Can LLMs Fix The Benchmarks? Cross-Model Property-Based Testing of Python Benchmarks

Tyler Holloway [1] **Youhui Wang** [1] **Simon Henniger** [1] **Nada Amin** [1]

Abstract

Code-generation benchmarks such as HUMANEVAL and MBPP define correctness by finite input–output tests, which cannot enforce semantic invariants. A program can pass every test yet violate its specification. We introduce X-PBT, a cross-model evaluation in which one model generates programs and an independent model generates property-based tests (PBTs) without access to the reference solution. On MBPP+ and HUMANEVAL, for Claude Sonnet 4.5, GPT-5.1, and GPT-4o-mini, a single validated property rejects 5.9% of MBPP+ and 3.1% of HUMANEVAL programs that pass the public tests. These rejections are largely complementary to the augmented example-based harness, which rejects 8.8% and 3.1% of the same programs but overlaps the property on only 19 of 83 and 3 of 21 rejected programs; adding one property increases rejections by 36% and 75%. On single-site mutants that pass the public suite but are wrong by construction, the property recovers 89% of the harness's rejection rate on MBPP+. A round-robin over generators shows these rates reflect the code, not the auditor: the induced code-quality ranking is invariant to which model writes the properties, and no model is lenient toward its own code. Generated properties also disagree with the canonical reference on 18.9% to 69.8% of tasks, and in one verified case a property automatically recovers a HUMANEVAL ground-truth defect previously repaired by hand.

1. Introduction

Benchmarks measure progress in machine learning by defining concrete tasks and standardizing evaluation procedures. In code generation, suites such as HUMANEVAL (Chen et al., 2021) and MBPP (Austin et al., 2021) function as de facto tests of program correctness. Their scores are used to compare models, justify research claims, and guide deployment decisions. Each benchmark defines correctness via an oracle; for most code-generation benchmarks, this oracle is a finite set $\{(x_i, y_i)\}_{i=1}^{n}$ of input–output examples. As generated code is deployed beyond benchmark settings, deficiencies in these example-based oracles become a direct reliability risk rather than a purely methodological concern.

Previous work shows that benchmark test suites can accept incorrect programs. Liu et al. (2023) expand the HUMANEVAL input suite by $80\times$, reducing pass@k by 19.3–28.9 percentage points across 26 models and reversing several model rankings. Yu et al. (2025) report that 28.4% of accepted SWE-BENCH LITE patches and 15.7% of accepted SWE-BENCH VERIFIED patches are false positives. Stroebl et al. (2024) show that verifier errors can correlate with model accuracy, so repeated sampling does not reliably remove evaluation bias. These methods increase test coverage but retain the same oracle form: a candidate f is accepted if $f(x_i) = y_i$ on a finite set of inputs $\{x_i\}$. However large this set becomes, it can exclude regions of the input space or fail to encode semantic invariants such as type preservation, permutation, idempotence, monotonicity, or structural validity. Passing all available example-based tests therefore does not imply that a program satisfies the intended specification.

The following MBPP examples illustrate defects that example-based suites systematically miss. In the task `similar_elements(t1, t2)`, the specification requires returning the shared elements of two tuples as a tuple, but multiple generated programs return the correct elements in a `list`. These programs pass the original tests and every held-out MBPP+ input, yet violate the semantic constraint

```
isinstance(result, tuple).
```

Augmenting the suite with additional input–output pairs exposes this defect only if the new tests explicitly check the output type.

A similar pattern appears in the task `shell_sort(my_list)`. A correct shell sort must eventually perform a pass with gap 1; otherwise, the list may remain unsorted. We construct two single-site mutants that both pass every test in the original and augmented

[1]Harvard University, Cambridge, MA. Correspondence to: Tyler Holloway <tylerholloway@g.harvard.edu>.

Preprint. July 16, 2026.

suites. The first initializes the gap as $n/3$ instead of $n/2$, yielding a gap of 0 on a two-element list and preventing any sorting pass. The second divides the gap by 3 instead of 2, so the gap sequence can skip 1 for some short lists. Both mutants leave short scrambled lists unsorted but still pass all 109 available tests. In the augmented suite, most short lists are already-sorted boolean lists, and the scrambled integer lists have lengths 7–10, for which both mutants still reach gap 1 and sort correctly. The suite therefore covers short inputs and scrambled inputs, but not short scrambled integer lists, which is precisely where these defects manifest.

We distinguish two classes of oracle failure. A *coverage failure* arises when the intended specification is clear, but the finite test suite does not exercise or encode some required behavior. A *specification failure* arises when the natural-language description or canonical reference implementation defines incorrect behavior or leaves the correct behavior underdetermined. Only coverage failures can be mitigated by adding more example-based tests.

This paper investigates whether code-generation benchmarks can be audited using executable semantic properties rather than only example-based tests. We introduce **X-PBT**, a framework in which one language model generates candidate programs and another generates properties and input generators, which are executed with Hypothesis (MacIver et al., 2019). Unlike additional input–output examples, a property can quantify over an input domain and encode relations that valid implementations must satisfy. X-PBT audits both sides of a benchmark oracle. To detect coverage failures, we retain properties that the canonical reference passes and apply them to model-generated programs that pass the benchmark's full example-based harness. To examine specification failures, we collect properties that the reference fails and treat them as candidates for manual adjudication against the task description and intended semantics.

Additionally, we introduce synthetic defects to evaluate oracle quality. Starting from each canonical solution, we apply single-site mutations to its abstract syntax tree and retain only mutants that the benchmark's existing tests accept. These mutants are incorrect by construction but observationally indistinguishable from the reference under the shipped harness. Across MBPP+ and HUMANEVAL, our study addresses five questions:

1. At what rate does an LLM-generated PBT find a falsifying input to the canonical reference solution?
2. Among LLM-generated programs that are plausible with respect to the public suite U_t, what fraction are rejected by a reference-validated PBT?
3. Over those same plausible programs, at what rate does the benchmark's augmented harness reject them, and how does that rate compare to the PBT's?
4. Over programs that are *known* to be incorrect, at what rate does the augmented harness reject them, and how does that rate compare to the PBT's?
5. Is the PBT-failure rate invariant under permutation of the code-generator and PBT-generator models?

Our results show that property-based auditing reveals semantic failures that test-set expansion alone systematically misses. A single generated property rejects 5.9% of MBPP+ and 3.1% of HUMANEVAL programs that pass the public suite, and overlaps the augmented harness on only 19 of 83 and 3 of 21 rejected programs, so adding one property increases the number of rejected programs by 36% and 75%. Against mutants that are incorrect by construction yet survive the shipped tests, one property recovers 89% of the harness's catch rate on MBPP+, and a round-robin across generators shows that this signal reflects properties of the code rather than of the auditor.

2. Related Work

2.1. Property-Based Testing

A property-based test (PBT) checks that some Boolean property $P(f(x))$ holds for every input x in a declared domain D (Claessen & Hughes, 2000). The runner samples random inputs from D and reports a counterexample if any sample falsifies P (Figure 1). Unit tests are the special case in which D is finite and explicitly enumerated, and P is the equality $f(x_i) = y_i$. HYPOTHESIS (MacIver et al., 2019) extended the QuickCheck idiom to Python and is the test runner X-PBT uses; when a property fails, its automatic *shrinker* progressively simplifies the failing input until no further simplification still triggers the failure, surfacing a minimal counterexample.

Ravi & Coblenz (2025) report, in a corpus study of 426 Python projects, that an individual PBT detects roughly 50× more mutations than an individual unit test on the same code, with PBTs that check for exceptions, collection membership, or types being over 19× more effective than other PBT categories; 76% of mutations detectable by a PBT are found within the first 20 random inputs. In a qualitative study of 30 practitioners at Jane Street, where PBT has been used in production OCaml for two decades, developers reach for PBT opportunistically when an obvious property is at hand (Goldstein et al., 2024); this is the pattern X-PBT automates by having an LLM propose properties from a function specification.

Recent work studies how well LLMs synthesize property-based tests (PBTs) and how those tests interact with example-based oracles. Vikram et al. (2023) find that LLMs generate correct PBTs for only 21% of properties extractable from function documentation. Allamanis & Yin (2025) re-

Same task, two test styles. Both fragments check `similar_elements(t1, t2)`, which should return the shared elements of two lists as a tuple.

Unit test

```
def test_similar_elements():
    # three explicit (input, output)
    ↪ pairs
    assert similar_elements(
        [1, 2, 3], [3, 4, 5]
    ) == (3,)
    assert similar_elements(
        [1, 2], [2, 3]
    ) == (2,)
    assert similar_elements(
        [], []
    ) == ()
```

Three concrete pairs. Whatever holds outside these three inputs is unconstrained.

Property-based test

```
@given(
    t1=st.lists(st.integers()),
    t2=st.lists(st.integers()),
)
def test_similar_elements(t1, t2):
    r = similar_elements(t1, t2)
    assert isinstance(r, tuple)
    assert all(
        x in t1 and x in t2 for x in
        ↪ r
    )
    assert len(set(r)) == len(r)
```

Three universal claims (return type, membership, no duplicates). Hypothesis samples random (t_1, t_2) and reports any counterexample.

Figure 1. Unit tests assert $f(x_i) = y_i$ for finitely many explicit pairs; property-based tests assert $\forall x \in D : P(f(x))$ and let a runner sample x. They do not provide formal guarantees that the property holds; instead, they run as many examples as possible to come close.

port a 28.2% unit-pass–but–PBT-fail rate for programs generated by Gemini Flash 2.0 on LBPP, and Ma et al. (2025) observe that an analogous gap widens from 20% on medium-difficulty problems to 40% on hard problems on a separate benchmark. Tanaka et al. (2025) show that PBTs and example tests are complementary: each individually detects 68.75% of bugs in their study, while using both increases detection to 81.25%. He et al. (2025) report that repair feedback grounded in natural-language specifications yields a 75.9% repair success rate, compared to 46.6% for feedback grounded only in public unit tests. Bose et al. (2025) report PBT-failure rates of 43%–52% for StarCoder and CodeLlama on HumanEval and MBPP+.

2.2. Code Generation Benchmarks and Evaluation Reliability

Liu et al. (2023) demonstrate that hand-written test suites in benchmarks such as HumanEval and MBPP are insufficient. Their EvalPlus methodology augments inputs by 80× via mutation; on the resulting MBPP+ benchmark, pass@k falls by 19.3–28.9 percentage points across 26 LLMs and several model rankings reverse. Liu et al. (2025) extend this diagnosis: HumanEval and MBPP achieve $> 99.16\%$ statement coverage but only 74.39% branch coverage and 87.69% mutation score, with mutation scores degrading on higher-complexity tasks. The augmented harness produced by EvalPlus remains an example-based oracle: the additional tests still take the form $f(x_i) = y_i$ and cannot express invariants such as return type, ordering preservation, or idempotence over an unbounded input domain.

The test-inadequacy problem extends to repository-scale and more challenging benchmarks. Yu et al. (2025) find that 28.4% of patches in SWE-Bench Lite and 15.7% in SWE-Bench Verified are erroneously labeled as passing because the held-out test suite does not constrain patch behavior tightly enough. Zhuo et al. (2025) attack a complementary axis with 1,140 Python tasks requiring multi-library function calls; GPT-4o, the strongest of 60 models they evaluate, solves only 60% of BigCodeBench-Complete versus 97% for human experts. Jain et al. (2025) address training-data contamination by sourcing problems from competitive platforms after each model's training cutoff, and observe that fine-tuned open-source models that perform well on HumanEval perform substantially worse on LiveCodeBench.

Among LLM-based test-generation systems, Chen et al. (2022) propose CodeT, in which a single model generates both candidate programs and candidate tests and selects the largest mutually consistent set. The same-model design is vulnerable to a blind spot shared between code and tests: a model that writes incorrect code for a task is plausibly the same model that writes a test that accepts that incorrect code. X-PBT's cross-model design (Sec. 4, RQ5) targets this confound directly.

3. Methodology

3.1. Problem Setup

We consider Python function-synthesis tasks. A task t consists of a natural-language prompt, a public test suite U_t, and a reference implementation r_t. The reference is the benchmark's canonical solution, typically human-written and treated as ground truth. A candidate program p is a Python function with the signature of r_t. Let D_t denote the set of inputs the task's specification admits. A candidate p is *plausible* for t if it passes every test in U_t. We say p is *overfit* to U_t if it is plausible but there exists some admissible input $x \in D_t$ on which $p(x)$ deviates from $r_t(x)$, even though no test in U_t reveals this discrepancy.

A property-based test (PBT) for task t is a function `test_pbt(fn)` where `fn` has the signature of the reference implementation r_t. It never calls or compares against r_t; the reference is not in scope at run time. Instead, the PBT defines Hypothesis strategies over a subset of the input domain D_t and asserts output invariants (for example, type and shape constraints, boundary behavior, or relations between inputs and outputs) that should hold for all admissible inputs. When Hypothesis happens to sample an input x that falsifies an asserted invariant, it reports x as a counterexample and the PBT fails. In our experiment, we ask whether a language model can synthesize such a PBT that rejects overfit programs, treating the PBT as a partial oracle that can refute candidates but whose satisfaction does not guarantee correctness.

The pipeline uses two language models in disjoint roles: one generates code and the other synthesizes properties. For each task t, the code model M_c maps the prompt to candidate programs p; we retain those that pass the benchmark's public tests and thus qualify as plausible overfits. The property model M_p maps the same task to a PBT. Both M_c and M_p are off-the-shelf instruction-tuned LLMs accessed via provider APIs, with no additional training or fine-tuning. We index every result by the pair (M_c, M_p), which allows us to test whether a given property model is especially effective at exposing errors in code produced by its paired code model.

We also construct non-model overfit candidates by applying single-point edits to the syntax tree of r_t, such as swapping comparison or arithmetic operators, and retaining only mutants that still satisfy the public tests. These mutants are incorrect by construction and serve as a control.

3.2. Benchmarks

We evaluate on MBPP+ (378 tasks) and HumanEval (164 tasks), two Python function-synthesis benchmarks with a canonical reference solution and an augmented test suite. Both are function-level: each task specifies a single named entry point, so a property over that function is well-defined. Each benchmark provides an original test suite and an EvalPlus-augmented suite. In MBPP+, both appear in one dataset: `test_list` contains the three original MBPP assertions and `test` contains the augmented harness. HumanEval ships only its original `check()` assertions; the augmented suite is provided in a separate `humanevalplus` dataset that we load and align by task identifier.

Benchmark	Original (weak)	Augmented (strong)
MBPP+ (378)	`test_list`	`test`
HumanEval (164)	`check()`	`humanevalplus`

For each task t, we use the original benchmark suite as the public suite U_t. It gates both model-generated candidates (§3.1) and syntax-level mutants (§3.3.4): only programs that pass U_t enter our study. We do not gate on the augmented suite U_t^{aug}, because that would exclude the programs of interest, which satisfy U_t but violate U_t^{aug}. Formally, we focus on programs p such that

$$p \in \text{Pass}(U_t) \ \wedge \ p \in \text{Fail}(U_t^{\text{aug}}).$$

We treat U_t^{aug} as a baseline overfitting detector: a stronger human-authored harness that flags programs missed by U_t. Our analysis measures how often model-generated properties identify the same overfit programs as U_t^{aug} (§3.4).

3.3. Pipeline

The pipeline consists of five stages operating over a single content-addressed store. Every artifact is keyed by a hash of its identity fields, including the generating model, which makes all stages idempotent and re-entrant. Re-running a stage skips artifacts already present.

1. **Seed.** Seed ingests each benchmark task, recording the prompt, reference solution, entry-point name, and the benchmark's public test suite. No model is invoked in this stage.
2. **Generate programs (M_c).** The code model M_c writes n candidate solutions per task. Identical candidates are stored once, with separate draw records so multiplicity is preserved without duplicating code. This stage performs no grading.
3. **Generate properties (M_p).** The property model M_p writes one `test_pbt(fn)` module per task from a specified prompt variant (§3.3.1). The validity of each generated property is unknown at this stage and remains unset.
4. **Reference-validate.** Run each property against its task's reference solution and mark it as valid or invalid. Exclude properties that the reference fails from

all downstream stages. This is the paper's key control and the last stage shared by both measurement routes: stage 5 consumes these validated properties for real programs, and the mutation procedure of §3.3.4 consumes the same set for synthetic programs.

5. **Evaluate.** Score the within-task cross-product of programs and test suites (§3.3.3), memoizing each verdict on the (program, suite) pair so no already-scored pair is re-executed. Reporting then reads only these stored verdicts: it applies the eligibility rule of §3.4 (the program passes U_t and the property is valid for that task) and aggregates the results into the catch rate and the 3×3 cross-model matrix.

3.3.1. GENERATION

We use three models in both roles: Claude Sonnet 4.5, GPT-5.1, and GPT-4o-mini, yielding a full 3×3 design over (M_c, M_p). Programs and properties are generated once per model and then scored over the complete cross-product. All models are accessed as frozen APIs with no additional training or fine-tuning. For each model and task, we draw $n = 1$ program and $n = 1$ property with a 2048-token limit and use the provider's default decoding temperature (approximately 1.0). This samples each model's default output distribution rather than enforcing a uniform temperature; some model families ignore explicit temperature settings, so a global value is not enforceable.

The code model M_c receives the task prompt and returns a complete Python solution in a single fenced block, from which we extract the first code block. The property model M_p receives a prompt template and returns a module defining `test_pbt(fn)`. We use two templates that differ only in whether they include the reference solution (REF vs. NOREF); both appear verbatim in Appendix A. All results in §4 use NOREF, in which the property model never sees the solution against which its property is validated.

3.3.2. REFERENCE VALIDATION

A generated property is *valid* for task t if the reference solution r_t passes it. A property that r_t fails may impose constraints stronger than the task specification or sample inputs outside the function's precondition, or it may be correct while the reference is wrong; executing it does not distinguish these cases. Such a property cannot support attribution: if it rejects both r_t and a candidate p, we cannot ascribe the failure specifically to p. We therefore discard these properties and exclude them from both the numerator and denominator of all subsequent metrics, rather than count them as failures to catch. This choice is conservative, since discarded properties may in fact be sound, so every reported catch rate is a lower bound. The fraction of properties that pass reference validation is itself a measured outcome, reported per model in §4.

3.3.3. SCORING

Properties execute in an isolated subprocess in their own process group. Each run has a 120-second wall-clock timeout and a 2 GiB address-space limit. The Hypothesis profile disables the deadline, disables the example database, restricts execution to the explicit and generate phases, and derandomizes input generation so verdicts are reproducible. The candidate is bound to the property's function-under-test parameter before the property module executes, preventing a module-level implementation inside the property from being tested instead of the candidate. The profile requests 100 examples per property, but the prompts instruct the model to emit its own `@settings` decorator, which overrides the profile when present. On MBPP+, approximately 95% of properties specify 100 examples, with the remainder specifying 200 or 10, so the example budget is model-influenced while all other profile settings are inherited.

3.3.4. MUTATION

Stage 5 evaluates properties on model-written programs whose correctness is unknown. To obtain a population of programs that are *known* to be wrong, we also construct synthetic overfit programs by mutation. This mutation pipeline runs after stage 4, consumes the same set of validated properties, and is independent of stage 5. For each task, we apply single-point mutations to the reference solution's abstract syntax tree, so each mutant differs from the reference at exactly one node. We deduplicate by source text, discard mutants identical to the original, and retain at most 20 mutants per task in traversal order.

A mutant is retained only if it passes the task's *weak* suite: `test_list` for MBPP+ and the original `check()` assertions for HumanEval. Any surviving mutant is therefore, by construction, a program that is wrong yet indistinguishable from the reference under the benchmark's original tests. Such a program is a synthetic overfit program constructed without a language model. Over this gated set we race two detectors: the benchmark's *strong* suite and each valid property. We do not attempt to identify semantically equivalent mutants. Consequently, a mutation that leaves behavior unchanged may still be retained, so both measured catch rates are lower bounds.

3.4. Analysis

A property *catches* a program only when Hypothesis reports a falsifying example: a concrete input on which the program violates an asserted invariant. Other kinds of failure are ambiguous. A property that raises before Hypothesis reports anything, a run killed at the wall-clock limit, and a worker that dies without reporting all surface as failures

but may reflect infrastructure or harness errors rather than defects in the program under test. Counting these as catches would inflate rates by the infrastructure's failure rate, so we classify failing runs and admit only reported falsifying examples into the numerator. Runs that never reach a verdict, time out, or die are excluded and reported separately (§4), where they account for 5.6% to 15.4% of apparent reference disagreements.

The mutation route is an exception. It scores only reference-validated properties, so each property terminates on the reference over the same derandomized inputs. A mutant that exceeds the wall-clock limit therefore differs behaviorally from the reference on inputs the reference handled, which we treat as a defect rather than an infrastructure artifact, and we admit it. Only a worker dying without reporting remains ambiguous, and we report that count separately.

Our unit of analysis is the (program, property) pair within a task. A pair is *eligible* if the program passes the task's public suite U_t and the property is valid for that task. Eligibility encodes the question: given a program the benchmark accepts and a property we have verified against the reference, does the property reject the program?

$$\text{overfit-catch rate} = \frac{|\{(p, \pi) \in \mathcal{E} : \pi \text{ rejects } p\}|}{|\mathcal{E}|}$$

where $\mathcal{E}$ is the eligible set. Properties failing validation are absent from $\mathcal{E}$ and contribute to neither term. We report this rate overall and as a 3×3 matrix over (M_c, M_p). The matrix makes the self-bias question testable: if the diagonal (a model's properties judging its own programs) differs systematically from the off-diagonal, then a property model's apparent strength partly reflects shared inductive bias with the code model, and properties from one model may not transfer as benchmark augmentations. For the mutation route, we report, per task and per property model, the fraction of gated mutants caught by the strong suite and the fraction caught by the generated property, over the same mutant set.

4. Results

RQ1: At what rate does an LLM-generated PBT find a falsifying input to the canonical reference solution?

For each (GENERATOR, BENCHMARK) configuration, we measure the fraction of tasks on which an LLM-generated PBT finds a falsifying input to the canonical reference. Across the six configurations, this fraction ranges from 18.9% to 69.8%.

The generators have the same ordering on both benchmarks: Claude Sonnet 4.5 attains the lowest rates (41.8% on MBPP+, 18.9% on HUMANEVAL), followed by GPT-5.1 (52.4%, 26.2%) and GPT-4o-mini (69.8%, 51.8%). These PBTs are generated without including the reference solution in the prompt. The reported rates are corrected for non-semantic failures: re-running all 845 apparent reference failures with an instrumented runner shows that 779 (92.2%) produce a falsifying input, while 35 never reach a verdict because of errors such as unresolvable imports or `NameErrors` in `test_pbt`, and 31 time out or lose their workers. The correction is largest where the raw rate is highest: all 158 apparent failures from Claude on MBPP+ are genuine, whereas GPT-4o-mini's rates drop by 6.4 percentage points on MBPP+ and 9.8 on HUMANEVAL. HUMANEVAL requires more correction than MBPP+ (15.4% vs. 5.6% of apparent failures).

A falsifying input establishes a disagreement between the property and the reference but not which is incorrect. In one instance this ambiguity can be resolved from the task specification. Claude's property for HUMANEVAL/124, `valid_date`, rejects the reference on `11-30-0056`, a valid date that the reference misclassifies; it also rejects `12-31-1999` and `01-31-2020`. The cause is operator precedence: `if month == 2 and day < 1 or day > 29` parses as `(month == 2 and day < 1) or (day > 29)`, so the reference rejects every day after the 29th, regardless of month. Liu et al. (2023) report the same defect in their Figure 4 as one of 18 incorrect HUMANEVAL ground truths they identified and repaired by hand. Our property recovers this defect automatically without seeing the reference solution, whereas the properties generated by GPT-4o-mini and GPT-5.1 do not detect it.

RQ2: Among LLM-generated programs that are plausible with respect to the public suite U_t, what fraction are rejected by a reference-validated PBT?

We measure the fraction of LLM-generated programs that both pass the public suite U_t and are rejected by at least one reference-validated PBT. We restrict to programs that pass U_t and to tasks with at least one validated property. On MBPP+, the initial 1134 generated programs shrink to 898 that pass U_t and then to 695 whose task has a validated property; on HUMANEVAL, the corresponding counts are 492, 439, and 393. The first filter removes programs the benchmark already rejects. The second removes tasks for which every generated property is refuted by the canonical reference, leaving no validated property with which to evaluate the surviving programs; these tasks are part of RQ1 and their exclusion makes the estimate conservative.

Over this filtered set, 5.9% of programs on MBPP+ (41/695) and 3.1% on HUMANEVAL (12/393) are rejected by a validated property. These programs pass the public tests but fail a property that the reference satisfies and that was generated without exposing the reference. Unlike RQ1, these rates require almost no correction for infrastructure

failures: on MBPP+, 129 of 131 apparent property failures are genuine refutations, 2 are crashes, and none are timeouts; on HUMANEVAL, 49 of 50 are genuine. This difference arises because RQ2 includes only properties that have already executed successfully against the reference, filtering out most properties prone to crashes or timeouts and thereby yielding a cleaner measurement.

RQ3: Over those same plausible programs, at what rate does the benchmark's augmented harness reject them, and how does that rate compare to the PBT's?

We evaluate the same population as in RQ2: 695 MBPP+ and 393 HUMANEVAL programs that pass U_t and have at least one validated property. For each task, EvalPlus supplies 100–250 additional test inputs, and labels each input with the output produced by the canonical reference implementation. Because these checks are simple equality assertions, their verdicts require no correction for crashes or timeouts. The augmented harness rejects 8.8% of MBPP+ programs (61/695), versus 5.9% rejected by a single generated property, and rejects 3.1% of HUMANEVAL programs (12/393), matching the property's 3.1% rate (Table 1). On HUMANEVAL, one generated property therefore rejects as many benchmark-passing programs as the augmented harness; on MBPP+, it reaches about two thirds of the harness's rejection rate.

	MBPP+	HUMANEVAL
Programs (N)	695	393
Rejected by harness	61 (8.8%)	12 (3.1%)
Rejected by PBT	41 (5.9%)	12 (3.1%)
of the programs either rejects:		
both	19	3
PBT only	22	9
harness only	42	9

Table 1. The augmented harness and a single generated property, over the same programs. N counts programs that pass the public suite and have at least one reference-validated property. The harness is EvalPlus's held-out input suite and its verdicts are exact; the PBT column counts genuine refutations only. The lower block partitions the programs either oracle rejects: the overlap is 19 of 83 and 3 of 21, so neither oracle approximates the other. Properties are generated without the reference in the prompt.

These aggregate rates, however, obscure how little the two oracles overlap. Of the 83 MBPP+ programs rejected by at least one oracle, only 19 are rejected by both; on HUMANEVAL, only 3 of 21 are rejected by both. Adding one generated property to the augmented harness increases the number of rejected programs from 61 to 83 on MBPP+ ($+36\%$) and from 12 to 21 on HUMANEVAL ($+75\%$). The oracles detect mostly disjoint failures: the property alone rejects 22 MBPP+ and 9 HUMANEVAL programs that pass all held-out EvalPlus test

RQ4: Over programs that are *known* to be incorrect, at what rate does the augmented harness reject them, and how does that rate compare to the PBT's?

The augmented harness rejects a larger fraction of known-incorrect programs than the PBT on both benchmarks, with a small margin on MBPP+ and a large margin on HUMANEVAL. For each task, we mutate the reference implementation at a single site and retain only mutants that still pass the public suite, yielding programs that are wrong by construction but accepted by the benchmark's own tests. This process yields 269 mutant–property pairs over 58 MBPP+ tasks and 171 over 40 HUMANEVAL tasks, and both oracles are evaluated on the same mutants. The harness rejects 62.8% of MBPP+ mutants versus the property's 55.8%, and 59.1% of HUMANEVAL mutants versus 26.3% (Table 3, appendix). As in RQ2, these verdicts require almost no correction: 150 of 152 MBPP+ catches and 45 of 46 HUMANEVAL catches are genuine refutations.

Recovery is the only rate directly comparable across generators. The harness's raw rejection rate ranges from 58.5% to 69.4% on MBPP+ even though it does not depend on any prompt, because a task appears in a generator's row only if that model produced a validated property for it. Within a row the mutants are identical, and there the recovery on MBPP+ is stable at 87%, 88%, and 92% across three generators. On eight MBPP+ rows the property dominates, rejecting at least one mutant that the harness never rejects; these include MBPP/428, the shell-sort task from the introduction, where Claude and GPT-4o-mini each kill two of three surviving mutants that all 109 shipped tests accept. The converse holds on four rows, where the harness rejects mutants that no property refutes.

On HUMANEVAL this pattern reverses: the property wins on two rows, while the harness wins on 19 rows (35 mutants). We hypothesize that this direction, opposite to RQ3, reflects the mutation process. A single-site operator swap perturbs arithmetic in an otherwise correct program, and increasing the number of concrete inputs allows the harness to detect many such arithmetic errors by brute force. In contrast, defects that a property could expose but the harness structurally cannot, such as returning a `list` where the specification requires a `tuple`, are absent from the mutant space defined by our operators. RQ4's population is therefore biased toward the harness by construction, which makes the 89% recovery on MBPP+ particularly informative.

RQ5: Is the PBT-failure rate invariant under permutation of the code-generator and PBT-generator models?

The round-robin experiment shows no evidence that PBT failure rates depend on whether the code and properties come from the same model. We evaluate every code genera-

tor against every property generator (Table 2).

	PBT generator		
Code generator	Claude	GPT-4o-mini	GPT-5.1
Claude Sonnet 4.5	0.9	5.6	3.8
GPT-5.1	2.0	8.2	4.5
GPT-4o-mini	2.1	10.6	5.2

Table 2. Percentage of plausible (program, validated property) pairs the property refutes, on MBPP+. Read down a column: the property generator is fixed, so the comparison isolates the code, and the ranking Claude < GPT-5.1 < GPT-4o-mini holds in every column. Diagonal entries, where one model wrote both, show no systematic advantage. Rates are *not* comparable across columns; see text.

Read down a column (fixed property generator): the induced ranking of code generators is identical in all three columns. Claude is refuted least, then GPT-5.1, then GPT-4o-mini ($0.9 < 2.0 < 2.1$; $5.6 < 8.2 < 10.6$; $3.8 < 4.5 < 5.2$). Every independently generated property set therefore induces the same ordering over code generators, so the measurement reflects properties of the code rather than of the auditor.

Along the diagonal, we see no consistent self-advantage. Claude's properties refute Claude's code least in that column (0.9 versus 2.0 and 2.1), which is compatible with either self-leniency or higher code quality, but the off-diagonal columns also rank Claude best, so better code suffices to explain this entry. The other diagonals go in the opposite directions: GPT-4o-mini's properties refute GPT-4o-mini's code most in its column (10.6 versus 5.6 and 8.2), and GPT-5.1's diagonal entry (4.5) lies near its column mean ($3.8, 5.2$). One diagonal below its column, one above, and one near the mean is consistent with no systematic self-calibration effect. Two caveats apply: rates must not be compared across columns, since GPT-4o-mini contributes only 90 validated properties to Claude's 220, so each column reflects a different task mix; and on HUMANEVAL, each cell contains between 0 and 3 refutations, too sparse to resolve an ordering, so we do not report that matrix.

5. Discussion

Limitations

Both oracles inherit the reference. The augmented harness takes its expected outputs from the canonical solution, and we admit only properties that solution passes. Both oracles are therefore anchored to the same implementation: where the reference is wrong, they are wrong together, and RQ3–RQ4 cannot detect such shared defects. The validity filter reinforces this coupling: on a task with an incorrect reference, any property that correctly exposes the bug is refuted and discarded as `valid=0`, so it preferentially removes the most informative properties. RQ1's 18.9–69.8% disagreement rates are computed before this filter and are the only results that do not rely on trusting the reference.

Disagreement is unadjudicated. Of the 779 refutations behind RQ1 we attribute exactly one, HUMANEVAL/124, and only because Liu et al. (2023) independently identified and repaired it. The remainder may reflect reference bugs, over-strong properties, or ambiguous specifications; we report a rate and an existence proof, not a classification. Likewise, a property refuting a program in RQ2 is evidence of semantic disagreement, not a proof that the program is incorrect.

The mutation population favors the harness. RQ4 mutates the reference at a single site, which primarily perturbs arithmetic and does not generate the structural defects that properties can uniquely detect, such as returning a `list` where the specification requires a `tuple`. That class of error is absent from the mutant population by construction, which biases RQ4 toward the example suite and makes MBPP+'s 89% recovery the more notable figure. We also do not detect equivalent mutants, so both oracles' rejection rates in RQ4 are lower bounds.

Cross-generator rates are not comparable. In RQ5, a task appears in the matrix for a given generator only if that model produced at least one validated property for it, so each generator contributes a different, self-selected task mix. Only within-column comparisons and within-row recovery are sound. On HUMANEVAL, the matrix is too sparse (0–3 refutations per cell) to resolve any ordering, so we do not report it. More broadly, our evidence covers two Python function-synthesis benchmarks and three models, and need not extend to repository-scale code or other languages.

Future Work

Reference dependence is the primary target for improvement. Validating each property against several independent implementations, rather than a single canonical solution, would let properties survive individual reference defects and turn the validity filter into a cross-check. Liu et al. (2023)'s 18 manually identified HUMANEVAL ground-truth defects provide a human-labeled baseline against which the refutation population can be adjudicated automatically.

Richer, defect-diverse mutation operators would remove RQ4's structural bias and let the two oracle forms be raced on the failures each is built to catch. Scaling beyond two benchmarks and three models to larger suites and repository-level code would test whether the observed specification-compliance gap is benchmark-specific or characteristic of LLM-generated code. The end goal is to move from auditing oracles to repairing them: using validated properties to strengthen a benchmark's oracle in place and automating the hand labor Liu et al. (2023) performed by inspection.

Acknowledgements

Funding provided by the Harvard Data Science Initiative. This research is partially supported by AWS Credits via an Amazon Research Award and Google Cloud Research Credits. Simon Henniger's work was supported by the Harvard John A. Paulson SEAS Prize Fellowship and the German Academic Fellowship Organization, funded by the German Federal Ministry for Economic Affairs and Energy.

Impact Statement

As LLM-generated code enters production at scale, silently incorrect programs pose risks that extend beyond ordinary software bugs. Code that passes its test suite but violates its specification is, in a precise sense, misaligned: it behaves as the evaluation signal rewards rather than as the developer intended. In safety-critical and financial systems, such failures have already caused fatalities and billions of dollars in losses. Our work demonstrates that property-based testing can surface these specification-compliance gaps, providing a lightweight verification layer that does not require model retraining. More broadly, methods that detect divergence between intended and observed behavior in model outputs contribute to the broader goal of AI alignment—ensuring that capable systems do what we ask of them, not merely what we measure.

References

Allamanis, M. and Yin, P. Disproving Program Equivalence with LLMs. *arXiv.org*, 2025. doi: 10.48550/ARXIV.2502.18473. URL https://arxiv.org/abs/2502.18473.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. In *arXiv preprint arXiv:2108.07732*, 2021.

Bose, S. et al. How effective are llm-generated pbts for detecting bugs? *arXiv preprint*, 2025.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. In *arXiv preprint arXiv:2107.03374*, 2021.

Claessen, K. and Hughes, J. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 268–279, 2000.

Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., and Head, A. Property-Based Testing in Practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13. ACM, apr 12 2024. doi: 10.1145/3597503.3639581. URL http://dx.doi.org/10.1145/3597503.3639581.

He, L., Chen, Z., Zhang, Z., Gao, X., and Sheng, L. Effective LLM Code Refinement via Property-Oriented and Structurally Minimal Feedback. *arXiv.org*, 2025. doi: 10.48550/ARXIV.2506.18315. URL https://arxiv.org/abs/2506.18315.

Jain, N., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

Liu, X., Sun, X., Bo, L., Hu, Y., Liu, X., and Ye, Z. Evaluating the Test Adequacy of Benchmarks for LLMs on Code Generation. *Journal of Software: Evolution and Process*, 37(7), jun 25 2025. ISSN 2047-7473. doi: 10.1002/smr.70034. URL http://dx.doi.org/10.1002/smr.70034.

Ma, Z., Zhang, T., Cao, M., Liu, J., Zhang, W., Luo, M., Zhang, S., and Chen, K. Rethinking Verification for LLM Code Generation: From Generation to Testing. *arXiv.org*, 2025. doi: 10.48550/ARXIV.2507.06920. URL https://arxiv.org/abs/2507.06920.

MacIver, D. R., Hatfield-Dodds, Z., et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

Ravi, S. and Coblenz, M. An Empirical Evaluation of Property-Based Testing in Python. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3897–3923, oct 9 2025. ISSN 2475-1421. doi: 10.1145/3764068. URL http://dx.doi.org/10.1145/3764068.

Stroebl, B., Kapoor, S., and Narayanan, A. The Limits of Inference Scaling Through Resampling. *arXiv.org*, 2024. doi: 10.48550/ARXIV.2411.17501. URL https://arxiv.org/abs/2411.17501.

Tanaka, H., Tanaka, H., Shimari, K., and Matsumoto, K. Understanding the Characteristics of LLM-Generated Property-Based Tests in Exploring Edge

Cases. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pp. 11–18. IEEE, nov 19 2025. doi: 10.1109/aiware69974.2025.00009. URL http://dx.doi.org/10.1109/AIware69974.2025.00009.

Vikram, V., Lemieux, C., Sunshine, J., and Padhye, R. Can Large Language Models Write Good Property-Based Tests? *arXiv.org*, 2023. doi: 10.48550/ARXIV.2307.04346. URL https://arxiv.org/abs/2307.04346.

Yu, B., Zhu, Y., He, P., and Kang, D. Utboost: Rigorous evaluation of coding agents on swe-bench. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3762–3774, 2025.

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pp. 66602–66656, 2025.

A. PBT Generation Prompts

We generate property-based tests with one of two templates, differing in whether the canonical reference solution is shown. All results reported in §4 use the NOREF variant (A.2), in which the property model never sees the reference it will be validated against.

We do not set a decoding temperature: each provider's default (approximately 1.0) applies, so these are single draws from each model's natural output distribution rather than greedy decoding. There is no format-conformance retry. We extract the first fenced code block from the response and store it as-is; a response that does not define a runnable `test_pbt(fn)` becomes a property that fails to execute, and is reported as such in §4 rather than being silently regenerated.

The placeholders `{prompt}`, `{code}`, and `{unit_tests}` are filled per task with the docstring, the canonical reference solution, and up to three of the task's public unit tests. The two templates do not use the same placeholders: A.1 interpolates all three, while A.2 interpolates only `{prompt}`.

A.1 With-reference

```
You are writing a property-based test using the Python Hypothesis library for a function
    described below.

## Task
{prompt}

## Reference solution
```python
{code}
```

Unit tests (for reference only - do NOT replicate these as properties)
{unit_tests}

Instructions
Write a Hypothesis `@given` test that:
1. Generates valid inputs using `hypothesis.strategies` (st)
2. Asserts PROPERTIES of the output - invariants that must hold for ALL valid inputs
3. Does NOT assert specific expected values (that would just re-implement a unit test)
4. Covers at least 2 distinct properties if possible

Good properties to consider:
- Output type and shape (result is always a list, length is non-negative, etc.)
- Boundary behaviour (empty input -> specific output)
- Relationship between input and output (all elements of output appear in input)
- Monotonicity (larger input -> larger/smaller output)
- Commutativity where applicable

Idempotency (`f(f(x)) == f(x)`) holds only for some functions, so assert it only when the
operation is genuinely idempotent AND the output is a valid input to the function. Many
    are not:
squaring is not idempotent, and a function returning a bool or int from a string cannot be
re-applied; asserting idempotency there would fail the correct solution.

Your response must be a single Python code block containing:
- All necessary imports (hypothesis, hypothesis.strategies as st, etc.)
- A function named `test_pbt(fn)` that takes the function under test as its only argument
- Inside `test_pbt`, define and immediately call a `@given(...)` inner function

Format:
```python
from hypothesis import given, settings, assume
from hypothesis import strategies as st

def test_pbt(fn):
   @given(...)
```
```

```
   @settings(max_examples=100)
   def _test(...):
      result = fn(...)
      # assert properties here
   _test()
```

Only output the code block. Do not explain.

A.2 No-reference (used for the reported results)

This variant withholds the reference solution and the unit tests, and suggests seven named property patterns rather than the invariant categories of A.1.

```
You are writing a property-based test using the Python Hypothesis library for a function
    described below.

## Task
{prompt}

## Instructions
Write a Hypothesis `@given` test that:
1. Generates valid inputs using `hypothesis.strategies` (st)
2. Draws inputs that actually exercise the function: when arguments are related, draw them
     together (e.g. pick a search character from the string, not independently), using `st.
    data()` if needed
3. Asserts PROPERTIES of the output: invariants that must hold for ALL valid inputs
4. Calls the provided `fn` as the function under test; do not redefine that function at
    the top level

Good properties to consider:
1. The test oracle: implement a simple/brute-force version inside the test (a helper, not
    a top-level redefinition of the function) and assert `fn` agrees with it
2. There and back again: an inverse round-trips: decode(encode(x)) == x
3. Different paths, same destination: two routes give the same result
4. Some things never change: an invariant is preserved (permutation, sum, bounds)
5. Solve a smaller problem first: relate f(x) to f on a smaller input
6. Hard to prove, easy to verify: the result satisfies its defining condition
7. The more things change, the more they stay the same: idempotency: f(f(x)) == f(x)

Your response must be a single Python code block containing:
- All necessary imports (hypothesis, hypothesis.strategies as st, etc.)
- A function named `test_pbt(fn)` that takes the function under test as its only argument
- Inside `test_pbt`, define and immediately call a `@given(...)` inner function

Format:
```python
from hypothesis import given
from hypothesis import strategies as st

def test_pbt(fn):
   @given(...)
   def _test(...):
      result = fn(...)
      # assert properties here
   _test()
```

Only output the code block. Do not explain.
```

B. Mutation Catch Rate by Generator

Table 3 decomposes RQ4's aggregate figures by property generator. The aggregate rates and the *Recovery* comparison are discussed in §4 (RQ4); only the per-generator breakdown is deferred here.

PBT generator	Mutants	Harness	PBT	Recovery
MBPP+				
Claude	130	76 (58.5%)	66 (50.8%)	87%
GPT-4o-mini	54	34 (63.0%)	30 (55.6%)	88%
GPT-5.1	85	59 (69.4%)	54 (63.5%)	92%
all	269	169 (62.8%)	150 (55.8%)	**89**%
HUMANEVAL				
Claude	76	36 (47.4%)	23 (30.3%)	64%
GPT-4o-mini	38	29 (76.3%)	8 (21.1%)	28%
GPT-5.1	57	36 (63.2%)	14 (24.6%)	39%
all	171	101 (59.1%)	45 (26.3%)	**45**%

Table 3. Mutants surviving the public suite, and how often each oracle rejects them. *Mutants* counts mutant-property pairs. *Harness* is EvalPlus's augmented suite; *PBT* counts genuine refutations only. *Recovery* is PBT / harness, and is the only figure comparable across rows: a task enters a row only when that model wrote a validated property for it, so the rows cover different mutants, which is why the harness's own rate varies by generator despite being a fixed suite. Within a row both oracles face identical mutants.

C. Additional Worked Example: HumanEval/32

Task (HumanEval/32, `find_zero`): return x such that `poly(xs, x) == 0`.
Preconditions: (i) `len(xs)` is even; (ii) leading coefficient `xs[-1]` is non-zero.

Reference solution (bisection):

```
def find_zero(xs):
    begin, end = -1.0, 1.0
    while poly(xs, begin) * poly(xs, end) > 0:
        begin *= 2.0; end *= 2.0        # widen until sign-change
    while end - begin > 1e-10:
        c = (begin + end) / 2
        if poly(xs, c) * poly(xs, begin) > 0:
            begin = c
        else:
            end = c
    return begin
```

LLM-generated PBT (assertions):

```
@given(st.lists(st.floats()))
def _test(xs):
    r = fn(xs)
    assert abs(poly(xs, r)) < 1e-4
```

Hypothesis falsifying input: `xs = [0.0, -0.0, 1.0, 0.0]`. Polynomial reduces to x^2; leading coefficient `xs[-1]` $= 0$ *violates precondition (ii)*. Bisection cannot make progress because `poly(`-1`)`$\cdot$`poly(1) = 1`$\cdot$`1 > 0`; the widening loop runs without ever finding a sign change.

Oracle	Reference	Why
Unit tests	**PASS**	Docstring examples respect preconditions
PBT	**FAIL**	PBT samples exclude no inputs (no `assume()`)

Figure 2. Reference-failing PBT, *precondition-ignoring* category. The docstring states two explicit preconditions on the input domain; the LLM-generated PBT does not encode them as `assume(...)` clauses, so Hypothesis samples inputs the specification excludes. The reference cannot satisfy a property checked outside its declared domain. The task is excluded from RQ2–RQ4.