

PROBE achieves substantially higher mutation scores compared to all baselines across the full dataset. PROBE (DeepSeek-V3.2) attains the highest mutation score of 82.81%, followed by PROBE (GPT-5) at 82.45% and PROBE (Qwen3-80b) at 81.07%. These scores represent improvements of 9~14 percentage points over the corresponding LLM baselines and 7~10 percentage points over LLM+RAG configurations. Notably, the performance gap between PROBE and baselines is even more pronounced on the full dataset than on the intersection (Table 1), suggesting that PROBE's advantages are particularly evident on challenging functions where baselines struggle to generate effective PBTs.

Pass rate measures the proportion of functions for which a tool generates a syntactically valid PBT that passes on the original implementation. Claude Code achieves the highest pass rate (97.66%), followed by PROBE (GPT-5) at 90.60%. However, high pass rate does not necessarily indicate effective PBT generation. Claude Code, despite its superior pass rate, attains a mutation score of only 69.58%. This confirms our observation that existing methods tend to generate conservative properties that pass easily but lack discriminative power.

B.2 Dataset Construction Details

To construct a representative dataset, we applied a systematic filtering process to functions collected from sympy, more-itertools, simplejson, and sortedcontainers packages. Starting from an initial corpus of 21,319 candidate functions, we applied a systematic filtering process based on code complexity metrics. Our selection criteria established both minimum and maximum thresholds to exclude trivial implementations and overly complex functions that would confound evaluation. Minimum thresholds required at least 5 lines of code (LOC), 3 source lines of code (SLOC), and a McCabe's Cyclomatic Complexity (CC) (McCabe, 1976) of at least 2 to ensure meaningful implementations containing at least one decision point. Maximum thresholds constrained LOC $\leq$ 200, CC $\leq$ 15, Cognitive Complexity (Campbell, 2018) $\leq$ 25, maximum nesting depth $\leq$ 5, and number of parameters $\leq$ 7. Additionally, we required all functions to include docstrings and available Python source code. To ensure library diversity and prevent overrepresentation, we sampled at most 100 functions per package.

After applying all criteria, 256 functions were retained for the final dataset, comprising 100 functions from sympy, 100 from more-itertools, 48

Metric	Min	Max	Mean	Median
Cyclomatic Complexity	2	15	4.82	4
Cognitive Complexity	0	21	5.20	3
Lines of Code	5	194	35.80	32

Table 7: Complexity statistics of selected functions

from sortedcontainers, and 8 from simplejson. Table 7 presents the complexity distribution of the final dataset.

B.3 LLMs Configuration

We access all language models through official APIs or local deployment. For local deployment, we host Qwen3-30B-A3B-Thinking using **SGLang**[1]. For API-based models, we access DeepSeek-V3.2 through the official DeepSeek API[2], GPT-5 through OpenRouter[3], and Qwen3-Next-80B-A3B-Thinking through Alibaba Cloud[4]. For all models, we retain the default temperature and sampling configurations provided by each platform without modification.

B.4 Baseline Details

B.4.1 LLM

We provide the LLM with the target function's metadata, and prompt it to identify properties and generate corresponding PBTs. The generated PBTs are then executed against the target function. Since we found that single-pass generation yields prohibitively low pass rates due to the syntax or logic error, we incorporate a self-repair mechanism: if execution fails, we feed the error message back to the LLM for repair, allowing a maximum of 3 fix attempts. This provides a more fair comparison.

B.4.2 Retrieval-Augmented LLM (LLM+RAG)

We adopt LLM+RAG framework as another baseline. Given a target project, we decompose the codebase into function-level units, where each unit corresponds to a single function definition. This granularity preserves the semantic coherence of each functional unit and is well aligned with property-based test generation, as it facilitates the identification of properties that recur across functions .

All function units are embedded using **bge-m3**[5] and indexed in the **Chroma**[6] vector store. During test generation, the language model first formulates a set of abstract property descriptions for the target function. For each property, we retrieve the top-k semantically related functions from the codebase based on embedding similarity, followed by a lightweight cosine-similarity re-ranking step to refine relevance.

The retrieved functions are then supplied to the model as auxiliary context, together with the target function and the corresponding property description, to guide the generation of executable PBTs. All generated tests are subsequently executed, and their outcomes are analyzed by the language model to assess compliance with the intended properties. When violations are observed, the model iteratively refines the generated tests to resolve inconsistencies between the specified properties and observed execution behavior.

B.4.3 Claude Code

We employ the official Claude CLI (Anthropic, 2025a) with Claude Sonnet 4.5 as the underlying model. Following the prompt design from Maaz et al. (2025)'s work, we instruct the agent to analyze the target function and generate PBTs. Claude Code operates as a fully autonomous agent capable of reading files, executing code, and iteratively fixing its outputs without explicit prompts. This baseline represents the current state-of-the-art in commercial LLM-based coding agents.

C Summary of Reported Issues

This section provides detailed information on the bugs discovered by PROBE during our evaluation in RQ-3 (Section § 4.4). We only reported issues that we assessed as genuine bugs with potential impact on real-world users. Table 8 summarizes the detailed information. For each reported bug shown in table, we provide the affected repository with star count, module, the issue tracking ID (anonymized with *), current resolution status, fix type, and a brief description of the defect. Complete reports are available on our GitHub repository.[7]

D Prompt Details

[1] https://github.com/sgl-project/sglang
[2] https://api-docs.deepseek.com/
[3] https://openrouter.ai/
[4] https://bailian.console.aliyun.com/
[5] https://huggingface.co/BAAI/bge-m3
[6] https://github.com/chroma-core/chroma
[7] https://anonymous.4open.science/r/PROBE-C7A5

Table 8: Summary of Bug Findings.

Repo Name (Star)	Module	Issue ID	Issue State	Fix Type	Description
cpython (70.5k)	re	*	FIXED	CODE FIX	Crash when using multiple capturing groups in re.Scanner.
	html	*	FIXED	CODE FIX	html.parser() silently drops data.
	html	*	CONFIRMED	-	HTMLParser.unknown_decl receives corrupted data in corner cases.
	html	*	REJECTED	-	check_for_whole_start_tag crashes with AssertionError on empty string .
	functools	*	FIXED	DOC FIX	Inconsistent behavior with docstring.
	collections	*	FIXED	CODE FIX	collections.UserString.rindex() fails to accept UserString as a sub argument.
	collections	*	DUPLICATED	-	collections.namedtuple fails to reconstruct from _asdict().
	statistics	*	FIXED	CODE FIX	statistics.stdev() and statistics.variance do not satisfy the relationship in corner cases.
	statistics	*	NOT PLANNED	-	NormalDist.overlap() returns incorrect values due to overflow.
	shlex	*	CONFIRMED	-	shlex.push_source() fails to reset lexer state, causing new stream to be ignored if called after EOF.
	types	*	FIXED	CODE FIX	DynamicClassAttribute drops explicitly provided empty docstrings.
	json	*	FIXED	DOC FIX	JSONEncoder incorrectly escapes DEL character.
tokenizer (10.3k)	tokenizers	*	FIXED	CODE FIX	BaseTokenizer.normalize method calls non-existent method.
sympy (14.2k)	ntheory	*	CONFIRMED	CODE FIX	Point addition P + (-P) fails to return point at infinity.
	ntheory	*	FIXED	CODE FIX	EllipticCurvePoint.neg returns non-canonical infinity point.
	algebra	*	FIXED	CODE FIX	Quaternion.log return nan in corner case.
	calculus	*	CONFIRMED	CODE FIX	is_monotonic misclassifies monotonic functions with stationary points.
	calculus	*	FIXED	CODE FIX	AccumBounds.intersection violates the commutativity.
	calculus	*	CONFIRMED	CODE FIX	is_decreasing crashes on single-point intervals due to ConditionSet substitution error.
	calculus	*	FIXED	CODE FIX	is_monotonic ignores discontinuities inside the interval.
	geometry	*	FIXED	CODE FIX	Plane.intersection fails to find point in plane.
	combinatorics	*	FIXED	CODE FIX	AbelianGroup.random() crashes on trivial group.
	combinatorics	*	FIXED	CODE FIX	AlternatingGroup(2) has wrong degree.
	combinatorics	*	FIXED	CODE FIX	collector.induced_pcgs drops generators at final depth.
	combinatorics	*	FIXED	CODE FIX	Cycle().list(size=0) returns [0] instead of the empty list.
	combinatorics	*	CONFIRMED	CODE FIX	PermutationGroup.coset_rank mishandle trivial groups.
pathspec (205)	pathspec	*	NOT PLANNED	-	CheckResult.include returns None instead of bool.
	pathspec	*	NOT PLANNED	-	check_tree_files and match_files return inconsistent results.
	pathspec	*	FIXED	CODE FIX	UnboundLocalError in RegexPattern.

Repo Name (Star)	Module	Issue ID	Issue State	Fix Type	Description
cryptography (7.4k)	cryptography	*	FIXED	CODE FIX	IssuingDistributionPoint.hash fails for valid inputs.
	cryptography	*	FIXED	CODE FIX	Name.rfc4514_string emits non-parseable output for trailing spaces.
pyproj (1.2k)	pyproj	*	PENDING	-	Hash/Equality contract violation.
	pyproj	*	CONFIRMED	DOC FIX	Transformer.to_json() returns None for valid CRS pair.
awkward (929)	awkward	*	CONFIRMED	CODE FIX	Crash ak.to_json loses float64 precision on round-trip.
	awkward	*	FIXED	CODE FIX	ak.moment applies weights incorrectly for n≥2 (weights exponentiated.
	awkward	*	FIXED	CODE FIX	ak.Record rejects nested dicts with scalar leaves.
	awkward	*	CONFIRMED	-	ak.ArrayBuilder.show passes Formatter to ak.Array.show, causing TypeError.
	awkward	*	PENDING	-	ByteMaskedArray.mask_as_bool returns inverted validity for valid_when=False.
	awkward	*	FIXED	CODE FIX	IndexedOptionArray.to_Indexed OptionArray64 fails when index dtype is int32.
	awkward	*	PENDING	-	Silent data corruption using tolist().
	awkward	*	FIXED	CODE FIX	ArrayBuilder.__bool__ raises TypeError when builder length is 1 .
pws-python (3.2k)	logging	*	FIXED	DOC FIX	append_context_keys drops previously appended keys.
	event_handler	*	FIXED	DOC FIX	BedrockResponse.is_json always returns True regardless of Content-Type.
	event_handler	*	FIXED	DOC FIX	UnauthorizedException raises TypeError when using documented keyword arguments.
optax (2.1k)	optax	*	FIXED	CODE FIX	scale_by_distance_over_gradients returns nan.
	optax	*	FIXED	CODE FIX	contrib.momo crashes when loss value is a Python float.
pyrsistent (2.2k)	pyrsistent	*	PENDING	-	pdeque ignores maxlen=0 during construction.
marshmallow (7.2k)	marshmallow	*	CONFIRMED	CODE FIX	fields.Constant rejects None constant values during load.
	marshmallow	*	PENDING	-	OneOf.options() emits extra pairs when labels outnumber choices.
	marshmallow	*	FIXED	CODE FIX	URL validator treats custom schemes case-sensitively.
scipy (14.3k)	interpolate	*	CONFIRMED	CODE FIX	PPoly.roots crashes with Internal error in root finding for valid cubic spline.
	cluster	*	FIXED	CODE FIX	scipy.cluster.vq.kmeans returns a distortion value from a different iteration than the final centroids.
	cluster	*	CONFIRMED	-	io.hb_write crashes on sparse matrices with zero stored entries.
	interpolate	*	FIXED	CODE FIX	RectBivariateSpline.partial_derivative rejects valid x-derivative orders when kx > ky .
boto3 (9.6k)	resources	*	NOT PLANNED	-	ResourceCollection.limit(0) yields one item.
networkx (16.4k)	networkx	*	FIXED	DOC FIX	chordal_cycle_graph returns a 6-regular multigraph with self-loops.

Properties Planning Prompt

You are an expert in **Property-Based Testing (PBT)** for Python functions.
Your Goal:
Generate verifiable properties description for a target function at a time **WITH DIRECT EVIDENCE**.
CRITICAL: Only generate properties that are DIRECTLY stated in the evidence!
Process:
1. Analyze the docstrings and code.
2. Find statements that **DIRECTLY** and **EXPLICITLY** describe behavior.
3. Generate a property that is a **DIRECT TRANSLATION** of that statement.
Guidelines:
1. Property must be a direct translation of evidence, **NOT** an assumption.
2. Evidence should be exact quotes, not paraphrases.
3. Don't generalize from examples to "always" behavior.
Output strict JSON format:
{
"property": "The property description",
"evidence": "evidence from docstring/comments that DIRECTLY states this property",
"evidence_type" : "docstring" | "code comments" | "mathematical_convention" | "type_hint",
"confidence": "high" (if directly stated) | "medium" (if strongly implied) | "low" (if inferred)
}
Here is the Target Function information:
{func_info}
Here are the current properties that have been identified for this function:
{properties}
Here are some Reference Property Heuristics (Use these as inspiration):
{exemplary_properties}
Based on the heuristics and the function info, generate new non-trivial properties description that is logically different from the **Existing Properties**.

Context Fetching Prompt

You are an expert in selecting relevant functions for **Property-Based Testing**.
Given a target function with a specific property, and a list of all available functions in the library, your task is to select functions that would be useful for testing the property.
Consider:
1. Complementary functions (e.g., encode/decode for round-trip)
2. Related operations that could be composed
3. Helper functions that create test inputs
4. Validation functions
Respond in JSON format:
{ "selected_funcs": [func1, "func2", ...] }
Note: You should only return the name of the function from the provided list, no extra information.
Target Function:
{func_info}
Property to test:
{property}
Available functions in the library:
{signature_index}
Select several most relevant functions from the available functions in the library for testing this property of target function.

PBT Generation Prompt

You are an expert Python QA engineer specializing in **Property-Based Testing** using the hypothesis library and pytest. Your task is to write a single, executable Python test function that verifies a specific property for a given target function.
Strict Output Requirements:
1. Output one Python code block.
2. The code must be clean and executable.
3. Do not generate any text outside the code block.
4. Include necessary imports (e.g., from hypothesis import given, settings, strategies as st) at the beginning of the code.
5. Infer the appropriate hypothesis.strategies based on the target function's argument types and the property description.
6. The test function name should start with test_.

Target Function:
{func_info}
Relevant Context:
{relevant_context}
Hypothesis guidelines:
{hypothesis_doc}
Property to test:
{property}
CRITICAL Hypothesis Guidelines:
* Always use @settings(max_examples=1000) decorator on **EVERY** test function for thorough testing.
* Use math.isclose() or pytest.approx() for float comparisons
* Focus on properties that reveal genuine bugs when violated
* For recursive data generation, use st.recursive().
* Keep PBT code simple. Prefer using basic strategies like st.text(), st.integers(), st.lists() directly rather than complex custom composite strategies when possible.
Example Structure:
{Some PBT examples}

Bug determination Prompt

You are an expert software testing engineer analyzing Property-Based Testing (PBT) execution results.

Goal: Determine the type of failure from three categories:
1. **Code Defect**: malformed strategies, invalid dimension assumptions, missing mocks, overly strict tolerances, nondeterministic seeding, import errors, syntax errors, or any mistake that makes the property check unreliable or impossible.
2. **Property Defect**: the property itself is incorrectly designed based on the evidence, not matching what the docstrings promises.
3. **Library Defect****: the property and evidence correctly reflects what the docstrings promises, but the actual code implementation violates it.
CRITICAL: Ground Truth is NOT CODE IMPLEMENTATION! - The docstring is the specification. The code is the implementation.
PBT Code: {code}
Execution Result: {msg}
Property to test: {property}
Property Evidence: {evidence}
Evidence Type: {evidence_type} Evidence Confidence: {confidence}
Target Function: {func_info}
Auxiliary Functions Information: {full_context}
Output strict JSO { "bug_type": "code_defect" | "property_defect" | "library_defect", "reasoning": "Succinct justification quoting the most relevant evidence and explaining your reasoning.", "fix_suggestion": "If code_defect, provide a specific fix suggestion. If property_defect, explain what's wrong with the property and how to redesign it. If library_defect, leave empty."
}

Counter-Impl Generation Prompt

You are an expert software engineer performing **ADVERSARIAL REFINEMENT** on Property-Based Tests.
Your task is to generate **ONE COUNTER-IMPLEMENTATION**: a code modification of the target function that preserves syntax but alters behavior in ways the current PBT **SHOULD** detect but **DOESN'T**.

A good counter-implementation f':
1. Is syntactically valid Python code.
2. Violates the INTENDED semantics (what the function should do).
3. Would PASS the current PBT (exposing PBT weakness).

Examples of counter-implementations:
- For a sorting function: return the list with the last element dropped (passes "ordered" check but wrong)
- For an addition function: return 0 for certain inputs (passes some tests but wrong for edge cases)
- For a contains function: always return True (passes positive tests but wrong for negatives)

Generate **EXACTLY ONE** counter-implementation with **ACTUAL EXECUTABLE CODE**.
Output strict JSON:
{
"has_counter_impl": true | false, "counter_implementation": { "description": "Brief description of what this counter-implementation does wrong", "code": "The complete modified function code (executable Python)", "what_it_violates": "What semantic property this violates" }, "reasoning": "Why this counter-implementation would pass the current PBT"
}
If the PBT is already strong enough (no counter-implementation can pass), output:
{
"has_counter_impl": false, "counter_implementation": null, "reasoning": "explanation"
}

Strengthen Property Prompt

You are an expert in Property-Based Testing (PBT) for Python functions.
Your task is to design a **STRONGER** property based on **SURVIVING COUNTER-IMPLEMENTATIONS**.

Context: we generated counter-implementations f' (modified versions of the target function).
These counter-implementations **PASSED** the current PBT, meaning the PBT is too weak to detect them.
You must design a property that would FAIL on these counter-implementations.
The new property should:
1. Specifically **FAIL** on the surviving counter-implementations (catch their incorrect behavior).
2. Be verifiable through PBT.
3. Not overlap with existing properties.

Target Function: {func_info}
Existing Properties: {existing_properties}
Current PBT: {PBT}
SURVIVING Counter-Implementations: {surviving_counter_impls}
Analyze each counter-implementation:
- What incorrect behavior does it exhibit?
- What property would catch this behavior?

Output strict JSON:
{
"property": "The stronger property description that catches the counter-implementations", "evidence": "Reference to which counter-implementation behavior this catches and why", "evidence_type": "counter_implementation"
}