# AI Coding Benchmarks Need Proofs, Not Just Tests

**Daneshvar Amrollahi** [1] **Mahyar Karimi** [2] **Brando Miranda** [1] **Leni Aniva** [1] **Chuyue Sun** [1] **Clark Barrett** [1] **Sanmi Koyejo** [1]

## Abstract

This paper argues that AI coding benchmarks need proof-based evaluation, not just passing tests. As frontier LLMs saturate test-based benchmarks, the signal they provide about code correctness degrades: a finite test suite cannot distinguish correct code from merely plausible code, and no finite curation effort can turn test passing into a general guarantee of semantic correctness. Proof-based evaluation, where each correctness claim is backed by a machine-checkable witness against an explicit formal specification, is the strongest practical oracle we know for moving beyond sampled behavioral checks. The window may be narrow: if benchmark targets shape training, then test-only leaderboards risk reinforcing code generation without corresponding proof-generation capability. Our argument rests on three planks: a structural argument that scaling tests cannot close the correctness gap, empirical evidence from two Lean 4 benchmarks under two frontier models, and a hand-formalized case-study grid drawn from SWE-Bench.

## 1. Introduction

Frontier coding evaluation runs through several test-based benchmarks: SWE-Bench Verified (Jimenez et al., 2024) (multi-file patch generation), LiveCodeBench (Jain et al., 2024) (competitive programming), Terminal-Bench (Merrill et al., 2026) (agentic command-line tasks), and Aider Polyglot (Gauthier, 2024) (multi-language editing). HumanEval (Chen et al., 2021) (the original single-function unit-test benchmark) is now effectively saturated and has largely dropped out of frontier release notes. Every recent frontier release reports against some subset of this list as primary evidence of coding capability. Every benchmark in the list shares the same oracle: a finite set of input-output pairs or task-completion checks that the model's output is run against.

Test-based oracles share a structural limit. A test suite checks a finite number of input-output pairs, and code that passes every test may still fail on inputs the suite never includes. A different oracle is available: *proof-based evaluation*. Each task carries a formal specification, a precise mathematical statement of what the code must do. The model's submission must include a machine-checkable proof that the code satisfies that specification on every input, not just the test cases. A passing proof is a correctness guarantee that holds across the entire input domain. A successful negation proof is a concrete counterexample showing where the code violates its specification. Two recent Lean 4 (Moura & Ullrich, 2021) benchmarks (Verina (Ye et al., 2026) and VeriBench (Miranda et al., 2025)) already operate this way, with formal specifications and machine-checking harnesses in place.

**Our position. To measure progress past the saturation of test-based benchmarks, AI coding benchmarks must require proof-based evaluation, not just passing tests.**

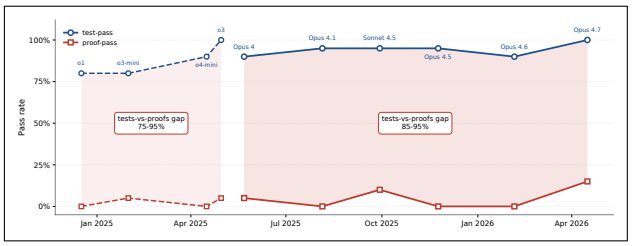


*Figure 1.* Test-pass (blue) and proof-pass (red) across two frontier reasoning lineages on the same 20-task Verina (Ye et al., 2026) subsample, fixed across snapshots with identical prompts. Solid lines mark Claude, dashed lines mark OpenAI o-series. The implementation prompt is scored by Verina's tests. The proof prompt asks the model to prove Verina's specification about the reference implementation, so the two axes measure independent capabilities (code-from-spec and proof-from-reference). One sample per task per prompt, with the previous Lean compiler error fed back at each retry (up to 3 typecheck retries and 10 proof retries). Illustrative of the trend. Primary numbers in §4.

Figure 1 bears this out across snapshots spanning months of frontier progress on both lineages: test-pass climbs to or

[1]Stanford University, Stanford, California, USA [2]TU Wien, Vienna, Austria. Correspondence to: Daneshvar Amrollahi <daneshvar@cs.stanford.edu>.

The $3^{rd}$ *AI for Math Workshop* at the $43^{rd}$ International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

near 100% while proof-pass stays in single digits. The full Verina (n=189) and VeriBench (n=153) numbers, on which the rest of the paper's claims rest, appear in §4.

On the most recent Claude snapshot the test oracle reads 100%. The benchmark can no longer distinguish better code from adequate code, and similar saturation or rapid leaderboard compression has appeared across several prominent test-based coding benchmarks. As tests saturate, the benchmark stops providing signal: a task every model solves gives no ranking, no measurement of progress, no feedback to training. Because evaluation targets shape training pipelines, each generation trained against tests alone narrows the window further. The window to switch oracles is closing now. Proof-based evaluation is one of the few available oracles that can continue measuring semantic correctness after sampled tests saturate, and Fig. 1 shows what the resulting gap looks like.

The position rests on three planks. **Plank 1** (§3) is the structural argument that no amount of scaling tests can close the correctness gap. **Plank 2** (§4) is a per-task measurement of when the gap appears on existing Lean 4 benchmarks, the *prover-limited zone*, where tests pass but no machine-checked verdict can be produced within budget. **Plank 3** (§5) is a set of case studies of merged-and-fixed SWE-Bench bugs, where a formal specification distinguishes buggy code from its upstream fix while the repository's test suite cannot.

## 2. Where the Proof Oracle Stands Today

In this paper, "proof oracle" means a fixed proof checker that accepts only if the submitted implementation is accompanied by a machine-checkable derivation of the task specification.

A parallel proof-based evaluation literature has matured rapidly in 2024 and 2025. Several Lean 4 corpora are purpose-built for formal verification (curated, single-function tasks with pre-written specifications): Verina (Ye et al., 2026), VeriBench (Miranda et al., 2025), FVAPPS (Dougherty & Mehta, 2025), Vericoding (Bursuc et al., 2025), CLEVER (Thakur et al., 2026), and the recent repository-scale VeriSoftBench (Xin et al., 2026). CLEVER additionally demands *specification isomorphism* rather than mere proof existence. They establish that proof-based evaluation works on tasks designed for it. Whether the oracle extends to real-world, multi-file, test-blessed production code (i.e., the kind frontier coding agents are actually benchmarked against) is the open question this paper addresses, inheriting the proof oracle from the formal verification benchmark literature and applying it to SWE-Bench production bugs (§5).

## 3. Why Tests Are Fundamentally Insufficient

We organize our argument in three layers of increasing depth: empirical failures (§3.1), statistical limits (§3.2), and mathematical limits (§3.3). The first two layers admit the response "use better tests." The third does not.

### 3.1. Empirical Failures: Benchmarks Have Cascaded

Most prominent test-based code benchmarks have been followed by a strengthening pass. HumanEval (Chen et al., 2021) was followed by HumanEval+ (EvalPlus), whose expanded test suite broke roughly one-third of previously "passing" solutions (Liu et al., 2023). MBPP (Austin et al., 2021) was followed by a sanitized variant and MBPP+. SWE-Bench (Jimenez et al., 2024) was followed by SWE-Bench Verified, by SWE-Bench+ (Aleithan et al., 2024) (which flagged 55.36% of audited resolutions as suspicious), and by UTBoost (Yu et al., 2025), whose LLM-generated test augmentation surfaced 92 erroneous patches and inflated leaderboard scores by 6 to 7 points. LiveCodeBench (Jain et al., 2024) emerged as a contamination-resistant successor at the function level.

Each pass catches real bugs. None addresses the underlying issue: *incompleteness*. The pattern is a cascade: a benchmark is released, models saturate it, a new class of test-invisible bug is discovered in the "passing" outputs, more tests or stricter curation close that class, until the next model generation finds the next test-invisible bug. The cascade can improve practical adequacy, but it cannot converge to a universal correctness guarantee unless the input domain and property are exhaustively covered: each curation effort is itself a finite test suite, vulnerable to the same failure mode it was built to address.

We emphasize that the claim is *not* that tests are uniformly weak. In our own experiments (§4.2), unit tests catch about 40% of LLM-generated VeriBench outputs at the `#guard` layer, a substantial signal on harder tasks. What tests cannot catch is the subtler bug that compiles, runs, and returns a plausible value on every test input. We later operationalize this gap as the *prover-limited zone* (§4.1): test-passing outputs for which our pipeline obtains neither a correctness proof nor a counterexample proof within budget. The structural and fundamental arguments below make precise why the cascade will not terminate in a finite number of rounds.

### 3.2. Statistical Limits: Sampling and Goodhart

**Goodhart's law.** When a measure becomes a target, it ceases to be a good measure. LLMs are increasingly trained and fine-tuned to maximize test-pass rates on coding benchmarks. Because test-passing code is a strict superset of correct code (it includes every correct implementation *plus* every incorrect implementation that happens to satisfy the

finite test oracle), there is a structural risk that models learn the distribution of *test-passing code* rather than the distribution of *correct code*. This risk is analogous to overfitting in supervised learning: the model minimizes evaluation loss without improving on the true objective.

**Finite sampling.** A test suite checks $k$ input-output pairs drawn iid from a sampling distribution over the input space. Consider a program whose buggy region has measure $\varepsilon$ under that distribution. The probability that the program passes all $k$ tests is:

$$P(\text{all } k \text{ tests pass} \mid \text{bug rate } \varepsilon) \;=\; (1-\varepsilon)^k. \qquad (1)$$

For rare bugs ($\varepsilon \approx 10^{-4}$), even $k = 1{,}000$ tests admit a probability of accepting buggy code exceeding 90%: $(1-10^{-4})^{1000} \approx 0.905$. This iid model is not a model of all benchmark construction. It is a lower-level illustration of why any sampled oracle can miss low-measure bug regions. Curated tests can increase effective $\varepsilon$ for known bug classes, but cannot eliminate unseen regions without exhaustive coverage. A formal proof has zero sampling error with respect to the stated formal specification. The same bound applies to any iid sampling scheme, including randomized fuzzing, with the bound staying close to one for the low $\varepsilon$ regime typical of high-dimensional structured inputs (e.g., database queries, ASTs, code).

**Concrete test-invisible bugs.** The bugs this analysis hides are not exotic. The `(lo + hi) / 2` midpoint computation in binary search overflowed silently in `java.util.Arrays.binarySearch` for nearly a decade: the test inputs were never large enough to trigger it, and every output agreed with the reference on the inputs that were. A race condition in a lock-free data structure evades every sequential test and appears only under concurrent workloads that a unit-test battery does not reproduce. A write-ahead log that appends a record before updating its index pointer in two separate writes leaves the index dangling if the system crashes between them. Sequential unit tests run to completion without simulating power loss, so the inconsistency only surfaces in production filesystems. Each of these bugs is caught by a specification and missed by any finite test suite that does not specifically anticipate it. To defend test-based evaluation is to claim that test designers can anticipate *every* failure mode, a claim refuted empirically in §3.1.

### 3.3. Mathematical Limits: Undecidability and Infinity

The deepest argument against test-based evaluation is *mathematical*. A *test oracle* executes code on inputs and compares outputs, a computational process with two fundamental limitations. First, it checks only finitely many inputs, so it cannot establish universal properties over infinite domains. Second, it is bounded by decidability, so it cannot verify uncomputable properties (*e.g.*, termination). A *proof oracle* changes the evaluation problem from deciding correctness automatically to checking a supplied correctness certificate. We make both claims precise below.

**First limit: finiteness.** No finite test suite verifies a "for every input" property over an input domain larger than the suite. Any input the suite leaves unchecked is fair game for an implementation that returns a forbidden value there and behaves correctly everywhere else.[1]

**A second, independent limit: undecidability.** Some correctness properties cannot be decided by *any* algorithm, not just by a finite test set. Termination on arbitrary inputs is uncomputable (Turing, 1937), and Rice's theorem extends this impossibility to every non-trivial semantic property of programs (Rice, 1953). Tests, being computable, cannot decide what no algorithm decides. This is a stronger limitation than finiteness: no amount of fuzz coverage or property-based generation can plug a gap that algorithmic computation itself cannot plug.

**Why proofs escape both limits.** Proofs do not try to be decision procedures. A proof oracle accepts a program by checking a machine-verified derivation that the program satisfies the property. The derivation is constructed once for the specific program at hand; nothing has to decide the property in general. So even when no algorithm decides the property in the worst case (termination, equivalence), a proof can still establish it for the program at hand. Tests ask "does this program work on the inputs I checked?" and bump into both limits. Proofs ask "is there a witness that this program works on all inputs?" and accept the witness when one is supplied. Proof-based evaluation is not complete: some correct programs may lack a proof within the chosen logic, libraries, or search budget. Its advantage is sound acceptance, not complete decision.

**Example 3.1** (Primality over infinite inputs)**.** A function `is_prime`$(n)$ implemented by trial division has a universal specification: it must agree with the mathematical definition of primality for every $n \in \mathbb{N}$. Any finite test suite reaches only finitely many inputs, leaving room for wrong base cases if omitted, or for loop-bound errors beyond the largest tested input. A Lean 4 proof that the loop realizes divisibility on every $n$ discharges the specification in one stroke, with no concrete witness required.

**Example 3.2** (Termination)**.** By the undecidability of the halting problem (Turing, 1937), no test suite can verify that a program terminates on all inputs. One would have to run

[1] Realized input domains are finite but astronomically large (e.g., $2^{64}$ values for a single `int64` input), so the test suite is always strictly smaller than the domain.

the program on every input and wait for it to halt, which is impossible for infinite input domains and non-terminating programs. Lean 4 supports termination proofs via well-founded recursion: a specific program can be proven terminating by exhibiting a measure that decreases on every recursive call, without deciding termination in general.

**Example 3.3** (Probabilistic programs)**.** Probabilistic correctness properties are statements about distributions (expected values, higher moments, termination behavior), and a finite sample can only *estimate* them, never decide them. Symbolic methods such as moment-based static analysis (Moosbrugger et al., 2022) and martingale-based termination proofs (Moosbrugger et al., 2021) derive exact distributional facts that no finite sample can replace.

**Example 3.4** (Symbolic mathematics libraries)**.** SymPy accounts for a substantial fraction of SWE-Bench tasks (17% of the full suite, 26% of SWE-Bench Lite). Its operations (simplification, integration, equation solving, series expansion, linear algebra) each carry a universal mathematical specification over an infinite expression space, and a bug that misclassifies a specific subclass of inputs hides between test cases. Lean 4 with Mathlib already states the relevant universal propositions across algebra, analysis, and number theory, so a formal proof against the mathematical spec discharges them in one stroke, without enumerating concrete witnesses.

Tests are executions over sampled behaviors. Proofs are certificates checked by a small trusted kernel against symbolic specifications. The distinction is not that proof checking is non-computational, but that a finite certificate can establish a universal property that no finite execution sample can establish.

## 4. An Empirical Instrument for the Test-Proof Gap

Section 3 established that test oracles cannot verify the universal and undecidable properties that proof oracles can. How large is this gap in practice? This section answers empirically. A five-way classification (§4.1) attaches a machine-checked verdict to each LLM output. Instantiating the classifier on Verina and VeriBench under two frontier models, Claude Opus 4.6 and GPT-5.1 with default reasoning effort (§4.2), reveals that a large fraction of test-passing outputs land in the *prover-limited* zone.

This experiment does not estimate the false-positive rate of tests. Instead, it estimates an evidence gap: among test-passing LLM outputs, how often can the same model produce a machine-checkable correctness or incorrectness witness under a fixed budget?

### 4.1. Method

**Inputs.** Each Verina and VeriBench task is a Lean 4 formalization of a single function. The task contains a signature, a precondition (a Lean predicate over the inputs), a postcondition (a Lean predicate over inputs and the function's output), a reference implementation, a reference proof that the reference implementation satisfies the postcondition under the precondition, and a unit-test suite of concrete (input, expected output) pairs compiled into `#guard` assertions. The candidate LLM is given the spec (signature, precondition, postcondition, imports, and auxiliary definitions) and is asked to produce a fresh implementation. The reference implementation and reference proof are hidden throughout.

**Pipeline.** Each LLM-generated implementation is routed through the four-stage pipeline of Figure 2, exiting at the first stage that decides its label. We first attempt to typecheck the code (up to $R_{\text{retry}} = 3$ retries), then run the task's `#guard` assertions, the executable Boolean checks Lean 4 evaluates at compile time, supplied with each Verina and VeriBench task as a unit-test layer. Outputs that survive both checks reach the proof stage: a fresh *spec proof* that the code satisfies its postcondition (up to $N_{\text{proof}} = 50$ LLM proof attempts with Lean error feedback between attempts), and, if that exhausts its budget, a fresh *negation proof* that some precondition-satisfying input falsifies the postcondition, i.e., a counterexample violating the spec (up to $N_{\text{neg}} = 50$ attempts). Every label except *prover-limited* carries a machine-checked derivation: a typecheck, a failing assertion, or a Lean 4 proof in either direction. A successful spec proof yields *confirmed-correct*, and a successful negation proof yields *confirmed-false-positive*, that is, the test oracle's false-positive verdict confirmed by a Lean counterexample. *prover-limited* is therefore an explicit abstention, not a silent default verdict.

**Setup.** The candidate LLMs are Claude Opus 4.6 and GPT-5.1 (snapshot `2025-11-13`, OpenAI's API-default reasoning effort), with the same model producing the implementation, spec proof, and negation proof for each task ($K = 1$ candidate per task). Code, spec-proof, and negation-proof prompts are fixed templates parameterized by the task signature, precondition, postcondition, and any auxiliary definitions. The spec-proof and negation-proof prompts both embed the candidate implementation. Sampling uses each provider's API defaults with `max_tokens=4096` per call. A candidate is test-passing iff the assembled Lean file with the task's `#guard` assertions compiles without error. Spec and negation proof searches are independent feedback-carrying rollouts. Each Lean call has a 120-second compile budget. Full prompts, configs, and per-task outputs will be released upon acceptance.

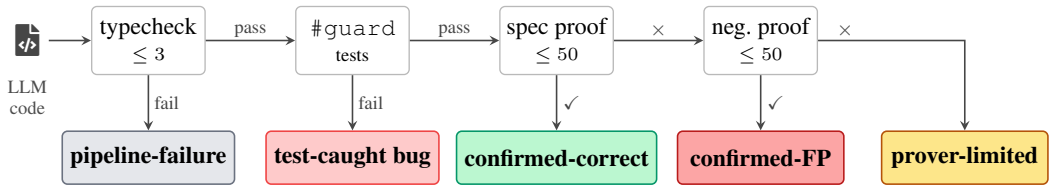


*Figure 2.* Five-way classification pipeline. Each LLM-generated program enters at the left and exits at exactly one of five mutually exclusive labels.

### 4.2. Results

We instantiate the pipeline on Verina (Ye et al., 2026) (189 Lean 4 tasks: 108 basic, 81 advanced) and VeriBench (Miranda et al., 2025) (153 Lean 4 tasks: 55 HumanEval ports, 41 introductory exercises, 32 Python-stdlib functions, 13 classical algorithms, and 12 security tasks).

| | Claude Opus 4.6 | | | | GPT-5.1 | | | |
|---|---|---|---|---|---|---|---|---|
| | Verina (189) | | VB (153) | | Verina (189) | | VB (153) | |
| Bucket | Count | % | Count | % | Count | % | Count | % |
| confirmed-correct | 29 | 15.3 | 1 | 0.7 | 35 | 18.5 | 1 | 0.7 |
| confirmed-false-positive | 0 | 0.0 | 0 | 0.0 | 0 | 0.0 | 0 | 0.0 |
| prover-limited | 113 | 59.8 | 59 | 38.6 | 101 | 53.4 | 43 | 28.1 |
| test-caught bug | 10 | 5.3 | 61 | 39.9 | 8 | 4.2 | 52 | 34.0 |
| pipeline-failure | 37 | 19.6 | 32 | 20.9 | 45 | 23.8 | 57 | 37.3 |

*Table 1.* Five-way classification under $N_{\text{proof}} = N_{\text{neg}} = 50$. VB = VeriBench. Counts in parentheses are task totals.

**Prover-limited share.** The prover-limited bucket captures a large fraction of LLM outputs across both benchmarks under both models: 60% of Verina and 39% of VeriBench outputs under Claude Opus 4.6, 53% of Verina under GPT-5.1, and 28% of VeriBench under GPT-5.1. These outputs pass every `#guard` test, yet neither a spec proof nor a negation proof closes within budget.

**Verina vs VeriBench.** Verina yields a non-trivial confirmed-correct fraction (15% under Claude Opus 4.6, 19% under GPT-5.1), whereas VeriBench yields essentially none (0.7% under Claude Opus 4.6, 0.7% under GPT-5.1). Two non-exclusive forces drive this gap. First, VeriBench's tests reject 40% of outputs before any proof is attempted (against 5% on Verina), and this lower test pass rate suggests harder tasks, meaning harder specs to prove. Second, the shrunken test-passing pool leaves fewer candidates that even reach the proof stage.

**Zero confirmed-false-positive.** The 0% confirmed-false-positive row admits two possible scenarios. First, the LLM outputs reaching the proof stage are in fact correct, in which case no counterexample exists by definition. Second, the outputs are not correct, but the LLM failed to provide a negation proof. The row alone does not distinguish these scenarios, though task simplicity and model strength plausibly favor the first. Even if the second scenario were the right read, producing a counterexample would still be harder than proving correctness: the LLM must first guess a specific input, then prove the precondition holds and the postcondition fails on it, with no Lean tactic searching the counterexample space. The prover-limited zone is therefore where the two oracles could disagree, and where the argument for proof-based evaluation in this paper applies.

## 5. SWE-Bench Case Studies

Six merged-and-fixed SWE-Bench bugs, formalized in Lean 4, show that the oracle gap from §3 is not a theoretical artifact: it appears in the same repositories practitioners use to evaluate AI coding agents. We select two bugs each from three production-code archetypes (algebraic/combinatorial errors, shape/size mismatches, control-flow slips) across three repositories. In every case, the repository's test suite accepts the pre-fix code; a Lean 4 specification grounded in the GitHub issue does not.

| Archetype | Repo | Instance |
|---|---|---|
| Algebraic / combinatorial | sympy | #11438 (`classify_diop`) |
| | sympy | #18810 (`generate_derangements`) |
| Shape / size | scikit-learn | #10986 (`logistic_regression_path`) |
| | scikit-learn | #10687 (`Lasso.coef_`) |
| Control flow | astropy | #14309 (`is_fits`) |
| | astropy | #14995 (`_arithmetic_mask`) |

These six were chosen for clarity of exposition from roughly twenty attempted formalizations. Some attempts captured the bug locally but relied on abstractions or assumptions that did not generalize to the surrounding repository, and we excluded those for readability rather than soundness.

**Worked example: derangement generator (sympy #18810).** A *derangement* of a list `perm` is a permutation with no fixed point: no index where the rearranged element equals the original. SymPy's `generate_derangements` should yield exactly the derangements of its input. The pre-fix code did this by filtering candidates against the wrong baseline: instead of comparing each candidate to `perm`, it compared to `sorted(perm)`. When the input is already sorted the two agree and the bug is silent. When the input is unsorted, the filter accepts non-

derangements. The gold fix replaces the sorted baseline with `perm` itself. The Lean 4 specification encodes the property as a one-line universal quantifier:

```
def spec (filter : List Nat -> List Nat ->
     Bool) : Prop :=
  forall perm p : List Nat,
    filter perm p = true ->
      forall pr in perm.zip p, pr.1 != pr
    .2
```

For any inputs the filter accepts, every position must disagree between `perm` and the candidate. The spec encodes *one direction only*: if the filter accepts, the input is a derangement. It does not also require the filter to accept every derangement, so a filter that rejects every input would still satisfy it. This is deliberate. Each spec targets the one invariant the bug violates, and additional checks are a straightforward extension when a bug requires it.

**Specification structure.** Our SWE-Bench formalizations are not full functional correctness proofs of the repositories. They are *bug-separating specifications*: formal regression properties derived from issue descriptions that the fixed code satisfies and the pre-fix code violates. Each specification is a universally quantified predicate over the full input space. We run the same specification against two implementations: the pre-fix code and the gold patch. The gold implementation satisfies the spec; the pre-fix implementation is refuted by a negation proof exhibiting a minimal concrete counterexample (Appendix A). The specification does not change between the two runs. The same condition separates correct from incorrect code. Inputs are quantified universally, not fixed to counterexample dimensions: the worked example's spec ranges over arbitrary `perm p : List Nat`, and the other five formalizations follow the same pattern, so a future bug in the same module reuses the same type verbatim.

**Cost and coverage.** The six formalizations have a median of 28 source lines of Lean each, making the per-bug cost of writing a soundness check roughly proportional to understanding the bug, not to building infrastructure from scratch. Scaling to the broader benchmark is aided by SWE-Bench's composition: roughly half of its 2,294 tasks (1,083 instances (Jimenez et al., 2024)) involve mathematical Python libraries, the domain where Lean's Mathlib infrastructure is most mature. The same archetypes (shape mismatches, control-flow slips, missing-case errors) recur commonly in the non-Mathlib half. The main open problem is infrastructure for multi-file, stateful, and concurrent code, which is what CSLib (Barrett et al., 2026) is building (§7). These case studies show that proof-based evaluation is not confined to synthetic Lean tasks and can be applied to representative SWE-Bench-style bugs. Estimating coverage over the full benchmark remains future work.

## 6. Alternative Views

This section engages the strongest objections to our position.

**Prover-limited outputs are mislabeled correct programs.** A reviewer might argue that the prover-limited bucket is nothing more than correct programs the pipeline failed to verify, so the bucket count is an artifact of the pipeline rather than evidence about the state of LLM coding. On Verina and VeriBench's small, single-function tasks, the factual half of this objection partially lands. The programs are simple, the candidate models are frontier, and the audit in §4 confirmed correctness on every case we inspected. We do not contest that the bulk of these outputs are likely correct programs.

That concession does not threaten the position of this paper. By definition, the prover-limited bucket is the set of LLM outputs for which the model produced no machine-checked verdict in either direction, neither a correctness proof nor a negation proof. Whether or not such an output is in fact correct, the model supplied no verified reasoning about its behavior, and a downstream user has no machine-checked basis to trust or reject it. That state of affairs, code emitted without any accompanying verified reasoning about its correctness, is precisely the failure mode this paper argues future work should eliminate. The bucket is not a defect of the pipeline. It is the measurement the pipeline is designed to surface.

**A stronger prover would close the gap.** We agree the prover-limited fractions reported in §4 depend on the benchmark, the model, and the proof-search budget. A different benchmark choice or a stronger prover would shift them. But the benchmark-choice half of the objection runs backward. Verina and VeriBench are the easy end of software verification: small, single-function, pure. Harder benchmarks (multi-file repositories, stateful code, concurrency, SWE-Bench-scale targets) would enlarge the bucket at today's proof capability, not shrink it. What Verina and VeriBench establish is the floor of the gap on the easiest tasks, not its ceiling.

Yes, a stronger prover would shrink the bucket on these tasks. That is precisely the research program this paper calls for in §7: invest in proof search and autoformalization-in-the-loop to drive the bucket down. The numbers in §4 measure where automated proof generation is today, not where it could be in principle. So the objection actually concedes the paper's thesis: the gap exists, and closing it is a research direction worth investing in.

**Proof-based evaluation has its own Goodhart problem.** If tests can be gamed, so can proofs: models could learn

to produce Lean that typechecks rather than code that is actually correct. Autoformalization-in-the-loop makes this worse, since the LLM writes the spec *and* writes the proof, so the same model sits on both sides of the closed loop. The concern is real, but the failure modes differ. The Lean kernel enforces a fixed calculus, not a learned one, so gaming requires either a kernel soundness bug (rare in decades of use) or a specification that does not capture intent (the faithfulness of autoformalization problem (Amrollahi et al., 2026)). Only the second is a live attack surface. The canonical form is *vacuous truth*: a spec of the form $P \Rightarrow \top$, or quantified over an empty hypothesis, which any implementation trivially satisfies.

This is a spec-review problem, structurally the same as test-review but more tractable. Beyond vacuous truth, a reviewer must also check for underconstrained postconditions, wrong abstractions of runtime behavior, and mismatches between source-language semantics and the Lean model. A specification is a single written statement of what the code should do. A test suite's intended property is only whatever the test designer happened to encode in examples. A reviewer can disagree that a spec captures the right intent, but cannot disagree about what the spec literally says. Lean's type theory pins that down. Not every ML researcher needs to read Lean, but benchmark credibility requires auditability by a mixed community of ML and formal-methods experts. We scope autoformalization accordingly: §7 places spec construction under human expert review at benchmark-build time, a one-time cost per task amortized across every model later evaluated against the benchmark.

**Strengthen the tests instead of shifting oracles.** The cheapest fix to the audits in §3.1 is more and better tests: HumanEval+-style augmentation, property-based testing, differential testing, LLM-generated edge cases. On this view, we have shown the oracle needs strengthening, not replacement. The first two layers of §3 accept this: stronger tests close many practical gaps and should be pursued. The third layer does not. Section 3.3's finiteness and undecidability limits hold for every finite test suite, so the fundamental layer forces a category shift, not another quantitative improvement.

**Writing specs is too costly.** Writing specifications is harder than writing tests, and the upfront cost is real: someone has to author and review each spec before it can evaluate anyone. Much of that cost has already been paid. Verina, VeriBench, FVAPPS, and Vericoding (§2) exist and are in use. The question is whether to scale what has started, not whether to start. A spec is written once and reused by every model evaluated against the benchmark, a substantial cost on day one but small by the tenth independent evaluation.

The relevant cost metric is the per-spec trajectory. Agentic autoformalization pipelines (Wu et al., 2022; Gulati et al., 2024; Gallardo et al., 2025) are actively reducing per-spec authoring cost, putting a 100-spec SWE-Bench leaderboard, enough for a reliable proof-pass rate, within today's hand-authoring budget, and 1,000 within reach as automation catches up.

**ML researchers don't read Lean.** Fluency is required of authors, not users. Model builders consume the leaderboard, not the Lean source, so the skill barrier falls on benchmark authors alone. Autoformalization-in-the-loop (§7) is the lever we propose for scaling authoring down further. It is enabling technology that is improving, not a silver bullet we already hold.

**It is too early to switch oracles.** The status quo is not free. Weak oracles cost the community in wasted research effort and misallocated training signal, compounded across every model generation trained against tests alone. Scaling to multi-file SWE-Bench patches remains open (§7). The imperative here is to start now, while proof-generation capability can still be jointly trained alongside code capability, not to finish tomorrow.

## 7. Conclusion

Many prominent test-based coding benchmarks are saturating or rapidly losing discriminative power. The prover-limited zone captures a large fraction of test-passing outputs on Verina and VeriBench, and extends to production code via the SWE-Bench formalizations of §5.

**What we are not claiming.** We are not claiming that all software benchmark tasks can or should be fully verified today. We are not claiming that proofs eliminate specification error. We are not claiming that tests should be abandoned. We are claiming that frontier coding evaluation should include proof-carrying tasks because test-only pass rates no longer provide sufficient evidence of semantic correctness.

**What to build.** Four concrete directions follow.

- **Standardized infrastructure (most urgent).** Lean 4 specification corpora, reproducible proof-existence pipelines, and leaderboards that report prover-limited rates alongside accuracy.
- **Verification infrastructure for real code.** CSLib (Barrett et al., 2026) is formalizing the CS foundations (models of computation, concurrency, programming-language semantics, complexity proofs), with work already started on Lean verification of real programming languages like Rust, one piece of the broader puzzle.

- **Proof verification as training signal.** Reinforcement learning from Lean type-checking success on spec or negation proofs, so that proof-generation capability keeps pace with code generation.
- **Autoformalization at scale.** Agentic autoformalization pipelines (Gallardo et al., 2025) should be pushed further, so specification production scales beyond hand curation.

The window to build it is narrowing. Without a proof-based evaluation target, proof-generation capability has no obvious training signal to track code-generation capability, and the prover-limited zone may continue to dominate test-passing outputs. Investing now in LLMs that can produce machine-checked proofs, while these capabilities can still be jointly trained by cheap signals, is one of the few paths to measuring frontier coding progress past test saturation.

## References


Aleithan, R., Xue, H., Mohajer, M. M., Nnorom, E., Uddin, G., and Wang, S. Swe-bench+: Enhanced coding benchmark for llms, 2024. URL https://arxiv.org/abs/2410.06992.

Amrollahi, D., Lopez, J., and Barrett, C. Faithful Autoformalization via Roundtrip Verification and Repair, 2026. URL https://arxiv.org/abs/2604.25031.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Barrett, C., Chaudhuri, S., Montesi, F., Grundy, J., Kohli, P., de Moura, L., Rademaker, A., and Yingchareonthawornchai, S. Cslib: The lean computer science library, 2026. URL https://arxiv.org/abs/2602.04846.

Bursuc, S., Ehrenborg, T., Lin, S., Astefanoaei, L., Chiosa, I. E., Kukovec, J., Singh, A., Butterley, O., Bizid, A., Dougherty, Q., Zhao, M., Tan, M., and Tegmark, M. A benchmark for vericoding: formally verified program synthesis, 2025. URL https://arxiv.org/abs/2509.22908.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Dougherty, Q. and Mehta, R. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2502.05714.

Gallardo, P., Raissi, M., Zhang, K., and Murthy, S. Agentic lean auformalization (ALA): An LLM collaborative approach to autoformalization in LEAN. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling*, 2025. URL https://openreview.net/forum?id=DNunBkhEUx.

Gauthier, P. Aider Polyglot coding benchmark. https://aider.chat/docs/leaderboards/, 2024. Accessed April 2026.

Gulati, A., Ladsaria, D., Mishra, S., Sidhu, J., and Miranda, B. An evaluation benchmark for autoformalization in lean4, 2024. URL https://arxiv.org/abs/2406.06555.

Jain, N., Han, K., 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, 2024. URL https://arxiv.org/abs/2403.07974.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

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, 2023. URL https://arxiv.org/abs/2305.01210.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., Shen, J., Ye, G., Lin, H., Poulos, J., Wang, M., Nezhurina, M., Jitsev, J., Lu, D., Mastromichalakis, O. M., Xu, Z., Chen, Z., Liu, Y., Zhang, R., Chen, L. L., Kashyap, A., Uslu, J.-L., Li, J., Wu, J., Yan, M., Bian, S., Sharma, V., Sun, K., Dillmann, S., Anand, A., Lanpouthakoun, A., Koopah, B., Hu, C., Guha, E., Dreiman, G. H. S., Zhu, J., Krauth, K., Zhong, L., Muennighoff, N., Amanfu, R., Tan, S., Pimpalgaonkar, S., Aggarwal, T., Lin, X., Lan, X., Zhao, X., Liang, Y., Wang, Y., Wang,

Z., Zhou, C., Heineman, D., Liu, H., Trivedi, H., Yang, J., Lin, J., Shetty, M., Yang, M., Omi, N., Raoof, N., Li, S., Zhuo, T. Y., Lin, W., Dai, Y., Wang, Y., Chai, W., Zhou, S., Wahdany, D., She, Z., Hu, J., Dong, Z., Zhu, Y., Cui, S., Saiyed, A., Kolbeinsson, A., Hu, J., Rytting, C. M., Marten, R., Wang, Y., Dimakis, A., Konwinski, A., and Schmidt, L. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Miranda, B., Zhou, Z., Nie, A., Obbad, E., Aniva, L., Fronsdal, K., Kirk, W., Soylu, D., Yu, A., Li, Y., and Koyejo, S. Veribench: End-to-end formal verification benchmark for AI code generation in lean 4. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL https://openreview.net/forum?id=rWkGFmnSNl.

Moosbrugger, M., Bartocci, E., Katoen, J.-P., and Kovács, L. Automated termination analysis of polynomial probabilistic programs. In Yoshida, N. (ed.), *Programming Languages and Systems*, pp. 491–518, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72019-3.

Moosbrugger, M., Stankovič, M., Bartocci, E., and Kovács, L. This is the moment for probabilistic loops. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi: 10.1145/3563341. URL https://doi.org/10.1145/3563341.

Moura, L. d. and Ullrich, S. The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. URL https://api.semanticscholar.org/CorpusID:120980829.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., and Chaudhuri, S. CLEVER: A curated benchmark for formally verified code generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=IbOacMF5qd.

Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: https://doi.org/10.1112/plms/s2-42.1.230. URL https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

Wu, Y., Jiang, A. Q., Li, W., Rabe, M. N., Staats, C., Jamnik, M., and Szegedy, C. Autoformalization with large language models, 2022. URL https://arxiv.org/abs/2205.12615.

Xin, Y., Chen, Q., Durrett, G., and Dillig, I. Verisoftbench: Repository-scale formal verification benchmarks for lean, 2026. URL https://arxiv.org/abs/2602.18307.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation. In *International Conference on Learning Representations (ICLR)*, 2026.

Yu, B., Zhu, Y., He, P., and Kang, D. Utboost: Rigorous evaluation of coding agents on swe-bench, 2025. URL https://arxiv.org/abs/2506.09289.

# A. SWE-Bench Case Studies: Specs and Counterexamples

For each of the six bugs in §5, we describe what the pre-fix code was supposed to do, what it actually did, and the Lean 4 specification that separates the two.

## A.1. Algebraic / combinatorial (sympy)

**#11438: Diophantine classifier.** SymPy's `classify_diop` inspects a Diophantine equation and tags it by algebraic form so downstream solvers can dispatch to the right routine. One such tag, `general_sum_of_even_powers`, is meant for equations of the form $\sum x_i^n = c$ for a single even exponent $n > 3$. Pre-fix, the check only verified that the total degree was even and exceeded 3. It never required every monomial's exponent to match. An equation like $x^4 + y^2 = c$ was wrongly tagged as a sum of fourth powers and routed to a solver that could not handle mixed exponents. The spec makes the missing requirement explicit: if the classifier accepts, every monomial's exponent must equal the claimed top degree.

```
def spec (classify : DiopEq -> Bool) : Prop :=
  forall eq, classify eq = true ->
    forall d in eq.other_degrees, d = eq.first_degree
```

*Counterexample:* the equation $x^4 + y^2 = c$. The pre-fix classifier accepts (degree 4 is even and exceeds 3). The spec then demands every monomial's exponent equal 4, i.e., $2 = 4$, which fails.

**#18810: derangement generator.** A *derangement* of a sequence $\sigma$ is a permutation $p$ where no index matches: $p_i \neq \sigma_i$ for every $i$. SymPy's `generate_derangements`$(\sigma)$ should yield exactly the derangements of $\sigma$. The pre-fix code filtered each candidate against the wrong baseline: instead of comparing to $\sigma$, it compared to $\mathrm{sort}(\sigma)$. When $\sigma$ is already sorted, the two agree and the bug is silent. When $\sigma$ is unsorted, non-derangements slip through.

```
def spec (filter : List Nat -> List Nat -> Bool) : Prop :=
  forall perm p, filter perm p = true ->
    forall pr in perm.zip p, pr.1 != pr.2
```

*Counterexample:* $\sigma = [1, 0]$, $p = [1, 0]$. The two sequences are identical, so every index is a fixed point and $p$ is not a derangement of $\sigma$. But the pre-fix filter compares $p$ to $\mathrm{sort}(\sigma) = [0, 1]$, which disagrees with $[1, 0]$ at both indices, so it accepts.

## A.2. Shape / size (scikit-learn)

**#10986: warm-start broadcast.** scikit-learn's `logistic_regression_path` supports *warm-starting*: a previous coefficient estimate is reused as the initial point for the next regularization level. In the binary multinomial case, scikit-learn internally represents the two softmax classes with `n_classes = 1` (one class is redundant), but the weight matrix `w0` still has two rows, and those two rows must be antisymmetric, `[-c, c]`, for the softmax parametrization to hold. The pre-fix code initialized `w0` by copying the warm-start `coef` into the first row and relying on numpy's broadcast to fill the rest. Broadcast silently duplicates `c` into both rows (yielding `[c, c]`), violating the required antisymmetry without any shape error. Bugs of this form recur throughout numerical Python, where a routine assignment can quietly produce the wrong value without raising any error.

```
def spec (f : Nat -> List (List Int) -> W0) : Prop :=
  forall c, (f 1 [c]).rows = [c.map (fun x => -x), c]
```

*Counterexample:* `n_classes = 1`, `coef = [[1]]`. The spec demands rows `[[-1], [1]]`. The pre-fix code returns `[[1]]` in our single-row abstraction (in the runtime, `[[1], [1]]` via broadcast).

**#10687: Lasso coefficient shape.** After fitting, scikit-learn's `Lasso` exposes the learned weights as `self.coef_`. The documented contract: for single-target regression, `coef_.shape == (n_features,)`, that is, a 1-D array with one entry per input feature. Downstream code (`_set_intercept`) reads `coef_` as a rank-1 array and breaks if it is not. The pre-fix code produced `coef_` by running the solver's 2-D output through `np.squeeze`, a generic routine that removes *every* axis of length 1. On a typical multi-feature fit this works. On a single-feature fit the solver returns shape `(1, 1)`, squeeze collapses both axes, and `coef_` becomes a 0-D scalar. The feature axis is gone, and downstream code crashes.

```
def spec (f : CoefMatrix -> CoefResult) : Prop :=
  forall feats, (f { rows := [feats] }).isVector = true
```

*Counterexample:* `feats = [1]` (one target, one feature). The solver's $1 \times 1$ output is fed through `np.squeeze`, both singleton axes collapse, and the pre-fix code returns `scalar 1` (rank 0), failing the spec's `isVector = true`.

### A.3. Control flow (astropy)

**#14309: FITS detection fall-through.** astropy's `is_fits` decides whether an input refers to a FITS file (a standard astronomical image format). It dispatches on three regimes: a raw file object (check the signature bytes), a filepath (check the extension), or anything else (inspect `args[0]` for an HDU-like object, where HDU is FITS's Header Data Unit, the basic structural element of a FITS file). Only one regime should fire per call. The pre-fix code implemented the filepath regime as `if filepath.lower().endswith(...): return True`, with no explicit `return False` on the mismatch case. When the filepath did not have a FITS extension, control fell through to the third regime and checked `args[0]`, so a filepath with the wrong extension but an HDU-like argument tacked on was misclassified as a FITS file.

```
def spec (is_fits : IsFits) : Prop :=
  forall e args, is_fits none (some e) args = e
```

*Counterexample:* the filepath regime fires (no `fileobj`, `filepath` present), the extension does not match (`e = false`), and `args[0]` happens to be HDU-like (`true`). The spec demands the result equal `e`, i.e., `false`. The pre-fix code falls through to `args[0]` and returns `true`, so `args` leaks into a decision that should depend only on the filepath.

**#14995: NDData mask arithmetic crash.** astropy's `NDArithmeticMixin._arithmetic_mask` computes the output mask when two `NDData` operands are combined arithmetically. Four cases: both have masks, only `self` has, only `operand` has, or neither has. All four should yield a well-defined output mask (the gold fix returns `self.mask` in the third case). The pre-fix code's third branch read `elif operand is None:`, but the intended check was `operand.mask is None`. In the arithmetic code path, `operand` is always a fully-constructed `NDData` object, so `operand is None` never fires. The third case is effectively dead, and control falls through to `handle_mask(self.mask, None)`, which raises, because one cannot `bitwise_or` with a `None`.

```
def spec (f : NDData -> NDData -> MaskResult) : Prop :=
  forall self operand, (f self operand).isOk = true
```

*Counterexample:* `self.mask = some 1`, `operand.mask = none`. The third case (operand present, its mask absent) should fire, but the buggy guard never matches. Control reaches `handle_mask(self.mask, None)` and raises, violating the spec's demand for a well-defined output on every mask combination.