VeriBench: An End-to-End Formal Verification Benchmark for AI Coding Agents in Lean 4

Anonymous Authors[1]

Abstract

Test-based coding benchmarks have repeatedly required strengthening passes—HumanEval gave way to HumanEval+, SWE-Bench to SWE-Bench Verified to SWE-Bench+, each release exposing test-invisible bugs in code that passed the prior suite—a structural limitation of finite testing: passing tests routinely leave behaviorally important bugs unobserved. Existing formal-verification benchmarks (e.g., VERINA, FVAPPS, CLEVER, DafnyBench) provide stronger machine-checkable signals, but many focus on proof completion, scaffolded verified generation, or isolated formal subtasks rather than the full path from developer-written source code to a verified formal artifact. We argue that trustworthy code-verification benchmarks must be *end-to-end* and *agentic*, scoring full Python-to-Lean autoformalization under verifier feedback, and must aggregate verification stages *conjunctively*—weakness at any stage penalizing the composite while preserving smooth evaluation signals that retain discriminative power across models still far from the frontier. We introduce VERIBENCH, a 452-task end-to-end Python-to-Lean 4 autoformalization benchmark spanning HumanEval-style programs, classical algorithms, Python standard-library functions, security examples, and 282 high-assurance-inspired tasks across 14 domains such as cryptography, aerospace, medical devices, and compilers. We score agents with the Smooth Conjunctive Score for Code verification (SCSC), a log-domain geometric mean over five per-task factors: $\mathrm{SCSC} = \exp\left(\frac{1}{5}\sum_i \log f_i\right)$, combining (i) the agent's Lean file typechecks, (ii) the agent's theorems verify without `sorry`, (iii) the agent's theorems semantically cover the gold theorems, and (iv,v) gold-side benchmark-validity gates (D_1, D_2) ensuring the gold reference itself compiles and proves cleanly before being used to score agents. Under an agentic verifier-feedback loop, Codex, Claude Code, and Leanstral-v2 reach SCSC of only 0.42, 0.36, and 0.23; iterative self-correction adds 14.3% over single-shot baselines, yet theorem–gold coverage stalls uniformly at ≤ 0.11 across all three agents—a *specification gap* validated by an LLM judge calibrated against five independent human raters (Pearson $r = 0.70$, $p < 10^{-11}$). VERIBENCH reframes code-verification evaluation from isolated proof search to end-to-end conjunctive grounding, surfacing specification synthesis as a bottleneck at least as severe as proof search and offering a measurable target for the next generation of verifiable AI coding agents.

1. Introduction

Test-based coding benchmarks have repeatedly required strengthening passes: HumanEval (Chen et al., 2021) gave way to HumanEval+ (Liu et al., 2023); SWE-Bench (Jimenez et al., 2024) to SWE-Bench Verified (OpenAI, 2024) to SWE-Bench+ (Aleithan et al., 2024). Each release exposed test-invisible bugs in code that already passed the prior suite, a pattern that reflects a structural limit of finite testing rather than an artifact of any single benchmark (Anonymous Authors, 2026): passing tests routinely leave behaviorally important bugs unobserved, and AI-assisted code has been shown to introduce security vulnerabilities at higher rates than unaided developers (Perry et al., 2023; Pearce et al., 2022). Trustworthy evaluation of AI coding agents—especially in security-critical and high-assurance domains—needs a signal stronger than test passing.

Formal verification offers such a signal: a machine-checkable certificate that a program meets a stated specification (Hoare, 1969). Existing formal-verification benchmarks (Ye et al., 2025; Dougherty & Mehta, 2025; Thakur et al., 2025; Loughridge et al., 2024; Sun et al., 2024) take

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

this direction, but each evaluates a slice of the verification pipeline rather than the full path from developer-written source code to a checked formal artifact. VERINA (Ye et al., 2025) and CLEVER (Thakur et al., 2025) feed agents natural-language descriptions or partial scaffolds and score component-level subtasks, such as specification generation, in isolation. FVAPPS (Dougherty & Mehta, 2025) and DAFNYBENCH (Loughridge et al., 2024) score proof completion against pre-written specifications. Across this prior work, the implicit assumption is that verification quality can be evaluated by scoring isolated subtasks one at a time and combining the results compensatorily, where high scores on some dimensions can mask failures on others (Zhang et al., 2025). This treatment misses the conjunctive nature of real verification, in which any broken stage invalidates the entire certificate.

We instead argue that a trustworthy code-verification benchmark must be *end-to-end*—scoring the complete path from a developer-written program to a checked formal artifact—and *agentic*, allowing iterative refinement under verifier feedback as production verification workflows actually proceed. Crucially, the per-stage scores must be aggregated *conjunctively* rather than averaged: weakness at any stage—the program does not typecheck, the theorems do not prove, or the agent's specifications do not cover the gold reference—should penalize the composite, mirroring the all-or-nothing semantics of formal correctness. At the same time, the aggregator must remain smooth so it retains discriminative power across models still far from the frontier, where rankings of partial competence carry the most information. This conjunctive end-to-end framing reorients evaluation from "which subtask does each model solve?" to "what fraction of the full verification chain does the model hold together?".

We instantiate this framing as VERIBENCH, a 452-task end-to-end Python-to-Lean 4 autoformalization benchmark spanning HumanEval-style programs (Chen et al., 2021), classical algorithms, Python standard-library functions, security examples adapted from MIT 6.858 (CSAIL), and 282 high-assurance-inspired tasks across 14 industry-relevant domains including cryptography, aerospace, medical devices (Jiang et al., 2012), and processor verification (Reid, 2016). To score agents on this benchmark, we propose the *Smooth Conjunctive Score for Code verification* (SCSC), a log-domain geometric mean over five per-task factors, $\mathrm{SCSC} = \exp\big(\frac{1}{5}\sum_i \log f_i\big)$, combining (i) the agent's Lean file typechecks, (ii) the agent's theorems verify without `sorry`, (iii) the agent's theorems semantically cover the gold theorems, and (iv,v) gold-side benchmark-validity gates (D_1, D_2) ensuring the gold reference itself compiles and proves cleanly before being used to score agents (Figure 3). The geometric mean realizes conjunctive aggregation in a smooth, partial-credit form: a near-zero in any factor pulls the composite toward zero, while small differences across factors remain discriminative.

We evaluate frontier coding agents under an agentic verifier-feedback loop in which each agent receives only the Python source and must autonomously generate a complete Lean 4 artifact—implementation, tests, theorems, and proofs. Codex (GPT-5.4), Claude Code (Sonnet 4.6), and Leanstral v2 reach SCSC of only 0.42, 0.36, and 0.23, respectively, with iterative self-correction adding 14.3% over single-shot baselines. Strikingly, theorem–gold coverage stalls uniformly at ≤ 0.11 across all three agents: agents routinely produce Lean files that typecheck, yet the theorems they state systematically fail to cover the specifications a human curator wrote for the same task. We name this regularity the *specification gap* and validate it with an LLM judge calibrated against five independent human raters (Pearson $r = 0.70$, $p < 10^{-11}$; Appendix F). The gap appears across program families and persists under self-correction, indicating that the dominant bottleneck is specification synthesis, not proof search.

VERIBENCH reframes code-verification evaluation from isolated proof search to end-to-end conjunctive grounding, surfacing specification synthesis as a bottleneck at least as severe as proof search and offering a measurable target for the next generation of verifiable AI coding agents.

Contributions.

- **An end-to-end agentic benchmark.** VERIBENCH: 452 tasks running from developer-written Python source to checked Lean 4 artifacts, spanning HumanEval-style programs, classical algorithms, Python standard-library code, security-critical examples, and 282 high-assurance-inspired tasks across 14 industrial domains (Section 3, Appendix C).
- **A smooth conjunctive score.** The Smooth Conjunctive Score for Code verification (SCSC), a five-factor log-domain geometric mean that enforces all-or-nothing verification semantics while preserving partial-credit signal for sub-frontier models (Section 4).
- **An empirical specification gap.** Frontier agents reach $\widetilde{S} \leq 0.42$ with theorem–gold coverage uniformly ≤ 0.11, isolating specification synthesis—not proof search—as the dominant failure mode (Section 5).
- **A human-calibrated coverage judge.** An LLM coverage judge calibrated against five independent human raters (Pearson $r = 0.70$, $p < 10^{-11}$), enabling scalable evaluation that tracks expert assessment of theorem semantic coverage (Appendix F).

Benchmark	Lang.	Scope	Executable	Source
MiniF2F	Lean	Proof-only	No	Math Competitions
Clover	Dafny	Code & Spec (Template)	Yes	Hand-crafted (Textbook Style)
VERINA	Lean 4	Modular (Template-based)	Yes	Competitive Programming
VeriBench	**Lean 4**	**End-to-End**	**Yes**	**Real + Security + Competitive Programming + CS**

Table 1. Comparison of Formal Verification Benchmarks. Unlike VERINA, which fills templates for individual components, VERIBENCH targets an underexplored setting: end-to-end translation of executable Python tasks (HumanEval-style programs, classical algorithms, Python standard-library functions, and pedagogical security examples) into structured Lean 4 artifacts containing implementations, tests, theorem statements, and proofs.

2. Related Work

Benchmarks for Code Verification. Loughridge et al. (2024) introduce DAFNYBENCH, the first large-scale benchmark for evaluating LLMs in formal software verification. It consists of over 750 Dafny programs (approximately 53K lines of code) stripped of verification "hints," requiring models to regenerate the missing annotations to pass the verifier. Evaluated on GPT-4, GPT-4 Turbo, Claude 3, and others, the best-performing system achieved a success rate of roughly 68%, demonstrating the potential of machine-assisted verification while highlighting performance variability with respect to program size, hint complexity, and retry strategies.

Targeting human-assisted development, Ye et al. (2025) benchmark the interactive synthesis of Lean 4 artifacts, feeding models natural language descriptions supplemented by optional code or specification scaffolds. We classify this methodology as a *scaffolded completion task*, fundamentally distinct from our focus on autonomous **end-to-end autoformalization**. By requiring agents to autoformalize existing software (specifically Python as a motivating example, given its ubiquity (GitHub, 2024)) into formal Lean models, we rigorously measure the ability to autonomously *architect* modularity and abstraction—critical design principles bypassed by scaffolded generation. This aligns with the practical necessity of verifying existing codebases rather than enforcing native Lean development. Furthermore, we eschew test-based specification evaluation, which fails on non-computable theorems, in favor of a prover-based coverage check that formally verifies whether the agent's specifications entail our ground reference.

In parallel, Dougherty & Mehta (2025) introduce FVAPPS, a machine-generated Lean 4 benchmark of 4,715 problems stratified by assurance level. Built via a test-driven LLM pipeline, it shows that top models like Claude Sonnet and Gemini 1.5 Pro prove only 30% and 18.5% of theorems, respectively, with human-written solutions still falling short at scale. CLEVER (Thakur et al., 2025) offers a more focused challenge: 161 Lean problems requiring both a formal spec and a correctness-proofed implementation. It's non-computable, spec-agnostic setup avoids test leakage and demands true reasoning—so far, models fully solve only 1 of 161 tasks. Finally, broadening the scope beyond formal proof, Ouyang et al. (2025) introduces KERNELBENCH, a benchmark for generating optimized GPU kernels for 250 PyTorch workloads. Using profiler feedback and examples, iterative loops boost success from 12% to over 70%, showcasing the impact of reinforcement-style correction.

VeriBench distinguishes itself from existing benchmarks by targeting the full pipeline of formal code verification grounded in realistic programming tasks. Unlike FVAPPS, which consists of thousands of machine-generated Lean problems optimized for scale and raw proof success rates, VeriBench uses human-curated Python functions drawn from foundational algorithms and practical programming contexts. Each example is paired with a complete Lean 4 formalization—including functional and imperative implementations, unit tests, correctness theorems, and machine-checkable proofs—emphasizing artifact completeness over sheer volume. Compared to VERINA, which decomposes the verification process into separate subtasks like spec generation and proof synthesis, VeriBench evaluates holistic translation performance: how well models can go from informal code and natural language to executable and provable formal artifacts. Moreover, VeriBench supports the evaluation of agentic systems that iteratively refine their outputs through feedback.

Evaluating Autoformalization. Zhang et al. (2025) decompose autoformalization evaluation into atomic properties judged by LLM ensembles and aggregated via a compensatory linear combination—effectively disjunctive, since high scores on some dimensions can mask failures on others—achieving moderate correlation with human assessments ($r = 0.479$ on Lean 4) without requiring gold references, though validated only on mathematical autoformalization with a single annotator. Our SCSC metric instead employs a non-compensatory geometric mean that enforces conjunctive semantics: any critical failure zeros the total score, matching the all-or-nothing nature of formal verification (see Appendix for extended discussion).

3. VeriBench

VeriBench is a Python-to-Lean 4 benchmark for end-to-end formal code verification. Each task begins with an executable Python file containing a docstring, reference implementation, and tests. The model must translate this input into a structured Lean 4 artifact—an implementation, tests, formal theorem statements, and proof attempts that the Lean kernel either discharges or accepts as declarations with explicit `sorry` placeholders. We use "machine-checkable" to mean kernel-typechecked: every theorem statement and proof obligation is verified by Lean, and gold proofs that still contain `sorry` are reported transparently rather than counted as discharged (see D_2 in Section 4). Unlike proof-completion or template-filling benchmarks, VeriBench asks models to infer the intended behavior of developer-written code and express it as verifiable Lean properties. It does so through a shallow embedding of each Python task, preserving source-level intent without requiring a full formal semantics of Python.

Composition. VeriBench is organized into five task families that progress from introductory reasoning to classical algorithms, security-critical programs, and production utility code. Together, these splits cover both small, interpretable properties and richer invariants over real-world software.

- **VeriBench-HumanEval**: 56 problems from the HumanEval benchmark (Chen et al., 2021). We extract signatures, docstrings, canonical implementations, and tests; package them as executable Python files; add edge cases where useful; and translate them into Lean 4 artifacts.
- **VeriBench-EasySet**: 41 small logic and introductory programming tasks, including factorial, list reversal, palindrome checking, maximum computation, and character counting. This split gives a controlled setting where failures are easy to diagnose.
- **VeriBench-CSSet**: 13 classical data-structure and algorithm tasks, including sorting, searching, and string manipulation. These programs are easy to implement but hard to verify, since their specifications require invariants such as sortedness, search completeness, and optimality.
- **VeriBench-SecuritySet**: 28 tasks adapted from MIT 6.858 labs, covering buffer overflows, privilege escalation, and race conditions. These examples require formalizations that expose safety preconditions and failure behavior.
- **VeriBench-RealCodeSet**: 32 functions adapted from Python standard-library modules, including `bisect.py` and `heapq.py`. We use simplified single-function adaptations of each routine—preserving the source-level algorithm and contract while normalizing types, removing module-level dependencies, and trimming corner-case handling unrelated to the verified property—rather than the verbatim CPython source. This split tests verification on routines developers commonly rely on, such as heap operations whose correctness depends on data-structure invariants.

Task format. Each VeriBench task pairs an executable Python reference file with a gold Lean 4 formalization. The Python file contains a concise docstring, a reference implementation, and unit tests; when needed, it states an input pre-condition and checks it before the main computation. The Lean artifact follows a fixed schema: functional implementation, unit tests, pre-condition, property theorems, post-condition, correctness theorem, and, when appropriate, an imperative implementation with tests and an equivalence theorem. This schema separates executable behavior, admissible inputs, atomic proof obligations, and bundled correctness claims, while supporting partial-credit evaluation when generated artifacts decompose the theorem set differently from the gold file.

Curation and provenance. For each task, curators write correctness theorems that capture intended behavior without merely restating the implementation. Because no finite benchmark can guarantee complete semantic coverage for arbitrary programs, VeriBench targets instance-level completeness: curators identify the central properties for each task and make them explicit. **Curation pipeline.** After `o3` drafts a Lean artifact, a human curator opens a GitHub issue with the draft. A second annotator—working independently from the issue opener—reviews the task against a checklist (theorem statements characterize behavior rather than restating the implementation; pre/post-conditions are non-trivial; proofs either compile or carry an explicit `sorry`; tests cover positive, negative, and edge cases), expands or repairs the theorem set, and merges only after the open issues are resolved. Disagreements between the two annotators are resolved by discussion in the GitHub issue thread, with the curated artifact reflecting the consensus version that lands on `main`. These edits often change types, theorem statements, or proof strategies; the repository history records the resulting provenance, and Appendix C reports per-task substantive edit fractions. We deliberately use human curation rather than a semantics-preserving Python-to-Lean translator (in the spirit of LeanIO frontends or CompCert-style extracted semantics): such pipelines are appropriate when the goal is to certify a single program against source-language semantics under all inputs, whereas VeriBench evaluates an agent's ability to *choose* a faithful Lean representation and *state* the right theorem boundaries from program intent. Curation does not provide a meta-theorem that the Lean artifact preserves Python semantics; this is an explicit trade-off discussed further in Appendix I.

Validity. VeriBench mitigates common risks in LLM-assisted benchmark construction through human curation,

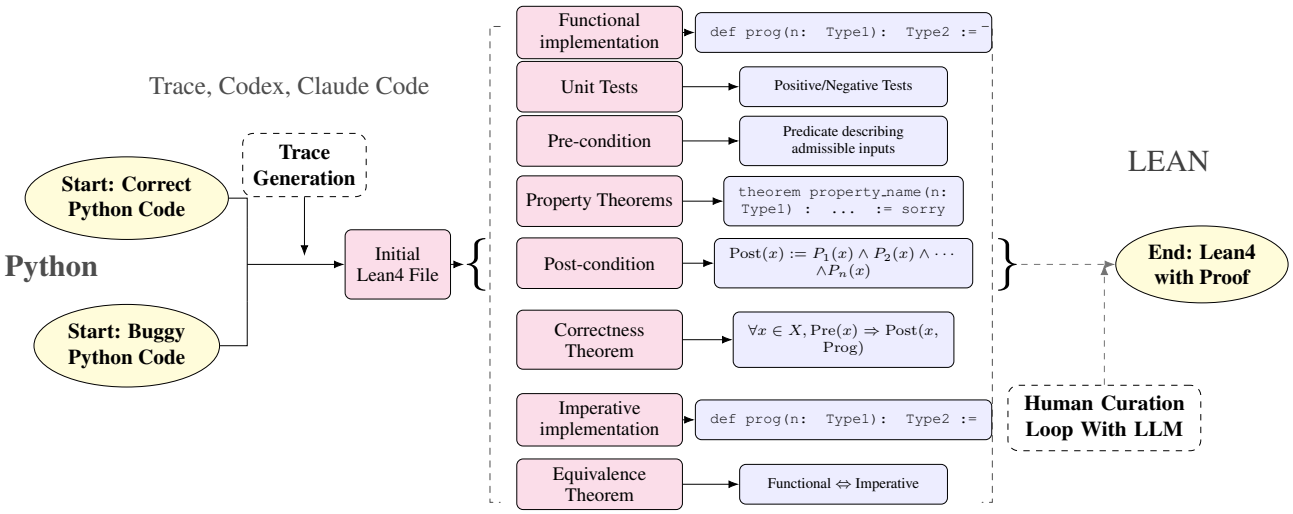

Figure 1. VeriBench construction pipeline: executable Python tasks are curated into standardized Lean 4 gold artifacts with implementations, tests, specifications, and proofs.

compiler checks, and execution-based validation. Every Python reference file has passing unit tests. Every gold Lean file must compile, so the Lean kernel checks the implementations, tests, theorem statements, and proofs. We also translate Python unit tests into Lean and verify that the gold Lean implementations satisfy them, providing a direct cross-language check between source programs and formal artifacts. Because curator edits routinely alter the draft artifacts, the released tasks are not simply raw model outputs.

Utility for evaluation. Prior work suggests that model rankings are often stable under modest benchmark noise (Salaudeen & Hardt, 2024). This supports comparative evaluation with VeriBench, while absolute scores should be read relative to the released specifications. Appendix C gives the construction pipeline, gold-file schema, Python input standard, and additional validity discussion.

Illustrative example. Figure 2 shows a simplified VeriBench-EasySet task together with its Lean gold formalization. The example illustrates the benchmark's central pattern: an executable Python reference on the left and a structured Lean artifact with tests and theorems on the right. Beyond simple value-level reasoning, VeriBench's `security_6858` split contains paired safe/unsafe tasks that exercise reference semantics, mutable buffer manipulation, and bounds-safety pre-conditions; an end-to-end heap/aliasing case study (gold Lean `Buffer` model, safety predicate, agent output, and per-mode TC_1 resolution) appears in Appendix J.

4. Evaluation Metric: The Smooth Conjunctive Score for Code Verification

To rigorously assess whether an AI agent has successfully autoformalised a Python program into Lean 4, one must verify that the agent's formalization is both internally sound and faithfully aligned with the human-curated gold reference. In VERIBENCH, evaluation goes beyond checking whether generated code compiles; it requires a multifaceted conjunction of constraints—approximated by a product that serves a dual role: like the chain rule of conditional probability, it decomposes the overall evaluation into a sequence of dependent factors, and like a logical conjunction, a single zero in any factor collapses the entire score.

Each benchmark problem involves two artifacts: a human-curated gold-standard Lean 4 formalization L^* (with implementation $L^*.f$, theorems $L^*.\mathrm{Thms}$, and tests $L^*.T$), and the agent-generated Lean 4 formalization $\hat{L}$ (with implementation $\hat{L}.f$, theorems $\hat{L}.\mathrm{Thms}$, and tests $\hat{L}.T$). We reserve the * superscript for artifacts that are both *correct* (bug-free) and *comprehensive* (covering all intended properties and tests).[1] The agent's output $\hat{L}$ is the primary object of evaluation.

The ideal score. We organize the evaluation into two conceptual sub-blocks within a *correctness* block $\mathcal{C}$: *internal consistency* IC, which checks that the agent's own artifacts are mutually coherent; and *theorem coverage* TC, which checks that the agent's theorems cover the obligations stated by the gold reference. We additionally include a *data quality* block $\mathcal{D}$, which checks that the gold Lean artifact is itself internally correct before it is used to judge the agent. The

[1] In formal methods, *correct* corresponds to *soundness* and *comprehensive* to *completeness*.

Algorithm 1 Input Python file (EasySet).

```python
# Implementation
def my_max(a: int, b: int) -> int:
    """
    Return the larger of two non-negative integers.
    """
    return b if a <= b else a
# Tests
from typing import Callable
def check(candidate: Callable[[int, int], int]) ->
     bool:
    assert candidate(7, 3) == 7
    # ... (other tests) ...
    return True
if __name__ == "__main__":
    assert check(my_max), f"Failed: {__file__}"
    print(f'All tests passed: {__file__}!')
```

Algorithm 2 Gold Lean 4 file (EasySet).

```
namespace MyMax
-- Implementation
def myMax (a b : Nat) : Nat :=
 if _ : a \le b then b else a
-- Tests
#eval myMax 7 3           -- expect 7
example : myMax 7 3 = 7 := by native_decide
-- Theorems
/-- Commutativity: Order does not matter. -/
@[simp] theorem max_commutativity (a b : Nat) : myMax
     a b = myMax b a := sorry
/-- Idempotence: Max of self is self. -/
@[simp] theorem max_idempotent (a : Nat) : myMax a a =
     a := sorry
end MyMax
```

Figure 2. Parallel VeriBench example (simplified for demonstration). Left: executable Python input. Right: gold Lean 4 formalization for the same EasySet task.

ideal score is a strict logical conjunction over these blocks:

$$S^* = \mathcal{C}^* \ \wedge \ \mathcal{D}^* = \mathrm{IC}^* \ \wedge \ \mathrm{TC}^* \ \wedge \ \mathcal{D}^* \tag{1}$$

In practice, each sub-block decomposes into a product of fractional components, and the conjunction is approximated by a product. We denote the finite-budget raw product with a tilde $\widetilde{(\cdot)}$:

$$\widetilde{S}_{\text{raw}} = \widetilde{\mathcal{C}} \ \cdot \ \widetilde{\mathcal{D}} = \widetilde{\mathrm{IC}} \ \cdot \ \widetilde{\mathrm{TC}} \ \cdot \ \widetilde{\mathcal{D}} \tag{2}$$

Concretely, $\equiv_{\text{eq}}$ returns the approximate coverage overlap between two sets of artifacts, $\vdash_{\text{pf}}$ returns the fraction of theorems the prover successfully discharges within timeout t_{pr}, and $\vdash_{\text{p}}$ returns the fraction of tests the implementation passes. In the ideal case with infinite budgets and a perfect prover, each component is binary and the product recovers the exact conjunction; with finite budgets, the fractional values degrade smoothly. Distinct test budgets (e.g., n_p vs. n_p'') indicate *disjoint* test suites, ensuring no two factors are numerically redundant by construction.

Correctness ($\widetilde{\mathcal{C}}$). The correctness score measures the quality of the agent's formalization across three factors (see Figure 3):

$$\widetilde{\mathcal{C}} = \underbrace{\hat{L}.T \ \vdash_{\text{p}}^{n_p} \ \hat{L}.f}_{(\mathrm{IC}_1)\text{ agent tests on agent code}} \cdot \underbrace{\hat{L}.\text{Thms} \ \vdash_{\text{pf}}^{t_{\text{pr}}} \ \hat{L}.f}_{(\mathrm{IC}_2)\text{ agent thms prove agent code}} \cdot \underbrace{\hat{L}.\text{Thms} \ \equiv_{\text{eq}}^{n_{\text{eq}}} \ L^*.\text{Thms}}_{(\mathrm{TC}_1)\text{ agent specs cover gold specs}} \tag{3}$$

Borrowing the turnstile ($\vdash$) from logic, $\vdash_{\text{p}}^{n_p}$ denotes test passing and returns the fraction of n_p tests passed, while $\vdash_{\text{pf}}^{t_{\text{pr}}}$ denotes formal proving and returns the fraction of theorems discharged within timeout t_{pr}. $\equiv_{\text{eq}}^{n_{\text{eq}}}$ measures approximate coverage overlap between two sets of artifacts—not semantic equivalence in the formal-methods sense, which is undecidable in general.

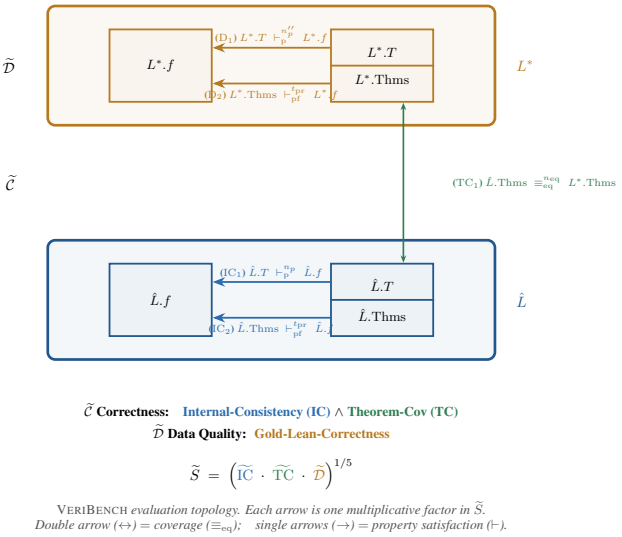

Figure 3. **VeriBench evaluation topology.** Each solid arrow is one multiplicative factor in $\widetilde{S} = \left(\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}\right)^{1/5}$. **Blue arrows (IC)**: agent's tests and theorems evaluated against agent's own code ($\hat{L}$, blue box). **Green double-arrow (TC)**: agent's theorems compared against the gold theorems $L^*.\text{Thms}$ (gold box). **Gold arrows ($\widetilde{\mathcal{D}}$)**: gold tests and theorems evaluated against the gold code L^*.

Internal Consistency ($\widetilde{\mathrm{IC}}$). The first two factors (IC_1–IC_2) check that the agent's own tests pass and that its own theorems are provable over its implementation. These are necessary but not sufficient: an agent whose own artifacts are mutually inconsistent is clearly unsound regardless of how it compares to the gold reference. Critically, the agent's internal artifacts could be trivially weak—a single vacuous unit test or a theorem equivalent to True. Such an agent would score perfectly on internal consistency while having verified nothing meaningful; TC guards against this.

Theorem Coverage ($\widetilde{\mathrm{TC}}$). Factor (TC_1) evaluates the agent's ability to *generate* correct specifications, not merely produce a compilable implementation. The coverage check

$\hat{L}.\text{Thms} \equiv_{\text{eq}}^{n_{\text{eq}}} L^*.\text{Thms}$ runs in one of three modes, in order of decreasing rigor, and the mode used is recorded per item alongside the score:

1. **Prover-established entailment.** For each gold theorem we attempt to derive it from the agent's theorem set via the Lean 4 prover (within timeout t_{pr}), and vice versa. A success here is a kernel-checked formal entailment.
2. **Syntactic / structural matching.** When statements are alpha-equivalent or differ only by trivial rewrites (renaming, currying, unit-shuffling), structural matching accepts the pair without invoking the prover.
3. **LLM-estimated semantic coverage.** When neither (1) nor (2) succeeds within budget, we fall back to an LLM-based semantic similarity judge calibrated against five independent human raters (Pearson $r = 0.70$, Appendix F). This mode produces an empirical proxy for coverage, not a formal proof of entailment, and its results are reported as such.

In our current evaluation, the LLM-judge mode dominates reported TC_1 values—formal cross-implication times out for most non-trivial pairs—so headline TC_1 should be read as "human-aligned semantic coverage estimate," not formal entailment.

Why coverage, not equivalence. We credit the agent whenever its theorems *cover* $L^*.\text{Thms}$—i.e., when a stronger agent theorem entails the gold obligation. A stronger agent theorem entails $L^*.\text{Thms}$, so it meets gold's obligations and more—crediting it rewards beating the reference instead of penalizing over-delivery. Strict equivalence would instead force the agent to match gold verbatim, punishing any additional guarantee the agent discovers. The name *Theorem Coverage* reflects this one-directional reading of the check: gold-side obligations are covered, not that the two specifications are logically equal.

Why not cross-apply gold artifacts to agent code? An earlier design included factors that applied gold tests and gold theorems directly to the agent's code ($L^*.T \vdash \hat{L}.f$ and $L^*.\text{Thms} \vdash \hat{L}.f$), as well as a Jaccard test-equivalence factor ($\hat{L}.T \equiv L^*.T$). We dropped all three for principled reasons. First, *implementation mismatch*: $\hat{L}.f$ and $L^*.f$ may use different representations (e.g., arrays vs. lists, curried vs. uncurried signatures), causing gold tests and theorems to fail at type-checking and producing systematic false negatives unrelated to correctness. Second, *test-equivalence is ill-defined*: the agent and gold generate structurally different test suites by design; Jaccard comparison over these heterogeneous sets is meaningless. Third, $\text{IC}_2 \wedge \text{TC}_1$ *provides an empirical proxy for the cross-proves check*: if the agent's theorems prove the agent's code (IC_2) and the agent's theorems align with the gold theorems (TC_1), then to the extent that the TC_1 alignment reflects genuine semantic coverage, the gold specification approximately holds for the agent's code. We emphasize that this is a probabilistic correlation, not a formal transitivity argument: TC_1 in mode 3 (Section above) is an LLM-judge estimate calibrated against humans, so $\text{IC}_2 \wedge \text{TC}_1$ should be read as "the agent's stated properties resemble the gold obligations and prove its own code," not as a kernel-checked proof that gold theorems hold of agent code. This empirical reading still avoids the practical difficulty of applying gold artifacts to agent code with mismatched signatures and internal structure.

Data Quality ($\widetilde{\mathcal{D}}$). The gold Lean artifact must also certify itself before it is trusted as a judge. We therefore include two gold-internal factors:

$$\widetilde{\mathcal{D}} = \underbrace{L^*.T \vdash_{\text{p}}^{n_p''} L^*.f}_{(\text{D}_1)\text{ gold tests on gold code}} \cdot \underbrace{L^*.\text{Thms} \vdash_{\text{pf}}^{t_{\text{pr}}} L^*.f}_{(\text{D}_2)\text{ gold thms prove gold code}} \quad (4)$$

These terms ensure the benchmark's gold Lean reference is internally correct, not merely assumed correct by construction.

Summary and scoring. The full evaluation decomposes into 5 factors across three sub-blocks: IC (IC_1, IC_2), TC (TC_1), and $\mathcal{D}$ (D_1, D_2). The raw product $\widetilde{S}_{\text{raw}} = \widetilde{\text{IC}} \cdot \widetilde{\text{TC}} \cdot \widetilde{\mathcal{D}}$ enforces approximate "AND" logic: a zero in any single factor collapses the entire score. However, the raw product compresses dynamic range—uniformly solid per-factor performance of 0.8 yields $0.8^5 \approx 0.33$, making it hard to distinguish competent from mediocre agents. We resolve this by reporting the *geometric mean* over all 5 factors as the primary score:

$$\widetilde{S} = \widetilde{S}_{\text{raw}}^{1/5} = \left(\widetilde{\text{IC}} \cdot \widetilde{\text{TC}} \cdot \widetilde{\mathcal{D}}\right)^{1/5} \quad (5)$$

An agent scoring 0.8 on every factor receives $\widetilde{S} = 0.8$ rather than 0.33, directly reflecting average per-factor quality while preserving the zero-collapses property.

Interpretability. Individual factors are reported separately alongside $\widetilde{S}$, making the metric interpretable: researchers can diagnose whether a low score stems from faulty code generation ($\widetilde{\text{IC}}$), missing or poor-quality specifications ($\widetilde{\text{TC}}$), or a flawed gold Lean reference ($\widetilde{\mathcal{D}}$). The final benchmark score is the macro-average over N problems: $\widetilde{S} = \frac{1}{N}\sum_{i=1}^{N} \widetilde{S}_i$.

Limitations. Metric fidelity is bounded by gold-theorem comprehensiveness: if $L^*.\text{Thms}$ does not cover some behavior of $L^*.f$, the empirical chain $\text{IC}_2 \wedge \text{TC}_1$ will not surface bugs on uncovered inputs. This is an inherent limitation of any specification-based metric—an explicit cross-proves

factor (gold theorems applied directly to agent code) would equally miss behaviors not captured by the gold theorems. We mitigate this through careful curation and the D_1/D_2 integrity gates, which ensure the gold reference is internally correct before it is used as a judge.

Alternative reporting: agent skill vs. benchmark item quality. A reviewer may reasonably argue that bundling D_1, D_2 into the headline agent score $\widetilde{S}$ confounds two distinct questions—"how good is the agent?" and "how trustworthy is this benchmark item?"—and that the cleaner accounting is to separate them. Under that decomposition, agent skill is $\widetilde{S}_{\text{skill}} = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1)^{1/3}$ reported *conditional on* a gold-validity gate $D_1 = D_2 = 1$ (or a soft threshold), and benchmark-item quality is $\widetilde{Q}_{\text{gold}} = (D_1 \cdot D_2)^{1/2}$ reported separately. When discarding imperfect items is undesirable, $\widetilde{Q}_{\text{gold}}$ enters as a confidence weight in sensitivity analyses rather than as a multiplicative component of agent skill. We retain the bundled $\widetilde{S}$ as the primary headline because it preserves a single zero-collapse property across all five factors, but plan to report $\widetilde{S}_{\text{skill}}$ alongside $\widetilde{Q}_{\text{gold}}$ in Appendix E for transparency.

5. Evaluation

We evaluate four frontier agents on the full Python-to-Lean translation task, where each agent receives the Python source file and must produce a structured Lean 4 artifact (implementation, tests, theorems, and proofs): **Codex** (GPT-5.4), OpenAI's coding agent; **Claude Code** (Sonnet 4.6), Anthropic's agentic coding CLI; **Baseline** (Sonnet 4.6), single-shot prompting without tool use; and **Leanstral (v2)**, a Lean-specialized model. Each agent operates autonomously: it receives only the Python file and must generate the complete Lean formalization end-to-end.

Experimental setup. All agents operate in the same evaluation harness against Lean 4 toolchain `leanprover/lean4:v4.22.0` with Mathlib `v4.22.0`, receiving a single Python file and a fixed system prompt to produce a Lean artifact following the schema in Section 3. Outputs are parsed by extracting the last fenced Lean block from the agent's transcript and compiled with `lake env lean`; failure to compile counts toward IC_1, not toward task non-completion. Generated proofs may use `sorry` (and contribute to IC_2 accordingly); `axiom` declarations and unsafe escape hatches are scanned and reported per-task in Appendix E, alongside hardware and compute budgets. Each agent runs under a fixed 1-hour per-task budget that absorbs within-budget compile/edit/retry as the agentic analogue of pass@k sampling. The deployed TC_1 judge is `gpt-5.3-codex`; one agent (Codex GPT-5.4) is in the same family—a self-bias risk discussed in Appendix I.

Agent	IC_1	IC_2	TC_1	D_1	D_2	$\widetilde{S}$
Oracle (gold)	0.898	0.604	1.000	0.898	0.604	0.783
Codex (GPT-5.4)	1.000	0.237	0.102	0.921	0.570	**0.417**
Claude Code (Sonnet 4.6)	1.000	0.114	0.098	0.921	0.568	0.358
Baseline (Sonnet 4.6)	0.310	0.282	0.105	0.923	0.567	0.344
Leanstral (v2)	0.292	0.065	0.058	0.923	0.567	0.225

Table 2. SCSC scores on VERIBENCH. $\widetilde{S} = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1 \cdot D_1 \cdot D_2)^{1/5}$. The oracle row uses the gold reference as both agent output and target; its sub-unity score ($\widetilde{S} = 0.783$) reflects incomplete proofs ($D_2 = 0.604$) in the current gold set.

Agent	Easy	CS	Real	HE	Sec.	$\widetilde{S}$
Oracle (gold)	0.884	0.746	0.749	0.724	0.688	0.783
Codex (GPT-5.4)	**0.443**	**0.321**	**0.414**	**0.409**	**0.411**	**0.417**
Claude Code (Sonnet 4.6)	0.407	0.291	0.369	0.309	0.358	0.358
Baseline (Sonnet 4.6)	0.435	0.000	0.306	0.328	0.333	0.344
Leanstral (v2)	0.300	0.146	0.178	0.206	0.172	0.225

Table 3. Per-split SCSC scores ($\widetilde{S}$) on VERIBENCH. The CS Set is hardest for agents, with Baseline scoring 0.000 due to zero-collapse. Even the Easy Set saturates below 0.45.

5.1. Overall Results

Reporting policy. Table 2 reports SCSC *conditional on the agent producing output* across 176 gold Lean files (n ranges from 165 to 176 per agent); a no-output-as-zero full-benchmark variant on the fixed denominator $N = 176$ is reported in Appendix E.

Theorem coverage ($\text{TC}_1 \leq 0.105$) is the dominant bottleneck, and internal consistency splits sharply: Codex and Claude Code achieve $\text{IC}_1 = 1.0$ but $\text{IC}_2 \leq 0.237$, revealing agents produce plausible test suites but struggle to generate sound proofs; the oracle ceiling is $\widetilde{S} = 0.783$ (not 1.0) because $\sim$40% of gold theorems still carry `sorry`. Codex (GPT-5.4) leads at $\widetilde{S} = 0.417$, followed by Claude Code (0.358), Baseline (0.344), and Leanstral v2 (0.225).

5.2. Per-Split Analysis

Table 3 breaks down $\widetilde{S}$ by dataset split.

The CS Set is hardest (Baseline scores 0.000 via zero-collapse; even Codex reaches only 0.321); Easy Set is most accessible ($\widetilde{S}$ up to 0.443); Security, RealCode, and HumanEval cluster in 0.3–0.41. No agent's per-split $\widetilde{S}$ exceeds 0.45, confirming that end-to-end formal code verification from Python remains a wide-open challenge; Appendix E extends this to all completed agent runs.

References

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. Improver: Agent-based automated proof optimization. *arXiv preprint arXiv:2410.04753*, 2024.

Aleithan, R., Xie, H., Lu, S., Ouellette, B. A., and Wang, S. SWE-Bench+: Enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024.

Aniva, L., Sun, C., Miranda, B., Barrett, C., and Koyejo, S. Pantograph: A machine-to-machine interaction interface for advanced theorem proving, high level reasoning, and data extraction in lean 4. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 104–123. Springer, 2025.

Anonymous Authors. AI coding benchmarks need proofs, not just tests, 2026. Anonymized for double-blind review.

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. *arXiv preprint arXiv:2509.22908*, 2025.

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. *arXiv preprint arXiv:2107.03374*, 2021.

Cheng, C.-A., Nie, A., and Swaminathan, A. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. In *Advances in Neural Information Processing Systems*, volume 37, pp. 71596–71642, 2024.

CSAIL, M. 6.858 computer systems security. URL https://ocw.mit.edu/courses/6-858-computer-systems-security-fall-2014/.

Dougherty, Q. and Mehta, R. Proving the coding interview: A benchmark for formally verified code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 72–79. IEEE, 2025.

Gao, G., Zhang, Y., Xu, J., Jiang, A., and Welleck, S. Herald: A natural language annotated lean 4 dataset. *arXiv preprint arXiv:2410.10878*, 2024.

GitHub. The state of open source and ai. *The Octoverse*, 2024. URL https://github.blog/news-insights/octoverse/octoverse-2024/. Accessed: 2025-02-06.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

Jiang, Z., Pajic, M., Moarref, S., Alur, R., and Mangharam, R. Modeling and verification of a dual chamber implantable pacemaker. In *International conference on tools and algorithms for the construction and analysis of systems*, pp. 188–203. Springer, 2012.

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? In *International Conference on Learning Representations*, 2024.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. Dspy: Compiling declarative language model calls into self-improving pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

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. *Advances in Neural Information Processing Systems*, 2023.

Loughridge, C., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024.

OpenAI. Introducing SWE-bench Verified. Technical report, 2024.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022. doi: 10.48550/arXiv.2108.09293. URL https://arxiv.org/abs/2108.09293.

Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, pp. 2785–2799, 2023.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290*, 2023.

Reid, A. F. End-to-end verification of arm® processors with ISA-Formal. In *Computer Aided Verification (CAV)*, 2016. URL https://alastairreid.github.io/papers/cav2016_isa_formal.pdf.

Salaudeen, O. and Hardt, M. Imagenot: A contrast with imagenet preserves model rankings, 2024. URL https://arxiv.org/abs/2404.02112.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzche, S., Durrett, G., Yue, Y., and Chaudhuri, S. Clever: A curated benchmark for formally verified code generation. *arXiv preprint arXiv:2505.13938*, 2025.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. *arXiv preprint arXiv:2407.03203*, 2024.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Zhang, L., Valentino, M., and Freitas, A. Beyond gold standards: Epistemic ensemble of llm judges for formal mathematical reasoning. *arXiv preprint arXiv:2506.10903*, 2025.

A. Technical Appendices and Supplementary Material

B. Related Work (Cont.)

Vericoding Benchmarks: Bursuc et al. (Bursuc et al., 2025) introduce vericoding—generating a formally verified implementation and proof from a pre-existing formal specification in Lean, Dafny, or Verus—and release the largest such benchmark to date with 12,504 tasks. Starting from a formal specification is a genuine strength: when the spec is correct, downstream proofs inherit its precision and bugs are caught mechanically. However, the formal spec is the most trust-critical artifact to get right—a subtly wrong spec propagates silently into every downstream proof, producing formally verified code that is verified against the wrong thing. VERIBENCH instead anchors evaluation to real Python programs, which play an analogous role to a formal spec—encoding developer intent concisely—while remaining executable and grounding verification in real-world behavior. Like a formal spec, a Python program can contain bugs; VERIBENCH's could introduce a block $\tilde{D}$ that handles this symmetrically, discounting benchmark problems where the Python source and gold Lean reference disagree. Beyond this, vericoding benchmarks are constructed from narrow formal verification datasets (DafnyBench, VERINA, CLEVER), leaving the vast corpus of existing open-source Python code entirely out of scope.

Herald: Herald (Gao et al., 2024) introduces a pipeline for generating natural language–formal language (NL-FL) paired datasets from Mathlib4 by leveraging compiler metadata—dependency graphs, proof states, and tactic explanations—to improve LLM-based informalization of formal math statements and proofs. Their dependency-level translation ordering and tactic-based augmentation yield 580k NL-FL statement pairs, and a formalizer fine-tuned on this data achieves 96.7% pass@128 on miniF2F-test. However, Herald operates entirely within the mathematical autoformalization setting: it translates pre-existing formal Lean theorems into natural language and back, whereas VERIBENCH requires agents to start from executable Python code and autonomously synthesize specifications, implementations, and proofs in Lean 4—a strictly different task in which no gold formal artifact is available to condition on.

Autoformalization. Complementing this line of work, Wang et al. (2024) present THEOREMLLAMA, a framework aimed at enhancing LLM translation into Lean 4. Drawing on over 100K proof examples from the Mathlib4 library, TheoremLlama employs a novel natural-language-to-formal-language (NL-FL) bootstrapping strategy and iterative proof synthesis. This enables the reuse of verified examples as templates for future translations. The framework achieves 36.48% and 33.61% accuracy on the MiniF2F-Valid and MiniF2F-Test benchmarks, respectively—surpassing GPT-4 by more than ten percentage points on both.

Techniques for Code Verification. IMPROVER (Ahuja et al., 2024) introduces a Lean-aware Chain-of-States prompting loop that integrates retrieval, best-of-n sampling, and iterative correction to rewrite formal proofs with improved properties. By optimizing for metrics such as brevity and readability, ImProver reduces the number of tactics by half, doubles proof readability, and boosts theorem prover acceptance rates by over 80%.

Targeting verified generation, CLOVER (Sun et al., 2024) primarily contributes a closed-loop consistency checking *method* for Dafny, utilizing a dataset of isolated code-specification-docstring triplets. In contrast, VERIBENCH targets the autonomous **end-to-end autoformalization** of full Python files. Our benchmark requires agents to generate the complete formal artifact (Lean code, specification, and docstring) by architecting a formal model that faithfully verifies the original software, rather than synthesizing fresh implementations from pre-packaged components.

Agentic Frameworks and Tools for Code Verification. TRACE (Cheng et al., 2024) proposes *generative optimization*, tuning entire computational workflows—including code, prompts, tool calls, and error signals—by treating execution traces as gradients in the OPTO framework. With a PyTorch-like API and the LLM-based optimizer OptoPrime, it supports diverse tasks such as prompt tuning, debugging, and robot control, rivaling specialized optimizers. Building on modular composition, DSPY (Khattab et al., 2024) treats LLM calls as declarative modules in a computational graph. Users define concise input–output signatures, and DSPy's compiler automatically bootstraps or fine-tunes pipelines using built-in "teleprompters." This enables a few-line programs to outperform expert-crafted prompts in math, QA, and agent workflows. Extending agentic capabilities to formal reasoning, PANTOGRAPH (Aniva et al., 2025) offers a programmatic interface to Lean 4 with support for advanced proof search. It exposes internal proof states and tactics for integration with learning agents, replacing human-facing interfaces with API-level control.

Evaluating Autoformalization with Reference-Free Judges. Zhang et al. (2025) decompose autoformalization evaluation into atomic properties—logical preservation, mathematical consistency, formal validity, and formal quality—judged by LLM ensembles and aggregated via a compensatory linear combination, notably without requiring gold-standard references. Their best configuration achieves moderate correlation with human assessments (Pearson $r = 0.662$ on Isabelle/HOL, $r = 0.479$ on Lean 4), with mathematical consistency correlation near zero on Lean 4 ($r = 0.047$ for direct judgment, $r = -0.011$ for OAP synthesis), and Cohen's κ between LLM judges and theorem provers never exceeding 0.30—all validated against a single human annotator on mathematical autoformalization tasks only, with no code verification evaluation. Their motivation for abandoning gold-standard comparison rests on a small, non-random sample of formalizations pre-selected by the very LLM whose reliability is uncertified, despite evidence that benchmark noise rarely distorts relative model rankings (Salaudeen & Hardt, 2024). More fundamentally, a compensatory aggregation is effectively disjunctive: high scores on some dimensions can offset failures on others, so a formalization that is mathematically inconsistent can still receive a passing score if its formal quality is rated highly. This is misaligned with formal verification, where a single critical defect invalidates the entire artifact. In contrast, our SCSC metric employs a non-compensatory geometric mean that enforces conjunctive semantics—any factor scoring zero zeros the total—smoothly approximating the all-or-nothing correctness guarantees demanded by formal code verification.

C. Additional VeriBench Details

Appendix overview. This appendix provides additional detail on how VERIBENCH is constructed, how Python reference files and Lean 4 gold files are standardized, and why this structure supports reliable evaluation.

C.1. Construction Pipeline

HumanEval. The HumanEval split extends the original HumanEval benchmark (Chen et al., 2021) into Lean 4 formal verification tasks. For each problem, we parse the Python function signature, docstring, canonical implementation, and unit tests. These elements are assembled into a standalone executable Python file, with additional edge-case tests added where useful. The corresponding Lean 4 file uses a shallow embedding of the same task and includes a function definition, natural-language documentation, executable tests written as `#eval` expressions and `example` theorems, and one or more formal correctness properties. When appropriate, the Lean artifact also includes an imperative implementation and an equivalence theorem relating it to the functional version.

EasySet. EasySet provides a simplified alternative to HumanEval for foundational programming and reasoning skills. Its tasks resemble short introductory programming exercises, including factorial, list reversal, palindrome checking, maximum computation, and character-frequency counting. Each task is small and self-contained, with Python and Lean 4 implementations, tests, and proof obligations that are easy to inspect but still require precise formalization.

CSSet. CSSet contains classical data-structure and algorithm problems from undergraduate computer science, including sorting, searching, dynamic programming, and string manipulation. For algorithms such as sorting, the split includes multiple algorithmic styles, from simple quadratic methods to divide-and-conquer approaches. Although such algorithms are often easy for LLMs to implement, formally verifying them requires richer invariants such as sortedness, permutation preservation, search completeness, and optimality.

SecuritySet. SecuritySet adapts programs from MIT 6.858 labs to expose models to security-critical behavior, including buffer overflows, privilege escalation, and race conditions. The Lean translations make safety preconditions and failure behavior explicit, requiring models to reason about the boundary between safe and unsafe executions rather than merely reproduce executable behavior. This split is intended to reflect high-assurance verification tasks that matter in practice.

RealCodeSet. RealCodeSet contains production-grade functions taken from the Python standard library, including routines from `bisect.py` and `heapq.py`. For example, a task based on `heappush` requires preserving the heap invariant after inserting an element. By incorporating widely used utility code, this split tests whether models can reason about optimized implementations whose correctness depends on nontrivial data-structure invariants.

Curation. Across all splits, curators aim to include the central semantic properties for each task. Because no practical benchmark can guarantee a complete theorem list for arbitrary programs, VERIBENCH uses a two-stage process: an

AI-assisted draft is followed by a second human curation pass, also AI-assisted, that expands and checks the theorem set. Curators revise implementations, types, theorem statements, and proof strategies as needed before accepting a task.

C.2. Gold-Standard Lean 4 Files

Purpose. Each gold file specifies the intended behavior of a Lean implementation. It contains the functional implementation, an imperative implementation when appropriate, a declarative *Pre-condition*, separate algebraic or semantic property theorems, a conjunctive *Post-condition*, a bundled *Correctness* theorem ($\mathrm{Pre} \to \mathrm{Post}$), and, when both implementations are present, an *Equivalence* theorem. This structure supports partial-credit evaluation and robust comparison across different theorem decompositions.

Required order. Each gold file follows a fixed order:

1. **Implementation.** A pure or functional definition of the target program.
2. **Unit tests.** Executable examples covering edge, positive, and negative cases. Positive tests check valid input-output pairs satisfying the pre-condition and intended properties; negative tests check claims or inputs that violate a predicate or safety condition.
3. **Pre-condition.** A predicate $\mathrm{Pre} : \textit{Input} \to \mathsf{Prop}$ defining admissible inputs. If the type already enforces the relevant constraint, Pre can be equivalent to `True`.
4. **Properties.** Atomic semantic properties stated as individual theorems, such as identity, commutativity, associativity, bounds, sortedness, safety, or preservation invariants.
5. **Post-condition.** A bundled specification, typically written as $Post(x) := P_1(x) \wedge P_2(x) \wedge \cdots \wedge P_n(x)$.
6. **Correctness.** A theorem stating that every admissible input satisfies the post-condition, usually of the form $\mathrm{Pre}(x) \Rightarrow Post(x, Prog)$.
7. **Imperative equivalence.** When an imperative implementation is included, the file adds imperative tests and proves equivalence with the functional implementation.

This ordering separates executable behavior, admissible inputs, atomic proof obligations, and bundled correctness statements, making it easier to compare candidates that package their theorems differently.

C.3. Standard for Correct Python Input Files

Scope. Each executable Python reference file accompanies a Lean gold file. It provides an operational reference for the task through code, tests, and, when needed, explicit pre-condition checks.

Required structure.

1. **Docstring.** One or two sentences describing the function's intent and any notable edge cases.
2. **Python pre-condition.** When needed, a pure predicate `pre(...)->bool` encoding admissible inputs.
3. **Implementation with early pre-check.** When a separate pre-condition is required, the implementation calls `pre(...)` before the main computation and raises `AssertionError` on invalid inputs.
4. **Unit tests.** A compact suite of positive tests on valid inputs and negative tests on invalid inputs.

C.4. Why VeriBench Does Not Claim a Complete Theorem List

One might hope to publish a fixed theorem set Σ and claim that proving every statement in Σ establishes complete correctness. For programs in Turing-complete languages, such a guarantee is unattainable in general. Rice's theorem implies that any nontrivial semantic property of programs is undecidable over all programs in a Turing-complete language. If a recursively

enumerable and sound theorem collection captured every true semantic statement about an arbitrary reference program, one could mechanically search that collection to decide many semantic properties, contradicting this limitation.

VeriBench therefore does not claim universal completeness. Instead, it pursues *instance-level completeness*: for many total, mathematically defined tasks, curators can identify the central correctness properties that characterize the intended behavior. The gold-file structure makes those properties explicit, decomposed, and auditable, while acknowledging that some true semantic facts may remain outside the released specification.

C.5. Additional Validity and Utility Discussion

Execution-based validation. We validate benchmark quality through executability rather than external auditing alone. Every task begins with a Python file whose unit tests pass. Every gold Lean file must compile; because the Lean kernel is a proof checker, successful compilation establishes that the implementation, tests, theorem statements, and proofs are type-correct and logically well-formed. We also manually translate Python unit tests into Lean and verify that the gold Lean implementations satisfy them, giving a direct cross-language check between the Python source and the Lean formalization.

Kernel-enforced correctness. A pull request must compile to be accepted. Because the Lean kernel is a proof checker, compilation implies that every implementation, unit test, theorem, and proof is logically sound. Frontier models struggle to compile these artifacts reliably even with agentic tool use, suggesting that benchmark construction required substantial human intervention and that many final tasks remain nontrivial for the systems used to draft them.

Manual audit with public provenance. After `o3` drafts each Lean 4 artifact, a curator opens a GitHub issue that is inspected by a second human reviewer, who edits the patch when needed and merges only after the issues are resolved. Every change, comment, and decision is preserved in the repository history, providing reproducible evidence of human oversight.

No leakage of final solutions. Curator edits routinely alter types, theorem statements, or proof strategies in ways the originating LLM cannot anticipate. The published tasks thus differ from the raw LLM output and are not trivially solvable by the same model.

Robustness to benchmark noise. Like any curated benchmark, VeriBench may contain imperfections in specification coverage or task design. However, prior work suggests that model rankings can remain stable under modest benchmark noise (Salaudeen & Hardt, 2024). This does not remove the need for careful auditing, but it supports the use of VeriBench for comparative evaluation even when absolute scores should be interpreted with appropriate caution.

Why the file structure matters.

- **Granular proof evaluation.** Separating the post-condition into individual theorems yields independent proof obligations and enables partial credit when only a subset is generated or proved.
- **Robust comparison.** Even if a generated file partitions properties differently from the gold file, its bundled *Correctness* theorem provides a common comparison target.
- **Clear separation of roles. Pre** states admissible inputs, each post-condition component is a theorem requiring proof, and **Correctness** bundles the resulting guarantees.
- **Imperative cross-check.** An imperative implementation plus an *Equivalence* theorem helps catch loop-to-recursion translation errors and strengthens confidence in the formalization.

D. Discussion on the Evaluation Metric

Why the geometric mean. Our score $\widetilde{S} = \left(\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}\right)^{1/5} = \left(\prod_{i=1}^{5} f_i\right)^{1/5}$ is the geometric mean of five atomic verification factors, each $f_i \in [0, 1]$ a soft truth value measuring the degree to which a verification condition holds. The product $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}} = \prod_{i=1}^{5} f_i$ is their smooth AND: the joint degree to which all five conditions hold simultaneously. The key insight is

that the $1/5$ exponent *distributes* over the product:

$$\left(\prod_{i=1}^{5} f_i\right)^{1/5} = \prod_{i=1}^{5} f_i^{1/5}.$$

Rather than any single factor contributing its full weight, each factor contributes its 5th root—its fair share of the joint truth. Equivalently, $\widetilde{S}$ answers: *what uniform per-conjunct truth value, applied five times multiplicatively, recovers the joint smooth AND $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}$?* That is, $\widetilde{S}^5 = \widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}$. This is the multiplicative analogue of the arithmetic mean, which asks what uniform value, summed n times, recovers the original sum.

Correcting multiplicative compression. Multiplying n soft truth values in $[0, 1]$ compresses the joint truth toward zero: a uniformly competent agent scoring 0.8 on every factor yields $0.8^5 \approx 0.33$, making it indistinguishable from a mediocre agent. The 5th root reverses this compression: $0.33^{1/5} = 0.8$, correctly recovering the average per-conjunct truth value of the smooth logical conjunction. An agent scoring 0.8 on every factor has a joint smooth-AND truth of $0.8^5 \approx 0.33$; the geometric mean inverts this to report 0.8—the true average degree to which each verification condition holds. Division by n corrects additive overshoot; the nth root corrects multiplicative compression. Both restore the result to the $[0, 1]$ scale of individual factors.

The hypercube volume intuition. The five soft truth values $\{f_i\}_{i=1}^5$ define the edge lengths of a 5-dimensional box with volume $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}} = \prod f_i$. $\widetilde{S} = \left(\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}\right)^{1/5}$ is the edge of the 5-dimensional *hypercube* with equal volume. A perfect agent ($f_i = 1$ for all i) occupies the unit hypercube; any failure shrinks the volume and hence the edge length $\widetilde{S}$. The geometric mean thus finds the uniform truth value whose hypercube has the same volume as the original box of heterogeneous truth values.

Non-compensatory "AND" semantics. The geometric mean penalizes imbalance at the *multiplication* step, not the root. A strong factor cannot rescue a weak one: $0.9 \times 0.1 = 0.09$, far below the balanced $0.5 \times 0.5 = 0.25$, whereas the arithmetic mean treats both identically. Crucially, a zero in any factor—a verification condition that fails entirely—collapses the joint smooth AND $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}$ to zero and hence $\widetilde{S}$ to zero, regardless of scores elsewhere. This enforces the verification semantics we want: every conjunct must hold simultaneously, and a single hard failure is disqualifying.

Why not the arithmetic mean? The arithmetic mean $\frac{1}{5}\sum_{i=1}^{5} f_i$ is fully compensatory: a perfect soft truth on one conjunct offsets a zero on another, yielding a positive average even when a critical verification condition has entirely failed. For example, an agent that never produces a compiling theorem ($f_{\text{proof}} = 0$) but excels at unit testing could score $\bar{f} = 0.67$—falsely reporting that the agent is mostly correct. This violates the fundamental semantics of formal verification, where correctness is a *conjunction*: every condition must hold simultaneously, and a single failure is disqualifying. The arithmetic mean measures "what is the average soft truth per condition," but ignores whether any condition failed outright. In this sense it has *disjunctive* flavor: just as a smooth OR is satisfied when *at least one* conjunct is true, the arithmetic mean is pulled upward by any strong factor—precisely the wrong semantics for verification, where we need every condition to hold simultaneously, not just some. $\widetilde{S}$ instead measures the average conjunct truth consistent with the joint smooth AND—a meaningfully different and verification-appropriate quantity. Similarly, simple summation of raw scores rewards breadth over depth and grows with the number of factors, losing the $[0, 1]$ interpretability that makes scores comparable across benchmarks.

Three sub-blocks, five primitive checks. VERIBENCH organizes the five soft truth values into three conceptual sub-blocks: $\widetilde{\text{IC}}$ (IC_1, IC_2), $\widetilde{\text{TC}}$ (TC_1), and $\widetilde{\mathcal{D}}$ (D_1, D_2). The $1/5$ exponent weights every primitive verification condition equally, rather than normalizing by sub-block—which would artificially upweight any sub-block decomposed more finely. An agent scoring 0.8 on all five conditions receives $\widetilde{S} = 0.8$, directly interpretable as the average degree to which each verification conjunct holds.

Why not the harmonic mean? The harmonic mean $H = n/\sum_{i=1}^{n}(1/f_i)$ natively averages rates that share a common output burden, which is why it is the right summary for F_1 (precision and recall sharing TP) but not for our setting: our five factors are independent soft truth values, not rates sharing a numerator. The harmonic mean is also undefined at $f_i = 0$ and has no structural connection to the multiplicative product $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}$. The geometric mean uniquely normalizes the strict intersection ($\prod f_i$) required by AND semantics and collapses cleanly to zero on any hard failure.

E. Comprehensive SCSC Results

Table 4 reports the full SCSC evaluation across all 20 agents with completed SCSC summaries, plus the gold oracle. The top block contains *frontier* agents evaluated with 2026-era models (Sonnet 4.6, Opus 4.6, GPT-5.4); the main agents from Section 5 are reproduced here across the frontier and specialized blocks. The middle block contains agents from the initial evaluation round using Claude 3.5 Sonnet across a range of agent scaffolds (DSPy, OpenCode, OpenHands, Claude Code, Aider, AlphaApollo, and the Trace family). The bottom block contains specialized open-source and hybrid agents—Lean-specialised models (Leanstral v1/v2, Vibe Leanstral), Devstral, and prover-LLM hybrids (DeepSeek-Prover+Sonnet, Goedel+Sonnet). All $\widetilde{S}$ entries are populated from the v3 LLM-judge TC_1 scores; rows with $\widetilde{S} = 0.000$ reflect a true zero-collapse (some factor exactly 0), not a missing computation.

Headline observations.

- **Frontier ceiling.** Codex (GPT-5.4) leads at $\widetilde{S} = 0.417$, Claude Code (Sonnet 4.6) is second at 0.358, and the single-shot Sonnet 4.6 baseline reaches 0.344—only 0.073 behind the strongest agent. The remaining completed frontier scaffolds, Trace++ (GPT-5.4/Opus 4.6), fall in 0.209–0.250, well below the oracle ceiling 0.783.
- TC_1 **is the universal bottleneck.** No agent exceeds $\mathrm{TC}_1 = 0.105$. Compile-clean agents with $\mathrm{IC}_1 = 1.0$ (Codex, Claude Code) still score $\mathrm{TC}_1 \leq 0.102$, indicating that the LLM judge views agent-generated theorems as failing to cover the gold specifications regardless of whether the file compiles.
- **Self-correction remains inconsistent.** Among Sonnet 3.5 agents, DSPy ReAct reaches 0.354 on its completed 58-task subset, above the full-run Sonnet 3.5 baseline at 0.280, but Claude Code (0.241) and the Trace self-correction variants (0.122–0.162) do not improve over that baseline. Current feedback loops therefore help in some scaffolds but do not reliably translate into higher SCSC.
- **Lean-specialised models do not dominate.** Leanstral v1/v2 and Vibe Leanstral all sit in 0.207–0.225, comparable to mid-tier general agents but well below Sonnet 4.6 frontier scaffolds. Pure prover-LLM hybrids (DeepSeek-Prover, Goedel) fail to produce internally consistent Lean files ($\mathrm{IC}_1 = 0$, $\widetilde{S} = 0$), confirming that strong proof generation alone is insufficient—an agent must produce a complete Lean artifact (implementation + tests + theorems) that compiles end-to-end.

E.1. Precise definitions of the mechanical factors

The mechanical factors $\mathrm{IC}_1, \mathrm{IC}_2, D_1, D_2$ are computed from a parsed view of the Lean source plus a single `lake build`, with no LLM in the loop. We pin their definitions here so that future replications match the reference implementation exactly.

IC_2 **— agent theorems prove agent code.** Let $\mathrm{Thms}(\hat{L})$ denote the multiset of `theorem` declarations parsed from the agent's Lean source $\hat{L}$. A theorem is *closed* when its declaration block contains no occurrence of `sorry` or `admit` (matched as whole tokens; `admit` is scored identically to `sorry`). Then

$$\mathrm{IC}_2(\hat{L}) \;=\; \begin{cases} \dfrac{\#\{\text{closed theorems in } \hat{L}\}}{|\mathrm{Thms}(\hat{L})|}, & \text{if } \hat{L} \text{ compiles under } \texttt{lake build} \text{ and } |\mathrm{Thms}(\hat{L})| > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Compilation failure forces $\mathrm{IC}_2 = 0$ even when no `sorry` is present, because the kernel never certified those declarations. The empty-theorem case also collapses to 0, to disincentivise raising IC_2 by deleting proof obligations rather than discharging them. A single `sorry`-bearing theorem in a file with k theorems therefore reduces IC_2 by $1/k$, which translates into an SCSC reduction of approximately $\big(1 - 1/k\big)^{1/5}$ on the geometric mean.

IC_1 **— agent tests pass on agent code.** Let $T(\hat{L})$ denote the agent's `example` declarations (the test suite encoded as Lean kernel checks), and let $e_{\hat{L}}$ be the number of compile-time errors emitted by `lake build` against $T(\hat{L})$. Then $\mathrm{IC}_1(\hat{L}) = \max\!\big(0,\ (|T(\hat{L})| - e_{\hat{L}})/|T(\hat{L})|\big)$ when $|T(\hat{L})| > 0$, and $\mathrm{IC}_1 = 0$ when $|T(\hat{L})| = 0$.

D_1, D_2 **— gold-side validity gates.** D_1 and D_2 apply IC_1 and IC_2 to the gold artifact L^* instead of the agent artifact $\hat{L}$, using the same parser and build pipeline. A gold task with $D_2 < 1$ has `sorry`s in its own theorem set; the metric reports such tasks transparently rather than excluding them, and the $\widetilde{S} = 0.783$ oracle ceiling in Table 4 reflects the current $D_2 = 0.604$ gold gap.

Detector and edge behaviour. The detector marks a theorem block as containing a placeholder whenever the regular expression `\b(sorry|admit)\b` matches anywhere in the block, including inside tactic scripts, comments, and string literals. This is a deliberate over-approximation: it can over-count placeholders (the literal token `sorry` inside an explanatory comment will mark the block as not-closed) in exchange for closing the loophole of hiding placeholders inside macros, antiquotations, or string-embedded `sorry`s. Source: `veribench_metric/factors/c2_agent_theorems.py` and `veribench_metric/lean_utils.py` in the artifact release.

F. Human–Judge Alignment (Zero-Shot vs. Leakage-Safe Remapping)

This appendix separates four quantities that are easy to conflate: (i) zero-shot judge–human alignment, (ii) post-hoc supervised remapping, (iii) repeat-based judge stability, and (iv) uncertainty of correlation estimates (SE/CI).

Evaluation protocol (leakage-safe for calibration). Each example is a (`gold_file`, `candidate_file`) pair with a human label. For calibrated runs, we split by `gold_file` (task-level split, not row-level), fit remapping on train tasks only, and evaluate on disjoint held-out tasks. All candidates from one task stay in the same split. This is the basis for the "leakage-safe" claim in this section.

What the isotonic step is doing. The isotonic step is a post-hoc *monotone calibration* from raw judge scores to the human 0–5 scale. Concretely, on the training tasks we fit a non-decreasing function g such that a raw judge score s is mapped to a calibrated prediction $g(s)$ that better matches the average human label. Because g is constrained to be monotone, higher raw judge scores cannot be mapped below lower raw judge scores, but the spacing between score levels can change. This matters because the raw judge often has the right coarse ordering but uses the 0–5 scale differently from humans, for example by compressing the mid-range or overusing a particular category. Isotonic remapping corrects that scale mismatch; it does not add task information at test time, alter the underlying candidate files, or turn the judge into a theorem prover. For this reason, isotonic calibration tends to improve Pearson correlation more than rank correlations: it fixes numeric calibration on the human scale, not necessarily the ranking of close examples.

Judge/model/prompt setup. Zero-shot and calibration analyses use `gpt-5.3-codex` judge outputs on the same human-labeled pair set. Prompt families include `P4_CHECKLIST` and the `Human Rating Prompt`. No new agent model is introduced here: items are fixed candidate Lean files from benchmark artifacts.

Run protocol and statistics. The evaluation unit is `gold_file::candidate_file`. The evaluated candidate files are fixed benchmark artifacts generated by multiple LLM families (including Claude Sonnet/Opus and GPT-5 variants); each gold file has 3 candidates. For each (judge, item), we run $k = 3$ repeated judge calls and store `repeat_scores`, `score_mean`, and `score_std`. The deployed item score is the repeat mean:

$$\hat{s}_i = \frac{1}{k} \sum_{r=1}^{k} s_{i,r}.$$

where $s_{i,r}$ represents the score for task i under call r. Correlations are computed between human labels $(h_i)_i$ and deployed judge scores $(\hat{s}_i)_i$. Repeat-based stability is defined on the full $N \times K$ repeat matrix via per-item repeat SD:

$$\sigma_i = \mathrm{sd}(s_{i,\cdot}), \quad S_{\text{mean}} = \frac{1}{N} \sum_i \sigma_i, \quad S_{\text{median}} = \mathrm{median}_i(\sigma_i),$$

where lower values indicate greater repeat stability.

Interpretation of alignment results. Zero-shot judge scores show moderate alignment with human labels. Isotonic remapping improves Pearson substantially (up to $r = 0.7033$ in held-out evaluation), indicating better scale matching to the

human 0–5 labels rather than a fundamentally different judge. The negative result is preserved: rank correlations do not uniformly improve under calibration (e.g., CV2 isotonic $\rho = 0.3211$, $\tau = 0.2651$). Therefore, the gains are mostly realized as improved numeric calibration on the human scale rather than stronger rank-order fidelity.

Practical significance (effect size). Correlation tells us association, but not how much prediction error changes in practice. To measure practical impact, we compare per-item absolute error before and after applying the method. For each item:

$$\Delta\mathrm{AE} = |\hat{y}_{\text{baseline}} - y| - |\hat{y}_{\text{method}} - y|.$$

If $\Delta\mathrm{AE} > 0$, the method is closer to the human score on that item. We summarize these paired per-item improvements with Cohen's d_z:

$$d_z = \frac{\mathbb{E}[\Delta\mathrm{AE}]}{\mathrm{SD}(\Delta\mathrm{AE})}.$$

Larger positive d_z means larger and more consistent error reduction. Here, **best-single** denotes the strongest single judge prompt variant evaluated without stacking, selected under the same held-out protocol. Here, **ridge-only** denotes a linear stacked predictor (ridge regression) over judge features without isotonic post-hoc remapping. Using conventional reference points (about 0.2/0.5/0.8 for small/medium/large):

- **Avg split:** isotonic vs best-single $d_z = 3.264$ (very large), isotonic vs ridge-only $d_z = 0.223$ (small).
- **Pass1:** isotonic vs best-single $d_z = 2.816$ (very large), isotonic vs ridge-only $d_z = 0.375$ (small-to-medium).
- **Pass2:** isotonic vs best-single $d_z = 2.226$ (very large), isotonic vs ridge-only $d_z = 0.423$ (small-to-medium).

Interpretation: isotonic remapping provides a large practical gain over the best single-prompt baseline, but only a modest incremental gain over ridge-only stacking.

Original lenient judge (P4_CHECKLIST).

```
Evaluate this in order:
1) What properties does REFERENCE IMPLEMENTATION guarantee?
2) Which are preserved by GENERATED THEOREM?
3) Are there missing core guarantees or trivialization?
4) Map to score in [0.0, 1.0]:
1.0 equivalent; 0.7-0.9 minor gaps; 0.4-0.6 meaningful gaps; 0.1-0.3 weak; 0.0
trivial/incorrect.

REFERENCE IMPLEMENTATION:
{reference_impl}

GENERATED THEOREM:
{generated_theorem}

Return JSON only:
{"score": <float 0.0-1.0>, "reasoning": "<one sentence>"}
```

Human rating prompt (used for human collection and LLM adaptation).

```
Rate how well agent Lean theorem statements cover gold Lean theorem statements.
Use Python source + gold Lean file + agent Lean file.
Scope: statement coverage only (not proof style/quality).
Coverage allows same theorem, logical equivalence, stronger implying theorem, or a
small lemma set that recovers the gold theorem.
Checklist: A key theorems covered? B pre/postconditions preserved or
strengthened? C theorem spam? D if spam, handful or dominates?
Base score 0--5: 5 all covered, 4 mostly covered, 3 partial with important
gaps, 2 weak (< half), 1 none covered but relevant non-vacuous theorem, 0
vacuous/broken.
Spam adjustment: C=No no change; C=Yes and D=handful lower up to 1; C=Yes and
D=dominates lower up to 2.
```

Future work. A recurring annotation issue is that absolute scalar scoring can be noisier than within-task comparison. A direct extension is ranking-first or joint three-candidate grading per task, followed by task-level aggregation before judge–human correlation.

G. Human Judgement Collection

This appendix documents how human labels were collected, grouped, and aggregated for the analyses in Appendix F. All scores use the discrete 0–5 scale.

Protocol summary. Five independent human raters scored theorem-coverage items on a 0–5 ordinal scale. For each item, raters saw the Python source, the gold Lean file, and the candidate Lean file, and were instructed to judge statement coverage rather than proof style. Raters worked independently, were instructed not to use LLMs or automated judges during labeling, and did not discuss or reconcile scores with one another while labeling. We therefore treat each submitted score as an independent expert judgment rather than a negotiated consensus label. Scores are aggregated either at the item level or task level as described below; the main calibration claim uses the held-out task-split Pearson result in Appendix F.

G.1. VB Rating Protocol

Rater groups used in analysis. For analysis only, we use two grouped label sets:

- **Lenient set**: old-prompt participant1 passes plus participant4 and participant5.
- **Expert/harsh set**: participant1 (correct prompt), participant2, and participant3.

These names are descriptive shortcuts for observed score distributions, not claims about rater quality.

Included files and counts. Lenient set files:

- `old_prompt/participant1_data_pass_1.tsv`
- `old_prompt/participant1_data_pass_2.tsv`
- `participant4_data_pass_1.tsv`
- `participant5_data_pass_1.tsv`

Expert/harsh set files:

- `participant1_data_correct_prompt.tsv`
- `participant2_data_pass_1.tsv`
- `participant3_data_pass_1.tsv`

Filtered merged counts (valid `human_score_0_5` rows): $n = 297$ for lenient and $n = 193$ for expert/harsh.

Contributor protocol (for future raters). To extend the dataset consistently:

1. Use the TC rubric and score each item on $\{0, 1, 2, 3, 4, 5\}$.
2. Review all three inputs for each item: Python source, gold Lean file, candidate Lean file.
3. Optionally record checklist notes (A/B/C/D), but always provide a score.
4. Save a TSV with columns `gold_file`, `candidate_file`, `human_score_0_5`.

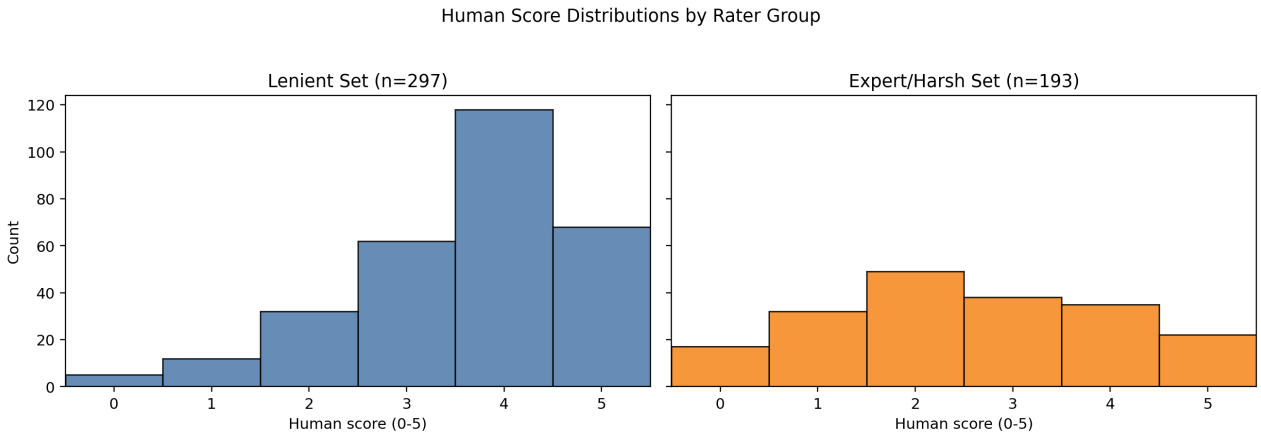

Figure 4. Human score distributions (0–5): lenient set ($n = 297$, left) and expert/harsh set ($n = 193$, right).

Collection constraints. Raters are instructed to work independently, not use LLMs or automated judges during labeling, and not coordinate with one another to resolve disagreements. These constraints are intended to preserve human-only annotation variability. Even with a shared rubric, disagreement is expected because theorem-level quality judgments are difficult and raters may apply the scale differently. The *calibration* discussed elsewhere in the paper refers to post-hoc calibration of the LLM judge to these fixed human labels; it does not mean the human raters were calibrated by discussing and harmonizing their answers.

G.2. Distribution of Human Scores

Distribution differences and relevance. Figure 4 shows that the two grouped sets have different score distributions. This matters because judge–human correlation depends on the target human distribution: changing prompt/rater regime can change both marginal score spread and calibration behavior. For this reason, we report strict-subset and aggregated analyses separately rather than pooling all settings into one headline number.

Cross-set disagreement (MSE/MAE). Table 8 reports simple error-style disagreement metrics between the two grouped sets.

Within-group score dispersion. To quantify how much raters vary on the same item within each grouped label set, we compute per-item score SD and IQR across available raters, then summarize these per-item dispersion values. Table 9 reports mean and median dispersion.

Inter-annotator agreement (human–human). Beyond MAE/MSE, we report pairwise human–human agreement over overlapping item keys using Pearson, Spearman, Kendall τ, and weighted Cohen's κ (linear and quadratic weights). Table 10 summarizes pairwise agreement across all rater pairs in the current collection.

Intraclass correlation (ICC). Because the final human target used for calibration is an average across raters rather than a single annotator, we also report intraclass correlation with absolute agreement. We use ICC(2,1) for single-rater reliability and ICC(2,k) for the reliability of the mean across k raters. Standard ICC requires a complete item$\times$rater matrix, so we compute it on complete-overlap subsets. Table 11 reports the main five-rater overlap and an expert/harsh sensitivity subset.

Interpretation of agreement statistics. The agreement pattern is consistent with a difficult semantic-labeling task: single-rater labels are noisy, but averaging across raters is materially more stable. We therefore interpret judge–human calibration as alignment to a noisy expert signal rather than a noise-free ground truth, and we prefer averaged human labels over any single annotator when reporting judge calibration.

Strict-set alignment summary. Table 12 reports strict-subset Pearson correlations for the expert/harsh subset. These correlations are higher than the main held-out task-split result and should be read as a subset analysis, not as the headline

calibration claim. Table 13 reports the corresponding aggregation analyses over strict and lenient label sources.

Aggregation experiments and what is averaged. Table 13 compares two aggregation schemes:

- **All prompts/users averaged**: average scores over available rater/prompt sources per item key before correlation.
- **Task-level averaging**: first aggregate candidate-level scores within each task, then correlate at task level.

Why the aggregated numbers are higher. These ~ 0.85 values come from task-level aggregation (averaging within task before correlating), which removes within-task rater noise and typically inflates correlation by construction. They do not contradict the item-level held-out task-split result of $r = 0.7033$ reported as the headline calibration: the two are different evaluation granularities and should not be compared as identical quantities. We report both so readers can see the full picture (item-level honesty + aggregated diagnostic) rather than only the more flattering aggregated number.

Interpretation of aggregation results. The two aggregation schemes yield very similar correlations (Table 13). This indicates that the observed alignment signal is not highly sensitive to whether averaging is performed across rater/prompt sources at the item level or at the task level.

Future work: pairwise-vs-absolute labeling. Fine-grained absolute scoring can be difficult when candidates are close. Preference-learning literature commonly uses pairwise judgments for this reason (Rafailov et al., 2023). In this work, we keep scalar labels and use aggregation to reduce per-rater/per-prompt scale noise.

H. All Evaluation Results (Comprehensive Dump)

This appendix collects every completed SCSC evaluation run in the repository so the paper carries every result we have, not just the headline ones. The most recent SCSC evaluation (**v3**, used in Section 5 and Appendix E) is the primary source of truth; earlier rounds (v1, v2) are retained verbatim so that conclusions can be traced back through the metric's evolution. Provenance for every table is the absolute path under `experiments/` listed in each caption.

H.1. SCSC v3 — full per-agent, per-split detail (current)

H.2. SCSC v2 — superseded by v3 (kept for traceability)

H.3. SCSC v1 — earliest scoring round (column names use C_1–C_5)

H.4. Harbor agent runs (Experiment 35, 2026-03-18 summary)

Source: `experiments/35_vb_x_harbor/results_summary/results_summary_2026-03-18.md`.

```
# VeriBench E3 - Agent Comparison Results

**Date:** 2026-03-18
**Dataset:** veribench@1.0
**Tasks:** 170 Python -> Lean 4
**W&B:**
  wandb.ai/[redacted-for-double-blind]
    /veribench-e3-agents
**Report:** [redacted for double-blind review]

> Run instructions (read before every run):
> When saving results from a new run, always write
> to a NEW file:
>   results_summary_YYYY-MM-DD__HH-MM-SS.md
> using the current date+time at run start.
> Example: results_summary_2026-03-17__14-23-05.md
> Use: OUTFILE=$RESULTS/\
>   results_summary_$(date +%Y-%m-%d__%H-%M-%S).md
> Do NOT overwrite this file or existing summaries.
> After the run, update TODO rows in this file
> with compile % and job dir.

---

## Experimental Setting
```

```
### Harbor Sandboxing - All Runs Isolated

Every agent evaluation was conducted through Harbor
0.1.44 (github.com/av-io/harbor), a sandboxed eval
framework that enforces strict isolation between
agents and gold reference solutions.

#### Container isolation

Each task runs inside a fresh Docker container:
  ghcr.io/[anonymized-org]/veribench-sandbox:latest
The container is:

- Spawned from scratch per task; no state carries
  over between tasks or agents.
- Given only the task instruction.md (Python
  function to formalize) and a writable
  /home/user/lean_project/ workspace.
- Gold Lean 4 references are never mounted or
  visible inside the container.
- Destroyed after the task completes; no shared
  filesystem between concurrent task containers.

Runs were executed with --n-concurrent 5 (up to 5
parallel tasks), each in its own isolated
container.

#### Anti-cheat verification (per task)

Harbor verifier runs verifier/anticheat.log checks
after every task:

| Check                          | Status    |
|:-------------------------------|:----------|
| Gold ref absence in container | PASS      |
| Canary string detection       | PASS      |
| Filesystem boundary           | PASS      |
| Cross-agent isolation         | PASS      |
| External internet reachable   | REACHABLE |

#### task.toml (confirmed for all runs)

  [environment]
  type = "docker"
  docker_image =
    "ghcr.io/[anonymized-org]/veribench-sandbox"
  allow_internet = true

#### Why this matters

1. No gold-ref leakage: agents cannot read
   solution.lean during the run.
2. No cross-agent contamination.
3. Reproducibility: same Docker image for all.
4. Limitation: internet access (allow_internet =
   true). Hardening planned.

---

## TL;DR

> Current SOTA (compile %):
>   Trace++ / gpt-5.4     = 93.5%
>   claude-code           = 83.5%
>   dspy-react / sonnet-4-6 = 77.6%
>   Trace++ / sonnet-4-6  = 41.8%
>   AlphaApollo           = 36.5%
>   baseline              = 32.9%
>   aider/sonnet          = 24.1%
>   Trace+                = 13.5%
> mini-swe-agent / sonnet-4-6 = 100% (SUSPICIOUS;
>   needs manual verification; was 0% w/ gpt-4o).
> All rows report 3 metrics: Compile %, Edit
> Distance, Judge Score (0-5). Missing = TODO.
>
> - Agent + Judge models: claude-sonnet-4-6.
> - Compile-check feedback is the key
>   differentiator; agents with a lake env lean
>   error loop dominate.
> - Proof-search models (Goedel, DeepSeek) are
```

```
>   local fixed-weight models.
>
> Naming convention:
> - Trace+  = compile-feedback loop only
>             (trace_selfdebug_agent.py).
> - Trace++ = compile-feedback + LLM quality
>             judge loop
>             (trace_selfimprove_agent.py).
> - Older trace_agent.py / trace_plus_agent.py runs
>   on a separate compute cluster - Appendix D.
```

H.5. Proof-failure audit (Experiment 47)

Table 17. Proof-failure categories from Experiment 47 (audit 2026-05-04).

file	line	kind	name	category	reason
cs_set/bfs.lean	89	theorem	soundness_thm	OTHER	no marker; tractable
cs_set/bfs.lean	95	theorem	optimality_thm	OTHER	no marker; tractable
cs_set/bfs.lean	157	theorem	bfs_equivalence_thm	OTHER	no marker; tractable
cs_set/dijkstra.lean	94	theorem	soundness_thm	OTHER	no marker; tractable
cs_set/dijkstra.lean	100	theorem	optimality_thm	OTHER	no marker; tractable
cs_set/dijkstra.lean	178	theorem	dijkstra_equivalence_thm	OTHER	no marker; tractable
cs_set/edit_distance.lean	20	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
cs_set/edit_distance.lean	172	example	¡anon¿	OTHER	no marker; tractable
cs_set/heap_sort.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
cs_set/heap_sort.lean	164	example	¡anon¿	OTHER	no marker; tractable
cs_set/heap_sort.lean	170	example	¡anon¿	OTHER	no marker; tractable
cs_set/heap_sort.lean	176	example	¡anon¿	OTHER	no marker; tractable
cs_set/heap_sort.lean	182	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_algorithm.lean	23	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
cs_set/heapsort_algorithm.lean	320	theorem	extractLoop_size	OTHER	no marker; tractable
cs_set/heapsort_algorithm.lean	327	theorem	extractLoop_size	OTHER	no marker; tractable
cs_set/heapsort_algorithm.lean	345	theorem	extractLoop_size	OTHER	no marker; tractable
cs_set/heapsort_algorithm.lean	501	theorem	heapsort_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_build_heap.lean	284	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_build_heap.lean	293	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_build_heap.lean	304	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_build_heap.lean	358	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_build_heap.lean	473	theorem	buildMaxHeap_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_heap_structure.lean	384	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_heap_structure.lean	420	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_heap_structure.lean	431	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_heap_structure.lean	471	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_max_heapify.lean	309	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_max_heapify.lean	631	theorem	maxHeapify_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_max_heapify.lean	636	theorem	minHeapify_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	348	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	360	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	371	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	382	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	393	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	412	example	¡anon¿	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	592	theorem	heapExtractMax_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	598	theorem	heapIncreaseKey_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_priority_queue.lean	604	theorem	maxHeapInsert_equivalence_thm	OTHER	no marker; tractable
cs_set/heapsort_problems.lean	22	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
cs_set/heapsort_problems.lean	570	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): siftUp
cs_set/heapsort_problems.lean	588	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_problems.lean	597	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify
cs_set/heapsort_problems.lean	621	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify, dAryMaxHeapify
cs_set/heapsort_problems.lean	646	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify, dAryMaxHeapify
cs_set/heapsort_problems.lean	658	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify, dAryMaxHeapify
cs_set/heapsort_problems.lean	668	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): maxHeapify, dAryMaxHeapify
cs_set/heapsort_problems.lean	680	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): youngifyDown, maxHeapify, dAryMaxHeapify
cs_set/heapsort_problems.lean	691	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): youngifyDown, maxHeapify, dAryMaxHeapify

file	line	kind	name	category	reason
cs_set/heapsort_problems.lean	703	theorem	build_methods_can_differ_thm	PARTIAL_DEPENDENT	references partial def(s): youngifyDown, maxHeapify, dAryMaxHeapify
cs_set/lcs.lean	20	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/10_MyEvenSumParity.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/11_MyFirstChar.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/12_MyStringLength.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/13_MyUppercase.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/13_MyUppercase.lean	105	example	¡anon¿	OTHER	no marker; tractable
easy_set/13_MyUppercase.lean	118	example	¡anon¿	OTHER	no marker; tractable
easy_set/13_MyUppercase.lean	124	example	¡anon¿	OTHER	no marker; tractable
easy_set/13_MyUppercase.lean	130	example	¡anon¿	OTHER	no marker; tractable
easy_set/13_MyUppercase.lean	149	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/13_MyUppercase.lean	212	theorem	myUppercase_equivalence_thm	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/14_MyRemoveSpaces.lean	97	theorem	remove_spaces_no_spaces	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	101	theorem	remove_spaces_idempotent	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	105	theorem	remove_spaces_no_space_resul	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	109	theorem	remove_spaces_length_le	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	113	theorem	remove_spaces_append	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	117	theorem	remove_spaces_preserves_non_spaces	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	134	theorem	no_spaces_thm	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	140	theorem	length_bound_thm	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	146	theorem	char_preservation_thm	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	163	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/14_MyRemoveSpaces.lean	216	theorem	myRemoveSpaces_equivalence	OTHER	no marker; tractable
easy_set/15_MyRepeatString.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/15_MyRepeatString.lean	185	theorem	myRepeatString_equivalence_thm	OTHER	no marker; tractable
easy_set/16_MyFactorial.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/16_MyFactorial.lean	219	theorem	myFactorial_equivalence_thm	OTHER	no marker; tractable
easy_set/18_MyFibonacci.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/18_MyFibonacci.lean	131	example	¡anon¿	OTHER	no marker; tractable
easy_set/18_MyFibonacci.lean	207	theorem	myFibonacci_equivalence_thm	OTHER	no marker; tractable
easy_set/19_MyPower.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/19_MyPower.lean	211	theorem	myPower_equivalence_thm	OTHER	no marker; tractable
easy_set/1_MyAdd.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/1_MyAdd.lean	195	theorem	myAdd_equivalence	OTHER	no marker; tractable
easy_set/20_MySumDigits.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/20_MySumDigits.lean	136	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): mySumDigits
easy_set/20_MySumDigits.lean	142	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): mySumDigits
easy_set/20_MySumDigits.lean	161	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/20_MySumDigits.lean	216	theorem	mySumDigits_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): mySumDigits
easy_set/21_is_palindrome.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/21_is_palindrome.lean	161	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): isPalindrome, firstDigit
easy_set/21_is_palindrome.lean	238	theorem	isPalindrome_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): isPalindrome
easy_set/23_is_prime.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/23_is_prime.lean	137	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): isPrime
easy_set/23_is_prime.lean	150	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): isPrime
easy_set/23_is_prime.lean	168	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/23_is_prime.lean	229	theorem	isPrime_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): isPrime
easy_set/24_matrix_multiply.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/24_matrix_multiply.lean	97	example	¡anon¿	OTHER	no marker; tractable
easy_set/24_matrix_multiply.lean	105	example	¡anon¿	OTHER	no marker; tractable
easy_set/24_matrix_multiply.lean	109	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): matrixPower
easy_set/24_matrix_multiply.lean	158	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): matrixPower
easy_set/24_matrix_multiply.lean	223	example	¡anon¿	OTHER	no marker; tractable
easy_set/24_matrix_multiply.lean	243	example	¡anon¿	OTHER	no marker; tractable
easy_set/24_matrix_multiply.lean	252	theorem	fastFibonacci_equivalence_thm	OTHER	no marker; tractable
easy_set/25_longest_palindromic_substring.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/25_longest_palindromic_substring.lean	127	example	¡anon¿	OTHER	no marker; tractable
easy_set/25_longest_palindromic_substring.lean	133	example	¡anon¿	OTHER	no marker; tractable
easy_set/25_longest_palindromic_substring.lean	139	example	¡anon¿	OTHER	no marker; tractable

file	line	kind	name	category	reason
easy_set/25_longest_palindromic_substring.lean	145	example	¡anon¿	OTHER	no marker; tractable
easy_set/25_longest_palindromic_substring.lean	233	theorem	longestPalindrome_equivalence_thm	OTHER	no marker; tractable
easy_set/26_longest_common_subsequence.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/26_longest_common_subsequence.lean	146	example	¡anon¿	OTHER	no marker; tractable
easy_set/26_longest_common_subsequence.lean	152	example	¡anon¿	OTHER	no marker; tractable
easy_set/26_longest_common_subsequence.lean	158	example	¡anon¿	OTHER	no marker; tractable
easy_set/26_longest_common_subsequence.lean	164	example	¡anon¿	OTHER	no marker; tractable
easy_set/26_longest_common_subsequence.lean	254	theorem	longestCommonSubsequence_equivalence_thm	OTHER	no marker; tractable
easy_set/28_length_of_longest_substring.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/28_length_of_longest_substring.lean	133	example	¡anon¿	OTHER	no marker; tractable
easy_set/28_length_of_longest_substring.lean	139	example	¡anon¿	OTHER	no marker; tractable
easy_set/28_length_of_longest_substring.lean	152	example	¡anon¿	OTHER	no marker; tractable
easy_set/28_length_of_longest_substring.lean	243	theorem	lengthOfLongestSubstring_equivalence_thm	OTHER	no marker; tractable
easy_set/29_longest_common_prefix.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/29_longest_common_prefix.lean	145	example	¡anon¿	OTHER	no marker; tractable
easy_set/29_longest_common_prefix.lean	152	example	¡anon¿	OTHER	no marker; tractable
easy_set/29_longest_common_prefix.lean	158	example	¡anon¿	OTHER	no marker; tractable
easy_set/29_longest_common_prefix.lean	177	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/29_longest_common_prefix.lean	245	theorem	longestCommonPrefix_equivalence_thm	OTHER	no marker; tractable
easy_set/2_MyMax.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/30_longest_valid_parentheses.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/30_longest_valid_parentheses.lean	139	example	¡anon¿	OTHER	no marker; tractable
easy_set/30_longest_valid_parentheses.lean	145	example	¡anon¿	OTHER	no marker; tractable
easy_set/30_longest_valid_parentheses.lean	151	example	¡anon¿	OTHER	no marker; tractable
easy_set/30_longest_valid_parentheses.lean	262	theorem	longestValidParentheses_equivalence_thm	OTHER	no marker; tractable
easy_set/31_count_even_digits.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/31_count_even_digits.lean	221	theorem	countEvenDigits_equivalence_thm	OTHER	no marker; tractable
easy_set/32_divisible_by_seven_and_eight_count.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/33_MyMinimum.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/34_perfect_square_or_add_five.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/34_perfect_square_or_add_five.lean	111	example	¡anon¿	OTHER	no marker; tractable
easy_set/34_perfect_square_or_add_five.lean	118	example	¡anon¿	OTHER	no marker; tractable
easy_set/34_perfect_square_or_add_five.lean	162	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/34_perfect_square_or_add_five.lean	216	theorem	perfectSquareOrAddFive_equivalence_thm	OTHER	no marker; tractable
easy_set/35_quadruple_abs_difference_plus_three.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/36_remove_digits.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/37_remove_first_five_letters.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/38_reverse_words.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/38_reverse_words.lean	112	example	¡anon¿	OTHER	no marker; tractable
easy_set/38_reverse_words.lean	119	example	¡anon¿	OTHER	no marker; tractable
easy_set/38_reverse_words.lean	126	example	¡anon¿	OTHER	no marker; tractable
easy_set/38_reverse_words.lean	136	example	¡anon¿	OTHER	no marker; tractable
easy_set/38_reverse_words.lean	155	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/38_reverse_words.lean	206	theorem	reverseWords_equivalence_thm	OTHER	no marker; tractable
easy_set/39_triple_all.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/39_triple_all.lean	196	example	¡anon¿	OTHER	no marker; tractable
easy_set/3_MyProduct.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/40_all_even.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/41_duplicate_chars.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/42_product_of_list.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/43_replace_spaces.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/43_replace_spaces.lean	108	example	¡anon¿	OTHER	no marker; tractable
easy_set/43_replace_spaces.lean	115	example	¡anon¿	OTHER	no marker; tractable
easy_set/43_replace_spaces.lean	122	example	¡anon¿	OTHER	no marker; tractable
easy_set/43_replace_spaces.lean	129	example	¡anon¿	OTHER	no marker; tractable
easy_set/43_replace_spaces.lean	148	theorem	correctness_thm	OTHER	no marker; tractable
easy_set/43_replace_spaces.lean	201	theorem	replaceSpaces_equivalence_thm	OTHER	no marker; tractable
easy_set/44_square_or_double.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found

file	line	kind	name	category	reason
easy_set/4_MyRemainder.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/4_MyRemainder.lean	199	theorem	myRemainder_equivalence_thm	OTHER	no marker; tractable
easy_set/5_MySquare.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/5_MySquare.lean	167	theorem	mySquare_equivalence	OTHER	no marker; tractable
easy_set/6_myMaxList.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/7_myEvanList.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/8_myReverse.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
easy_set/9_MyOddSumParity.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humanevalXL_9__isPrime.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humanevalXL_9__isPrime.lean	199	theorem	equivalence_thm	OTHER	no marker; tractable
humaneval_set/humanevalXL_9__isPrime.lean	218	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humanevalXL_9__isPrime.lean	226	theorem	isPrime_even_gt_two	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	230	theorem	isPrime_composite	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	234	theorem	isPrime_has_divisor	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	238	theorem	isPrime_no_divisors	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	246	theorem	isPrime_factor_monotone	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	250	theorem	isPrime_only_even_prime	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	254	theorem	isPrime_sqrt_suffices	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	258	theorem	isPrime_divisor_symmetry	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	269	theorem	isPrime_no_nontrivial_divisors	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humanevalXL_9__isPrime.lean	346	theorem	isPrime_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): isPrime
humaneval_set/humaneval_0_hasCloseElements.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_0_hasCloseElements.lean	125	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_0_hasCloseElements.lean	132	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_0_hasCloseElements.lean	139	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_0_hasCloseElements.lean	146	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_0_hasCloseElements.lean	166	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_0_hasCloseElements.lean	222	theorem	hasCloseElements_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_10_is_palindrome.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_10_is_palindrome.lean	138	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_10_is_palindrome.lean	152	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_10_is_palindrome.lean	159	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_10_is_palindrome.lean	166	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_10_is_palindrome.lean	185	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_10_is_palindrome.lean	250	theorem	makePalindrome_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_11_string_xor.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_11_string_xor.lean	174	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_11_string_xor.lean	193	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_11_string_xor.lean	250	theorem	string_xor_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_12_longest.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_12_longest.lean	115	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_12_longest.lean	137	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_12_longest.lean	144	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_12_longest.lean	163	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_12_longest.lean	224	theorem	longest_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_13__gcd.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_13__gcd.lean	168	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_13__gcd.lean	187	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_13__gcd.lean	240	theorem	gcd_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_14_all_prefixes.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_14_all_prefixes.lean	116	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_14_all_prefixes.lean	124	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_14_all_prefixes.lean	131	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_14_all_prefixes.lean	150	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_14_all_prefixes.lean	203	theorem	allPrefixes_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_15_string_sequence.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_15_string_sequence.lean	111	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_15_string_sequence.lean	118	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_15_string_sequence.lean	125	example	¡anon¿	OTHER	no marker; tractable

file	line	kind	name	category	reason
humaneval_set/humaneval_15_string_sequence.lean	131	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_15_string_sequence.lean	150	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_15_string_sequence.lean	203	theorem	string_sequence_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_16_count_distinct_characters.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_16_count_distinct_characters.lean	131	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_16_count_distinct_characters.lean	148	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_16_count_distinct_characters.lean	220	theorem	countDistinctCharacters_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_17_parse_music.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_18_how_many_times.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_18_how_many_times.lean	145	example	¡anon¿	PARTIAL_DEPENDENT	references partial def(s): how_many_times
humaneval_set/humaneval_18_how_many_times.lean	239	theorem	how_many_times_equivalence_thm	PARTIAL_DEPENDENT	references partial def(s): how_many_times
humaneval_set/humaneval_19_sort_numbers.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_19_sort_numbers.lean	153	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_19_sort_numbers.lean	163	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_19_sort_numbers.lean	171	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_19_sort_numbers.lean	179	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_19_sort_numbers.lean	253	theorem	sortNumbers_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_1_separate_paren_groups.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_1_separate_paren_groups.lean	164	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_1_separate_paren_groups.lean	183	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_1_separate_paren_groups.lean	246	theorem	separateParenGroups_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_20_find_closest_elements.lean	71	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	75	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	83	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	87	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	95	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	99	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	103	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	107	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	128	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_20_find_closest_elements.lean	137	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_20_find_closest_elements.lean	146	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_20_find_closest_elements.lean	157	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_20_find_closest_elements.lean	175	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_20_find_closest_elements.lean	206	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	214	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	218	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	226	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	230	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_20_find_closest_elements.lean	235	theorem	findClosestElements_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_21_rescale_to_unit.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_21_rescale_to_unit.lean	68	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_21_rescale_to_unit.lean	127	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_21_rescale_to_unit.lean	134	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_21_rescale_to_unit.lean	141	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_21_rescale_to_unit.lean	160	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_21_rescale_to_unit.lean	225	theorem	rescale_to_unit_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_22_filter_integers.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_22_filter_integers.lean	138	theorem	filter_integers_eq	OTHER	no marker; tractable
humaneval_set/humaneval_23_strlen.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_24_largest_divisor.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_24_largest_divisor.lean	115	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_24_largest_divisor.lean	122	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_24_largest_divisor.lean	130	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_24_largest_divisor.lean	138	example	¡anon¿	OTHER	no marker; tractable

file	line	kind	name	category	reason
humaneval_set/humaneval_24_largest_divisor.lean	220	theorem	largestDivisor_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_25_factorize.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_25_factorize.lean	122	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_25_factorize.lean	130	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_25_factorize.lean	139	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_25_factorize.lean	154	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_25_factorize.lean	236	theorem	factorize_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_26_remove_duplicates.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_26_remove_duplicates.lean	118	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_26_remove_duplicates.lean	128	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_26_remove_duplicates.lean	136	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_26_remove_duplicates.lean	220	theorem	removeDuplicates_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_27_flip_case.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_27_flip_case.lean	116	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_27_flip_case.lean	123	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_27_flip_case.lean	131	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_27_flip_case.lean	141	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_27_flip_case.lean	214	theorem	flipCase_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_28_concatenate.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_28_concatenate.lean	156	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_29_filter_by_prefix.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_29_filter_by_prefix.lean	151	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_2_truncate_number.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_2_truncate_number.lean	102	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_2_truncate_number.lean	110	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_2_truncate_number.lean	124	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_2_truncate_number.lean	131	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_2_truncate_number.lean	150	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_30_get_positive.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_31_is_prime.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_31_is_prime.lean	149	theorem	composite_thm	OTHER	no marker; tractable
humaneval_set/humaneval_31_is_prime.lean	156	theorem	prime_def_thm	OTHER	no marker; tractable
humaneval_set/humaneval_31_is_prime.lean	174	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_31_is_prime.lean	212	theorem	isPrime_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_32_findZero.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_32_findZero.lean	127	theorem	bracket_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_findZero.lean	161	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_findZero.lean	195	theorem	findZero_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_poly.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_32_poly.lean	109	theorem	empty_poly_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_poly.lean	116	theorem	constant_poly_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_poly.lean	123	theorem	linear_poly_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_poly.lean	141	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_32_poly.lean	170	theorem	poly_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_33_sort_third.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_33_sort_third.lean	131	theorem	length_preservation_thm	OTHER	no marker; tractable
humaneval_set/humaneval_33_sort_third.lean	138	theorem	non_third_preservation_thm	OTHER	no marker; tractable
humaneval_set/humaneval_33_sort_third.lean	146	theorem	third_sorted_thm	OTHER	no marker; tractable
humaneval_set/humaneval_33_sort_third.lean	153	theorem	idempotent_thm	OTHER	no marker; tractable
humaneval_set/humaneval_33_sort_third.lean	171	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_33_sort_third.lean	212	theorem	sortThird_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_34_unique.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_34_unique.lean	99	theorem	no_duplicates_thm	OTHER	no marker; tractable
humaneval_set/humaneval_34_unique.lean	106	theorem	sorted_thm	OTHER	no marker; tractable

file	line	kind	name	category	reason
humaneval_set/humaneval_34_unique.lean	113	theorem	subset_thm	OTHER	no marker; tractable
humaneval_set/humaneval_34_unique.lean	120	theorem	completeness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_34_unique.lean	138	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_34_unique.lean	200	theorem	unique_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_35_max_element.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_36_fizz_buzz.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_36_fizz_buzz.lean	120	theorem	monotonicity_thm	OTHER	no marker; tractable
humaneval_set/humaneval_36_fizz_buzz.lean	155	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_36_fizz_buzz.lean	213	theorem	fizzBuzz_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_37_sort_even.lean	121	theorem	length_preservation_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	128	theorem	odd_preservation_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	136	theorem	even_sorting_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	143	theorem	idempotent_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	161	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_37_sort_even.lean	224	theorem	sortEven_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_38_encode_cyclic.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_38_encode_cyclic.lean	130	theorem	length_preservation_thm	OTHER	no marker; tractable
humaneval_set/humaneval_38_encode_cyclic.lean	137	theorem	decode_correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_38_encode_cyclic.lean	151	theorem	triple_encode_thm	OTHER	no marker; tractable
humaneval_set/humaneval_38_encode_cyclic.lean	169	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_38_encode_cyclic.lean	224	theorem	encode_cyclic_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	126	theorem	result_is_prime_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	133	theorem	result_is_fib_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	140	theorem	monotone_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	147	theorem	positivity_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	165	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_39_prime_fib.lean	215	theorem	primeFib_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_3_below_zero.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_3_below_zero.lean	255	theorem	belowZero_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_40_triples_sum_to_zero.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_40_triples_sum_to_zero.lean	134	theorem	specification_thm	OTHER	no marker; tractable
humaneval_set/humaneval_40_triples_sum_to_zero.lean	149	theorem	small_list_thm	OTHER	no marker; tractable
humaneval_set/humaneval_40_triples_sum_to_zero.lean	166	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_40_triples_sum_to_zero.lean	225	theorem	triplesSumToZero_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_41_car_race_collision.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_42_incr_list.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	127	theorem	specification_thm	OTHER	no marker; tractable
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	141	theorem	single_element_thm	OTHER	no marker; tractable
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	148	theorem	all_positive_thm	OTHER	no marker; tractable
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	165	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_43_pairs_sum_to_zero.lean	222	theorem	pairsSumToZero_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_44_change_base.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_44_change_base.lean	127	theorem	non_empty_thm	OTHER	no marker; tractable
humaneval_set/humaneval_44_change_base.lean	134	theorem	single_digit_thm	OTHER	no marker; tractable
humaneval_set/humaneval_44_change_base.lean	143	theorem	digits_valid_thm	OTHER	no marker; tractable
humaneval_set/humaneval_44_change_base.lean	160	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_44_change_base.lean	217	theorem	changeBase_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_45_triangle_area.lean	46	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	50	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	54	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	62	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	66	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	70	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	78	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	82	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	85	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	111	theorem	zero_base_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_45_triangle_area.lean	117	theorem	zero_height_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_45_triangle_area.lean	124	theorem	commutativity_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_45_triangle_area.lean	142	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)

file	line	kind	name	category	reason
humaneval_set/humaneval_45_triangle_area.lean	163	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	167	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	175	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	179	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	187	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_45_triangle_area.lean	191	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_46_fib4.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_46_fib4.lean	142	theorem	recurrence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_46_fib4.lean	167	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_46_fib4.lean	243	theorem	fib4_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_47_median.lean	69	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	73	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	81	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	85	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	89	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	93	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	101	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	105	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	108	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	125	theorem	empty_list_thm	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	131	theorem	singleton_thm	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	138	theorem	sort_length_thm	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	155	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	182	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	186	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	194	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	198	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	206	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	210	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_47_median.lean	214	theorem	median_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_48_is_palindrome.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_48_is_palindrome.lean	134	theorem	reverse_thm	OTHER	no marker; tractable
humaneval_set/humaneval_48_is_palindrome.lean	141	theorem	wrap_thm	OTHER	no marker; tractable
humaneval_set/humaneval_48_is_palindrome.lean	159	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_48_is_palindrome.lean	218	theorem	is_palindrome_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_49_modp.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_49_modp.lean	114	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_49_modp.lean	118	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_49_modp.lean	122	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_49_modp.lean	215	theorem	modular_equiv_thm	OTHER	no marker; tractable
humaneval_set/humaneval_49_modp.lean	242	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_4_mean_absolute_deviation.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_4_mean_absolute_deviation.lean	121	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_4_mean_absolute_deviation.lean	128	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_4_mean_absolute_deviation.lean	134	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_4_mean_absolute_deviation.lean	141	example	¡anon¿	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_4_mean_absolute_deviation.lean	160	theorem	correctness_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_4_mean_absolute_deviation.lean	215	theorem	meanAbsoluteDeviation_equivalence_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
humaneval_set/humaneval_50_encode_shift.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_50_encode_shift.lean	130	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_50_encode_shift.lean	160	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_50_encode_shift.lean	203	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_50_encode_shift.lean	273	theorem	encodeShift_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_50_encode_shift.lean	277	theorem	decodeShift_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_51_remove_vowels.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_52_below_threshold.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_52_below_threshold.lean	324	theorem	equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_5_intersperse.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_5_intersperse.lean	296	theorem	intersperse_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_6_parse_nested_parens.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_6_parse_nested_parens.lean	153	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_6_parse_nested_parens.lean	172	example	¡anon¿	OTHER	no marker; tractable

file	line	kind	name	category	reason
humaneval_set/humaneval_6_parse_nested_parens.lean	200	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_6_parse_nested_parens.lean	263	theorem	parseNestedParens_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_7_filter_by_substring.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_82_prime_length.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_82_prime_length.lean	173	theorem	consistency_thm	OTHER	no marker; tractable
humaneval_set/humaneval_82_prime_length.lean	191	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_82_prime_length.lean	272	theorem	primeLength_equivalence_thm	OTHER	no marker; tractable
humaneval_set/humaneval_8_sum_product.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_8_sum_product.lean	287	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_9_rolling_max.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
humaneval_set/humaneval_9_rolling_max.lean	137	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_9_rolling_max.lean	172	example	¡anon¿	OTHER	no marker; tractable
humaneval_set/humaneval_9_rolling_max.lean	191	theorem	correctness_thm	OTHER	no marker; tractable
humaneval_set/humaneval_9_rolling_max.lean	254	theorem	rollingMax_equivalence_thm	OTHER	no marker; tractable
realcode_set/0_bisect_right.lean	134	theorem	bisect_left_partition_thm	OTHER	no marker; tractable
realcode_set/0_bisect_right.lean	142	theorem	bisect_right_partition_thm	OTHER	no marker; tractable
realcode_set/0_bisect_right.lean	171	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/0_bisect_right.lean	187	theorem	bisect_equiv_thm	OTHER	no marker; tractable
realcode_set/10_statistics_mean.lean	115	theorem	spec_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
realcode_set/10_statistics_mean.lean	142	theorem	mean_equiv_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
realcode_set/11_statistics_median.lean	165	theorem	median_equiv_thm	OTHER	no marker; tractable
realcode_set/12_statistics_variance.lean	129	theorem	spec_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
realcode_set/12_statistics_variance.lean	155	theorem	variance_equiv_thm	FLOAT_DEPENDENT	theorem statement or proof goal mentions Float (kernel-opaque)
realcode_set/13_statistics_correlation.lean	157	theorem	spec_thm	OTHER	no marker; tractable
realcode_set/13_statistics_correlation.lean	163	theorem	symmetry_thm	OTHER	no marker; tractable
realcode_set/13_statistics_correlation.lean	193	theorem	correlation_equiv_thm	OTHER	no marker; tractable
realcode_set/15_functools_reduce.lean	157	theorem	reduce_equiv_thm	OTHER	no marker; tractable
realcode_set/16_functools_lru_cache.lean	220	theorem	caching_thm	OTHER	no marker; tractable
realcode_set/16_functools_lru_cache.lean	223	theorem	eviction_thm	OTHER	no marker; tractable
realcode_set/16_functools_lru_cache.lean	259	theorem	lru_equiv_thm	OTHER	no marker; tractable
realcode_set/17_functools_partial.lean	154	theorem	positional_thm	OTHER	no marker; tractable
realcode_set/17_functools_partial.lean	180	theorem	fillSlots_equiv_thm	OTHER	no marker; tractable
realcode_set/17_functools_partial.lean	189	theorem	mergeKw_equiv_thm	OTHER	no marker; tractable
realcode_set/18_textwrap_indent.lean	151	theorem	idempotent_thm	OTHER	no marker; tractable
realcode_set/18_textwrap_indent.lean	178	theorem	indent_equiv_thm	OTHER	no marker; tractable
realcode_set/19_textwrap_dedent.lean	195	theorem	idempotent_thm	OTHER	no marker; tractable
realcode_set/19_textwrap_dedent.lean	222	theorem	dedent_equiv_thm	OTHER	no marker; tractable
realcode_set/1_insort_right.lean	161	theorem	insort_sorted_thm	OTHER	no marker; tractable
realcode_set/1_insort_right.lean	186	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/1_insort_right.lean	197	theorem	insort_equiv_thm	OTHER	no marker; tractable
realcode_set/20_html_escape.lean	96	theorem	ordering_thm	OTHER	no marker; tractable
realcode_set/20_html_escape.lean	123	theorem	escape_equiv_thm	OTHER	no marker; tractable
realcode_set/21_shlex_quote.lean	162	theorem	quote_equiv_thm	OTHER	no marker; tractable
realcode_set/22_shlex_join.lean	183	theorem	join_equiv_thm	OTHER	no marker; tractable
realcode_set/23_base32.lean	179	theorem	length_thm	OTHER	no marker; tractable
realcode_set/23_base32.lean	205	theorem	encode_equiv_thm	PARTIAL_DEPENDENT	references partial def(s): b32encode_func
realcode_set/24_json_decoder.lean	206	theorem	loads_equiv_thm	OTHER	no marker; tractable
realcode_set/25_urllib_parse.lean	193	theorem	safe_thm	OTHER	no marker; tractable
realcode_set/25_urllib_parse.lean	223	theorem	quote_equiv_thm	OTHER	no marker; tractable
realcode_set/25_urllib_parse.lean	228	theorem	unquote_equiv_thm	OTHER	no marker; tractable
realcode_set/26_secrets_token.lean	188	theorem	compare_equiv_thm	OTHER	no marker; tractable
realcode_set/27_graphlib.lean	168	theorem	order_thm	OTHER	no marker; tractable
realcode_set/28_statistics_median_low.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
realcode_set/28_statistics_median_low.lean	350	theorem	median_low_equivalence_thm	OTHER	no marker; tractable
realcode_set/29_statistics_median_high.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
realcode_set/29_statistics_median_high.lean	323	theorem	median_high_equivalence_thm	OTHER	no marker; tractable
realcode_set/2_bisect_left.lean	128	theorem	bisect_left_left_partition_thm	OTHER	no marker; tractable
realcode_set/2_bisect_left.lean	136	theorem	bisect_left_right_partition_thm	OTHER	no marker; tractable
realcode_set/2_bisect_left.lean	164	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/2_bisect_left.lean	175	theorem	bisect_equiv_thm	OTHER	no marker; tractable
realcode_set/30_statistics_covariance.lean	18	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
realcode_set/30_statistics_covariance.lean	215	theorem	shift_invariance_thm	OTHER	no marker; tractable
realcode_set/30_statistics_covariance.lean	292	theorem	covariance_equivalence_thm	OTHER	no marker; tractable
realcode_set/31_statistics_quantiles.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
realcode_set/31_statistics_quantiles.lean	366	example	¡anon¿	OTHER	no marker; tractable
realcode_set/31_statistics_quantiles.lean	376	example	¡anon¿	OTHER	no marker; tractable

file	line	kind	name	category	reason
realcode_set/31_statistics_quantiles.lean	483	theorem	benchmark_quantiles_equivalence_thm	OTHER	no marker; tractable
realcode_set/3_insort_left.lean	162	theorem	insort_sorted_thm	OTHER	no marker; tractable
realcode_set/3_insort_left.lean	187	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/3_insort_left.lean	198	theorem	insort_equiv_thm	OTHER	no marker; tractable
realcode_set/4_heappush.lean	156	theorem	heappush_invariant_thm	OTHER	no marker; tractable
realcode_set/4_heappush.lean	174	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/4_heappush.lean	181	theorem	heappush_equiv_thm	OTHER	no marker; tractable
realcode_set/5_heappop.lean	234	theorem	heappop_invariant_thm	OTHER	no marker; tractable
realcode_set/5_heappop.lean	251	theorem	correctness_thm	OTHER	no marker; tractable
realcode_set/5_heappop.lean	258	theorem	heappop_equiv_thm	OTHER	no marker; tractable
realcode_set/6_queue_simple.lean	200	theorem	get_equiv_thm	OTHER	no marker; tractable
realcode_set/8_collections_counter.lean	181	theorem	update_equiv_thm	OTHER	no marker; tractable
realcode_set/9_collections_chainmap.lean	219	theorem	get_equiv_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	96	theorem	copy_safe	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	122	theorem	safe_copy_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	129	theorem	overflow_detection_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	136	theorem	length_preservation_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	163	theorem	correctness_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	219	theorem	unsafeCopy_equivalence_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/0_unsafeCopy.lean	227	theorem	copy_overflow	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	192	example	¡anon¿	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	193	example	¡anon¿	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	200	theorem	safe_input_safe_request	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	397	theorem	safe_url_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	410	theorem	ssrf_detection_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	426	theorem	safe_handling_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/10_serverSideRequestForgery.lean	454	theorem	correctness_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/1_unsafeMultiply.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/1_unsafeMultiply.lean	99	theorem	unsafeMultiply_detects_overflow	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/1_unsafeMultiply.lean	153	theorem	safe_agreement_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/1_unsafeMultiply.lean	180	theorem	correctness_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/1_unsafeMultiply.lean	242	theorem	safe_agrees_with_unsafe	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/2_unsafeMemoryAccess.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/2_unsafeMemoryAccess.lean	82	theorem	valid_memory_access	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/2_unsafeMemoryAccess.lean	166	theorem	valid_access_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/3_unsafeLinkedList.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/4_unsafeCounter.lean	17	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_6858/vulnerabilities/4_unsafeCounter.lean	152	theorem	unsafe_race_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/4_unsafeCounter.lean	159	theorem	value_bounds_thm	OTHER	no marker; tractable
security_set/security_6858/vulnerabilities/4_unsafeCounter.lean	177	theorem	correctness_thm	OTHER	no marker; tractable
security_set/security_python/safe/0_no_shell_injection_safe.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_python/safe/1_no_command_injection_safe.lean	15	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_python/safe/2_no_executable_escalation_safe.lean	15	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_python/vulnerabilities/0_shell_injection.lean	16	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_python/vulnerabilities/1_command_injection.lean	15	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found
security_set/security_python/vulnerabilities/2_executable_escalation.lean	15	unknown	¡unknown¿	OTHER	no enclosing theorem/example header found

```
# Proof-failure audit  2026-05-04T00:54:17

Dataset root:
  /home/user/veribench/veribench_dataset
    /lean_src/veribench
Total sorries scanned: 576

## Category counts

| Category            | Count | %     |
|---------------------|-------|-------|
| FLOAT_DEPENDENT     |   44  |  7.6% |
| PARTIAL_DEPENDENT   |   38  |  6.6% |
| NATIVE_DECIDE_LEAK  |    0  |  0.0% |
| OTHER               |  494  | 85.8% |

## Per-subset breakdown
```

```
| Subset       | FLOAT | PART | NDL | OTH | Tot |
|--------------|-------|------|-----|-----|-----|
| cs_set       |     0 |   15 |   0 |  36 |  51 |
| easy_set     |     0 |   10 |   0 | 114 | 124 |
| humaneval_set|    40 |   12 |   0 | 249 | 301 |
| realcode_set |     4 |    1 |   0 |  60 |  65 |
| security_set |     0 |    0 |   0 |  35 |  35 |

Columns: FLOAT=FLOAT_DEPENDENT,
         PART=PARTIAL_DEPENDENT,
         NDL=NATIVE_DECIDE_LEAK,
         OTH=OTHER, Tot=Total.

## Interpretation

- FLOAT_DEPENDENT and PARTIAL_DEPENDENT sorries
  are likely unprovable as written (per private
  communication with a domain expert).
- NATIVE_DECIDE_LEAK sorries are theorems that
  lean on native_decide, which only soundly
  discharges ground equalities. If the goal
  contains a free variable, the proof is wrong.
- OTHER is the queue for the Claude/Codex proof
  loop.

## Top files by unprovable-class sorries

| File                                          | Sor |
|-----------------------------------------------|-----|
| cs_set/heapsort_problems.lean                 |  10 |
| humaneval_set/humanevalXL_9__isPrime.lean     |  10 |
| humaneval_set/humaneval_0_hasCloseElements    |   6 |
| humaneval_set/humaneval_20_find_closest       |   6 |
| humaneval_set/humaneval_4_mean_abs_dev        |   6 |
| cs_set/heapsort_build_heap.lean               |   5 |
| humaneval_set/humaneval_21_rescale_to_unit    |   5 |
| humaneval_set/humaneval_2_truncate_number     |   5 |
| humaneval_set/humaneval_32_poly.lean          |   5 |
| humaneval_set/humaneval_45_triangle_area      |   4 |
| easy_set/20_MySumDigits.lean                  |   3 |
| easy_set/23_is_prime.lean                     |   3 |
| humaneval_set/humaneval_32_findZero.lean      |   3 |
| easy_set/21_is_palindrome.lean                |   2 |
| easy_set/24_matrix_multiply.lean              |   2 |
| humaneval_set/humaneval_18_how_many_times     |   2 |
| realcode_set/10_statistics_mean.lean          |   2 |
| realcode_set/12_statistics_variance.lean      |   2 |
| realcode_set/23_base32.lean                   |   1 |
```

I. Limitations and Disclosure

This appendix collects the explicit threats-to-validity disclosures referenced from the main text. We discuss them in full here because each affects how absolute numbers in Tables 2–3 should be read.

Judge family and self-bias risk. The deployed coverage judge for TC_1 in this paper is OpenAI's `gpt-5.3-codex`. One of the evaluated agents—Codex (GPT-5.4)—comes from the same model family. This raises a legitimate self-bias concern: the same family that generates the candidate artifact also estimates its semantic coverage against the gold reference. We adopt this configuration because (i) `gpt-5.3-codex` achieves the strongest held-out human-judge correlation ($r = 0.7033$ under leakage-safe isotonic calibration, Appendix F) among the judge configurations we tested, and (ii) the calibration set was rated by five human raters spanning multiple labs and was scored by an *independent* GPT-5.3-Codex rubric judge, not the deployed judge itself. The unresolved methodological gap is that we have not yet run the deployed judge directly against the human-rated anchor; a cross-family judge swap on a 50–100-task subset (e.g. Claude Opus 4.6 or Gemini 3-pro as judge) is the natural sanity check. This is planned for the camera-ready version and would either confirm ranking stability or surface a quantifiable bias correction. Until that swap is done, agent-level absolute scores in Section 5 should be read as *at risk* of within-family inflation for Codex specifically; the qualitative ordering (Codex $>$ Claude Code $>$ Baseline $>$ Leanstral) is also separately supported by the within-family-judge-free IC_1 and IC_2 factors.

Pass@1 vs. pass@k. Theorem-proving benchmarks frequently report pass@16 or pass@32 from *external* sampling on top of single-shot generation. VeriBench's evaluation does not stack such an outer loop on top of the agent, but this is not a pass@1 setting in disguise. Each agent runs under a fixed 1-hour per-task budget and is free to use that budget to compile,

observe Lean kernel errors, edit, retry, and self-correct as many times as it chooses, submitting only its final artifact. This within-budget tool loop is the agentic analogue of internal pass@k sampling: it explores a search tree of candidate Lean files, prunes branches via compiler feedback, and commits one element of that tree to evaluation. Reporting external pass@k on top of this loop would mix two distinct compute axes—wall-clock budget and outer-sample count—in a way that is not directly comparable to either single-shot ATP benchmarks or to fixed-budget agentic harnesses such as SWE-bench. We choose the fixed-budget framing because it more faithfully represents how these agents are deployed in practice.

Back-translation and judge unidirectionality. Autoformalization pipelines often perform back-translation (Lean → natural language → Lean) to surface translation hallucinations. VeriBench does not currently run a formal back-translation check on agent artifacts as part of TC_1. The two closest analogues already in our pipeline are: (i) the prover-entailment mode of TC_1, in which the Lean kernel verifies that an agent theorem is provable from the gold theorem set (a hard, kernel-checked partial coverage signal); and (ii) bidirectional human-rater calibration of the judge against gold artifacts (Appendix F). Neither catches every hallucinated theorem, and the LLM-judge mode of TC_1 in particular is unidirectional (judge reads gold as ground truth, scores agent against it). A one-pass reverse-judge run on the $n = 75$ human-rated calibration set—scoring the gold theorem set against the agent file rather than the reverse—is the cheapest sanity check for unidirectionality, and is planned for the camera-ready revision.

Judge-optimism risk in autoformalization. Headline numbers in LLM-mediated autoformalization can be sensitive to who is doing the grading. Herald (Gao et al., 2024), for example, reports 96.7% pass@128 on miniF2F-test under its own evaluation pipeline; whether that level of headline accuracy translates to fully kernel-verified end-to-end performance on out-of-distribution tasks is a separate empirical question, and the pattern of high judge-mediated scores on syntactically plausible but semantically incomplete Lean artifacts is now well-documented in the broader autoformalization literature. We anchor our judge-family concern (above) against this risk: agent-level absolute TC_1 numbers mediated by an LLM judge should be read as upper bounds, not as kernel-checked guarantees. The two direct mitigations already in our pipeline are (i) the prover-entailment mode of TC_1 that requires the Lean kernel to discharge the agent theorem from the gold theorem set, and (ii) human-anchored calibration of the deployed judge (Pearson $r = 0.7033$ against five human raters, Appendix F); the planned cross-family judge swap for camera-ready (above) is the third.

Curation does not certify Python semantics. The systematic-translation contrast in Section 3 (Curation pipeline) makes a tradeoff that should be stated explicitly here: the released gold Lean artifact is not accompanied by a meta-theorem that it preserves the semantics of its Python source under all inputs. Compiler frontends such as LeanIO and CompCert-style extracted-semantics pipelines do produce such meta-theorems, at the cost of substantial language-level engineering and restricting the source-language fragment. VeriBench's design goal is different: we evaluate an agent's ability to choose a faithful Lean representation and state the right theorem boundaries from program intent. Curation enforces local agreement (executable tests pass; a second annotator checks the theorem set against the docstring), but a determined adversary could still construct Python inputs where the Lean model and the source program diverge. This affects the interpretation of agent-level TC_1 scores under semantic shift but does not affect within-paper ranking, which is determined relative to a fixed gold reference.

J. Heap-Aliasing Case Study Pointer

The raw `security_6858` dataset contains 227 paired `safe`/`vulnerabilities` Python entries derived from MIT 6.858 labs; the curated VeriBench-SecuritySet evaluated in Section 5 is the 28-task subset for which a gold Lean artifact has been hand-formalized and reviewed. The pairs cover buffer overflows, privilege escalation, and race conditions, and the gold Lean side typically models the C-level buffer as a length-indexed `List` of bytes or a structure with an explicit `Pre` that constrains source and destination lengths; the property theorem set then characterizes the safety invariant that distinguishes the safe variant from the unsafe one (e.g., the safe variant's theorem set includes a no-overflow invariant that the unsafe variant's set cannot establish). A reviewer can inspect a representative task by reading from `veribench_dataset/lean_src/veribench/security_set/security_6858/safe/` and the corresponding `vulnerabilities/` entry alongside the Python sources under `py_src/security_set/security_6858/`. A full end-to-end walkthrough—verbatim gold Lean, one agent's output, per-mode TC_1 resolution (prover-entailment vs. structural-match vs. LLM-judge), and the final SCSC score—is the natural extension of this section for the camera-ready revision.

Agent	Model	IC_1	IC_2	TC_1	D_1	D_2	$\widetilde{S}$	n
Frontier agents (2026 models)								
Oracle (gold)	—	0.898	0.604	1.000	0.898	0.604	0.783	176
Codex (GPT-5.4)	GPT-5.4	1.000	0.237	0.102	0.921	0.570	**0.417**	165
Claude Code (Sonnet 4.6)	Sonnet 4.6	1.000	0.114	0.098	0.921	0.568	0.358	165
Baseline (Sonnet 4.6)	Sonnet 4.6	0.310	0.282	0.105	0.923	0.567	0.344	168
Trace++ (GPT-5.4)	GPT-5.4	0.905	0.143	0.014	0.923	0.567	0.250	168
Trace++ (Opus 4.6)	Opus 4.6	0.141	0.233	0.022	0.926	0.588	0.209	135
Agents from initial round (Claude 3.5 Sonnet)								
DSPy ReAct (Sonnet 3.5)	Sonnet 3.5	0.569	0.310	0.069	0.931	0.491	0.354	58
Baseline (Sonnet 3.5)	Sonnet 3.5	0.268	0.189	0.065	0.923	0.567	0.280	168
Claude Code (Sonnet 3.5)	Sonnet 3.5	0.942	0.072	0.022	0.932	0.578	0.241	103
OpenCode (Sonnet 3.5)	Sonnet 3.5	0.901	0.052	0.017	0.920	0.575	0.211	162
OpenHands (Sonnet 3.5)	Sonnet 3.5	0.715	0.023	0.018	0.927	0.565	0.172	165
AlphaApollo (Sonnet 3.5)	Sonnet 3.5	0.250	0.038	0.035	0.900	0.496	0.171	40
Trace++ Self-Improve (Sonnet 3.5)	Sonnet 3.5	0.400	0.023	0.022	0.921	0.568	0.162	165
Trace+ Self-Debug (Sonnet 3.5)	Sonnet 3.5	0.292	0.008	0.023	0.923	0.567	0.122	168
Aider (Sonnet 3.5)	Sonnet 3.5	0.229	0.000	0.029	0.914	0.532	0.000	35
Specialized and open-source agents								
Leanstral (v2)	Leanstral	0.292	0.065	0.058	0.923	0.567	0.225	168
Vibe Leanstral	Leanstral	0.284	0.082	0.045	0.955	0.549	0.223	88
Leanstral (v1)	Leanstral	0.337	0.046	0.048	0.946	0.541	0.207	92
Vibe Devstral2	Devstral-v2	0.447	0.003	0.034	1.000	0.596	0.122	47
Hybrid DeepSeek+Sonnet	DS-Prover+3.5	0.000	0.000	0.000	1.000	0.574	0.000	55
Hybrid Goedel+Sonnet	Goedel+3.5	0.000	0.000	0.000	0.983	0.570	0.000	59

Table 4. Comprehensive SCSC evaluation across all completed v3 agent runs on VERIBENCH. $\widetilde{S} = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1 \cdot D_1 \cdot D_2)^{1/5}$. IC_1 is the fraction of agent files that compile (Lean-kernel-verified); IC_2 is the fraction of agent theorems without `sorry`; TC_1 is the LLM-judge theorem coverage between agent and gold theorems (median of three Sonnet 4.6 ratings, normalised to $[0, 1]$); D_1, D_2 are the gold-side analogues of $\mathrm{IC}_1, \mathrm{IC}_2$. Runs without completed SCSC summaries are omitted. n = number of tasks with agent output. Source: `experiments/38_scsc_full_metric_evals/expt_v3/results/`.

Method	Pearson $r \uparrow$	Spearman $\rho \uparrow$	Kendall $\tau \uparrow$
Zero-shot (avg labels, $n = 75$)	0.5154	0.4720	0.4115
Isotonic (pass1 $\rightarrow$ pass2, $n = 75$)	0.6537	0.3777	0.3236
Isotonic (pass2 $\rightarrow$ pass1, $n = 75$)	0.5468	0.4008	0.3719
Isotonic CV2 task-split (held-out avg, $n = 75$)	**0.7033**	0.3211	0.2651

Table 5. **Human–judge alignment under task-level held-out evaluation for calibration runs.** Zero-shot row is the raw judge score; isotonic rows apply a post-hoc non-decreasing remapping fit on train tasks only, then evaluate on disjoint held-out tasks. The strongest held-out gain is in Pearson, while rank metrics remain lower.

Method	Pearson p-value	Spearman p-value	Kendall p-value
Zero-shot (avg labels, $n = 75$)	3.60×10^{-6}	2.84×10^{-5}	$\approx 3.18 \times 10^{-7}$
Isotonic (pass1 $\rightarrow$ pass2, $n = 75$)	2.04×10^{-10}	8.34×10^{-4}	1.36×10^{-3}
Isotonic (pass2 $\rightarrow$ pass1, $n = 75$)	6.71×10^{-7}	4.84×10^{-4}	5.72×10^{-4}
Isotonic CV2 task-split (held-out avg, $n = 75$)	$\mathbf{5.61 \times 10^{-12}}$	$\approx 5.95 \times 10^{-3}$	$\approx 9.90 \times 10^{-4}$

Table 6. **p-values for Table 5.**

Method	Pearson 95% CI	Spearman 95% CI	Kendall 95% CI
Zero-shot (avg labels, $n = 75$)	[0.3222, 0.6674]	[0.2698, 0.6343]	[0.2537, 0.5692]
Isotonic (pass1 $\rightarrow$ pass2, $n = 75$)	[0.5011, 0.7669]	[0.1649, 0.5570]	[0.1692, 0.4780]
Isotonic (pass2 $\rightarrow$ pass1, $n = 75$)	[0.3609, 0.6910]	[0.1865, 0.5788]	[0.2142, 0.5296]
Isotonic CV2 task-split (held-out avg, $n = 75$)	[0.5635, 0.8040]	[0.0967, 0.5145]	[0.1073, 0.4228]

Table 7. **95% confidence intervals for correlation estimates.** These intervals quantify uncertainty in the estimated correlations, not run-to-run judge score variation.

Metric	Value	Notes
Item-level MSE	2.4909	overlap $n = 75$ item keys
Item-level MAE	1.2906	overlap $n = 75$ item keys
Task-level MSE	2.2987	overlap $n = 25$ tasks
Task-level MAE	1.2650	overlap $n = 25$ tasks

Table 8. Cross-set disagreement between lenient vs expert/harsh grouped datasets using MSE/MAE.

Group	Items	Raters/item (med.)	SD (mean)	SD (median)	IQR (median)
Lenient set	87	4	0.856	0.816	1.000
Expert/harsh set	71	3	0.624	0.577	0.500

Table 9. Within-group human score dispersion on the 0–5 scale, computed per item and then summarized across items. "Items" is the number of (`gold_file`, `candidate_file`) keys with at least two ratings in that group; "Raters/item (med.)" is the median number of raters contributing to an item in that group. Lower SD/IQR indicates tighter within-group agreement.

Metric	Pairwise mean	Pairwise median
Pearson r	0.496	0.522
Spearman ρ	0.413	0.453
Kendall τ	0.353	0.384
Cohen's κ (linear)	0.287	0.291
Cohen's κ (quadratic)	0.415	0.385

Table 10. Human–human inter-annotator agreement over pairwise rater overlaps. For each rater pair, agreement is computed only on the items both raters scored (pairwise overlap in (`gold_file`, `candidate_file`) keys ranges from 44 to 75), and the table reports the mean and median across all such rater pairs.

Panel	Items	Raters	ICC(2,1)	ICC(2,k)
Five-rater overlap	44	5	0.346	0.726
Expert/harsh overlap	44	3	0.626	0.834

Table 11. Intraclass correlation with absolute agreement on complete-overlap subsets. ICC(2,1) measures the reliability of a single human score; ICC(2,k) measures the reliability of the mean score across the k raters in that panel. The five-rater calculation uses the current one-pass panel (`participant1_data_correct_prompt.tsv`, `participant2`, `participant3`, `participant4`, `participant5`); the expert/harsh row is a sensitivity analysis on the stricter three-rater subset.

Method	Pearson r (vs LLM judge scores)
Zero-shot	**0.964**
Linear OOF	0.962
Isotonic OOF	0.957

Table 12. Strict-set correlation summary (expert/harsh subset). The LLM judge scores are generated with GPT-5.3-Codex with the same `Human Rating Prompt`.

Aggregation experiment	Zero-shot r	Linear OOF r	Isotonic OOF r
All prompts/users averaged	0.8523	0.8084	0.8694
Task-level averaging	0.8519	0.8075	0.8697

Table 13. **Task-level aggregated correlations** between the LLM judge and human raters. Both rows report Pearson r at a coarser granularity than the headline item-level held-out result: scores are first averaged within each task (or across rater/prompt sources for the same item key) before correlation. The resulting correlations (≈ 0.85) are higher than the item-level held-out task-split number ($r = 0.7033$ in Table 5) because aggregation reduces measurement noise; this is the expected statistical effect, not a contradiction. The headline calibration claim in the paper is the item-level held-out value; these aggregated numbers are auxiliary diagnostics and should not be substituted for it.

Agent	Split	IC1	IC2	TE1	D1	D2	IC	TE	D	S_tilde	n
Oracle (gold)	Easy	1.000	0.735	1.000	1.000	0.735	0.857	1.000	0.857	0.884	41
Oracle (gold)	CS	1.000	0.481	1.000	1.000	0.481	0.694	1.000	0.694	0.746	19
Oracle (gold)	HE	0.964	0.463	1.000	0.964	0.463	0.668	1.000	0.668	0.724	56
Oracle (gold)	Real	1.000	0.485	1.000	1.000	0.485	0.697	1.000	0.697	0.749	32
Oracle (gold)	Security	0.429	0.915	1.000	0.429	0.915	0.626	1.000	0.626	0.688	28
Trace++ (GPT-5.4)	Easy	0.927	0.126	0.000	1.000	0.735	0.341	0.000	0.857	0.000	41
Trace++ (GPT-5.4)	CS	0.923	0.141	0.000	1.000	0.478	0.360	0.000	0.692	0.000	13
Trace++ (GPT-5.4)	HE	0.944	0.145	0.007	0.963	0.463	0.370	0.007	0.668	0.214	54
Trace++ (GPT-5.4)	Real	0.844	0.083	0.044	1.000	0.485	0.265	0.044	0.697	0.272	32
Trace++ (GPT-5.4)	Security	0.857	0.237	0.021	0.607	0.656	0.451	0.021	0.631	0.280	28
Leanstral (v2)	Easy	0.463	0.146	0.049	1.000	0.735	0.260	0.049	0.857	0.300	41
Leanstral (v2)	CS	0.154	0.019	0.046	1.000	0.478	0.054	0.046	0.692	0.146	13
Leanstral (v2)	HE	0.296	0.055	0.052	0.963	0.463	0.127	0.052	0.668	0.206	54
Leanstral (v2)	Real	0.125	0.031	0.094	1.000	0.485	0.062	0.094	0.697	0.178	32
Leanstral (v2)	Security	0.286	0.029	0.046	0.607	0.656	0.090	0.046	0.631	0.172	28

Table 14. SCSC v3 detailed table. Source: `detailed_table.csv (expt v3)`. v3 numbers populate Section 5 and Appendix E.

Agent	Split	IC1	IC2	TE1	D1	D2	IC	TE	D	S_tilde	n
Oracle (gold)	Easy	0.976	0.710	1.000	0.976	0.710	0.832	1.000	0.832	0.863	41
Oracle (gold)	CS	1.000	0.481	1.000	1.000	0.481	0.694	1.000	0.694	0.746	19
Oracle (gold)	HE	0.964	0.463	1.000	0.964	0.463	0.668	1.000	0.668	0.724	56
Oracle (gold)	Real	1.000	0.485	1.000	1.000	0.485	0.697	1.000	0.697	0.749	32
Oracle (gold)	Security	0.429	0.915	1.000	0.429	0.915	0.626	1.000	0.626	0.688	28
Codex (GPT-5.4)	Easy	1.000	0.226	0.102	0.976	0.710	0.475	0.102	0.832	0.438	41
Codex (GPT-5.4)	CS	1.000	0.085	0.085	1.000	0.478	0.291	0.085	0.692	0.321	13
Codex (GPT-5.4)	HE	1.000	0.226	0.111	0.962	0.472	0.476	0.111	0.674	0.409	53
Codex (GPT-5.4)	Real	1.000	0.239	0.107	1.000	0.476	0.489	0.107	0.690	0.414	30
Codex (GPT-5.4)	Security	1.000	0.342	0.086	0.607	0.656	0.585	0.086	0.631	0.411	28
claude-code (Sonnet 4.6)	Easy	1.000	0.160	0.095	0.976	0.710	0.400	0.095	0.832	0.402	41
claude-code (Sonnet 4.6)	CS	1.000	0.057	0.077	1.000	0.478	0.239	0.077	0.692	0.291	13
claude-code (Sonnet 4.6)	HE	1.000	0.067	0.094	0.963	0.463	0.259	0.094	0.668	0.309	54
claude-code (Sonnet 4.6)	Real	1.000	0.117	0.123	1.000	0.473	0.342	0.123	0.688	0.369	30
claude-code (Sonnet 4.6)	Security	1.000	0.165	0.089	0.593	0.676	0.407	0.089	0.633	0.358	27
Baseline (Sonnet 4.6)	Easy	0.439	0.412	0.117	0.976	0.710	0.425	0.117	0.832	0.430	41
Baseline (Sonnet 4.6)	CS	0.000	0.000	0.100	1.000	0.478	0.000	0.100	0.692	0.000	13
Baseline (Sonnet 4.6)	HE	0.315	0.259	0.104	0.963	0.463	0.286	0.104	0.668	0.328	54
Baseline (Sonnet 4.6)	Real	0.250	0.203	0.109	1.000	0.485	0.225	0.109	0.697	0.306	32
Baseline (Sonnet 4.6)	Security	0.321	0.357	0.089	0.607	0.656	0.339	0.089	0.631	0.333	28
Leanstral	Easy	0.463	0.146	0.112	0.976	0.710	0.260	0.112	0.832	0.350	41
Leanstral	CS	0.154	0.019	0.100	1.000	0.478	0.054	0.100	0.692	0.170	13
Leanstral	HE	0.296	0.055	0.111	0.963	0.463	0.127	0.111	0.668	0.240	54
Leanstral	Real	0.125	0.031	0.113	1.000	0.485	0.062	0.113	0.697	0.184	32
Leanstral	Security	0.286	0.029	0.054	0.607	0.656	0.090	0.054	0.631	0.177	28

Table 15. SCSC v2 detailed table. Source: `detailed_table.csv (expt v2)`. Superseded by v3; differences are documented in the v3 README.

Agent	Split	C1	C2	C3	C4	C5	IC	TE	D	S_tilde	n
Oracle (gold)	Easy	0.976	0.710	1.000	0.976	0.710	0.832	1.000	0.832	0.863	41
Oracle (gold)	CS	1.000	0.469	1.000	1.000	0.469	0.685	1.000	0.685	0.739	19
Oracle (gold)	HE	0.964	0.463	1.000	0.964	0.463	0.668	1.000	0.668	0.724	56
Oracle (gold)	Real	1.000	0.485	1.000	1.000	0.485	0.697	1.000	0.697	0.749	32
Oracle (gold)	Security	0.429	0.906	1.000	0.429	0.906	0.623	1.000	0.623	0.685	28
Codex (o3)	Easy	0.680	0.209	0.180	0.960	0.701	0.377	0.180	0.820	0.444	25
Codex (o3)	CS	0.556	0.069	0.278	1.000	0.576	0.195	0.278	0.759	0.361	9
Codex (o3)	HE	0.667	0.240	0.278	0.963	0.533	0.400	0.278	0.717	0.470	27
Codex (o3)	Real	0.615	0.147	0.185	1.000	0.478	0.301	0.185	0.692	0.381	13
Codex (o3)	Security	0.500	0.318	0.138	0.688	0.578	0.399	0.138	0.630	0.387	16
Trace+ (GPT-5.4)	Easy	0.951	0.248	0.000	0.976	0.710	0.486	0.000	0.832	0.000	41
Trace+ (GPT-5.4)	CS	0.923	0.154	0.000	1.000	0.478	0.377	0.000	0.692	0.000	13
Trace+ (GPT-5.4)	HE	1.000	0.251	0.000	0.963	0.463	0.501	0.000	0.668	0.000	54
Trace+ (GPT-5.4)	Real	0.781	0.022	0.000	1.000	0.485	0.131	0.000	0.697	0.000	32
Trace+ (GPT-5.4)	Security	0.964	0.096	0.000	0.607	0.558	0.305	0.000	0.582	0.000	28
Trace+ (o3)	Easy	0.756	0.529	0.000	0.976	0.710	0.633	0.000	0.832	0.000	41
Trace+ (o3)	CS	0.692	0.290	0.000	1.000	0.478	0.448	0.000	0.692	0.000	13
Trace+ (o3)	HE	0.778	0.561	0.000	0.963	0.463	0.661	0.000	0.668	0.000	54
Trace+ (o3)	Real	0.625	0.194	0.000	1.000	0.485	0.348	0.000	0.697	0.000	32
Trace+ (o3)	Security	0.536	0.221	0.000	0.607	0.558	0.344	0.000	0.582	0.000	28
Trace+ (o4-mini, run1)	Easy	0.659	0.352	0.000	0.976	0.710	0.482	0.000	0.832	0.000	41
Trace+ (o4-mini, run1)	CS	0.692	0.253	0.000	1.000	0.478	0.418	0.000	0.692	0.000	13
Trace+ (o4-mini, run1)	HE	0.815	0.461	0.000	0.963	0.463	0.613	0.000	0.668	0.000	54
Trace+ (o4-mini, run1)	Real	0.406	0.083	0.000	1.000	0.485	0.183	0.000	0.697	0.000	32
Trace+ (o4-mini, run1)	Security	0.571	0.167	0.000	0.607	0.558	0.309	0.000	0.582	0.000	28
Trace+ (o4-mini, run2)	Easy	0.854	0.485	0.000	0.976	0.710	0.643	0.000	0.832	0.000	41
Trace+ (o4-mini, run2)	CS	0.692	0.290	0.000	1.000	0.478	0.448	0.000	0.692	0.000	13
Trace+ (o4-mini, run2)	HE	0.778	0.390	0.000	0.963	0.463	0.550	0.000	0.668	0.000	54
Trace+ (o4-mini, run2)	Real	0.312	0.008	0.000	1.000	0.485	0.049	0.000	0.697	0.000	32
Trace+ (o4-mini, run2)	Security	0.750	0.270	0.000	0.607	0.558	0.450	0.000	0.582	0.000	28
Trace+ (o4-mini, run3)	Easy	0.805	0.435	0.000	0.976	0.710	0.591	0.000	0.832	0.000	41
Trace+ (o4-mini, run3)	CS	0.846	0.303	0.000	1.000	0.478	0.506	0.000	0.692	0.000	13
Trace+ (o4-mini, run3)	HE	0.722	0.441	0.000	0.963	0.463	0.564	0.000	0.668	0.000	54
Trace+ (o4-mini, run3)	Real	0.344	0.139	0.000	1.000	0.485	0.218	0.000	0.697	0.000	32
Trace+ (o4-mini, run3)	Security	0.464	0.166	0.000	0.607	0.558	0.278	0.000	0.582	0.000	28
Trace+ (GPT-4.1)	Easy	0.707	0.391	0.000	0.976	0.710	0.526	0.000	0.832	0.000	41
Trace+ (GPT-4.1)	CS	0.615	0.236	0.000	1.000	0.478	0.381	0.000	0.692	0.000	13
Trace+ (GPT-4.1)	HE	0.741	0.442	0.000	0.963	0.463	0.572	0.000	0.668	0.000	54
Trace+ (GPT-4.1)	Real	0.312	0.081	0.000	1.000	0.485	0.159	0.000	0.697	0.000	32
Trace+ (GPT-4.1)	Security	0.464	0.121	0.000	0.607	0.558	0.237	0.000	0.582	0.000	28
Codex-old (GPT-4o, run1)	Easy	0.000	0.000	0.000	0.976	0.710	0.000	0.000	0.832	0.000	41
Codex-old (GPT-4o, run1)	CS	0.000	0.000	0.000	1.000	0.478	0.000	0.000	0.692	0.000	13
Codex-old (GPT-4o, run1)	HE	0.000	0.000	0.000	0.963	0.463	0.000	0.000	0.668	0.000	54
Codex-old (GPT-4o, run1)	Real	0.000	0.000	0.000	1.000	0.485	0.000	0.000	0.697	0.000	32
Codex-old (GPT-4o, run1)	Security	0.000	0.000	0.000	0.607	0.558	0.000	0.000	0.582	0.000	28
Codex-old (GPT-4o, run2)	Easy	0.537	0.384	0.000	0.976	0.710	0.454	0.000	0.832	0.000	41
Codex-old (GPT-4o, run2)	CS	0.154	0.154	0.000	1.000	0.478	0.154	0.000	0.692	0.000	13
Codex-old (GPT-4o, run2)	HE	0.370	0.258	0.000	0.963	0.463	0.309	0.000	0.668	0.000	54
Codex-old (GPT-4o, run2)	Real	0.250	0.177	0.000	1.000	0.485	0.210	0.000	0.697	0.000	32
Codex-old (GPT-4o, run2)	Security	0.214	0.165	0.000	0.607	0.558	0.188	0.000	0.582	0.000	28

Table 16. SCSC v1 detailed table. Source: `detailed_table.csv (expt v1)`. The C_i columns correspond to early names for the SCSC factors that were renamed to $\mathrm{IC}_1, \mathrm{IC}_2, \mathrm{TC}_1, D_1, D_2$ in v2/v3.