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

Brando Miranda[1,†] **Srivatsava Daruru**[1] **Ethan S. Hersch**[1] **Zhanke Zhou**[4] **Allen Nie**[1,5] **Daneshvar Amrollahi**[1] **Leni Aniva**[1] **Iddah Mlauzi**[1] **Kirill Acharya**[1] **Elyas Obbad**[1] **Dilara Soylu**[1] **Weston Kirk**[1] **Zixiao Jolene Wang**[2] **Kai Fronsdal**[1] **Ying Li**[1] **Donald Poindexter Jr.**[1] **Rakshit Kaushik**[1] **Shurui Liu**[1] **Yegor Denisov-Blanch**[1] **Steven Dillmann**[1] **Simon Obstbaum**[1] **Santiago Cuellar**[7] **John Sarracino**[8] **Rylan Schaeffer**[1] **Mo Tiwari**[1,6] **Donghyun Lee**[3] **Bo Han**[4] **Sanmi Koyejo**[1]

[1]Stanford University [2]Harvard University [3]UC Berkeley [4]Hong Kong Baptist University
[5]Google DeepMind [6]Google [7]Galois, Inc. [8]Lawrence Livermore National Laboratory
[†]Correspondence: `brando9@stanford.edu`

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: finite test suites can 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*, so that weakness at any stage strongly penalizes the score while preserving partial-credit signal among sub-frontier systems. We introduce VERIBENCH, an 884-task end-to-end Python-to-Lean 4 autoformalization benchmark whose canonical core (602 tasks) spans HumanEval-style programs, classical algorithms, Python standard-library functions, introductory programming tasks, and a 460-task security-critical subset adapted from the MIT 6.858 corpus, complemented by a 282-task high-assurance-inspired expansion across 14 domains such as cryptography, aerospace, medical devices, and compilers. We report SCSC as an evaluation-chain score and separately decompose agent-side skill ($\mathrm{IC}_1, \mathrm{IC}_2, \mathrm{TC}_1$) from gold-side benchmark quality (D_1, D_2): the agent-skill component checks (i) the generated Lean file typechecks, (ii) the stated theorems verify without `sorry`, and (iii) these theorems semantically cover the gold reference; the gold-side component checks whether the gold reference itself passes its tests and proves cleanly. Under an agentic verifier-feedback loop, Codex, Claude Code, and Leanstral v2 reach an agent-skill score $\widetilde{S}_{\mathrm{skill}}$ of only 0.29, 0.22, and 0.10 (with the five-factor evaluation-chain $\widetilde{S}_5$ at 0.42, 0.36, and 0.23, and gold-side $\widetilde{Q}_{\mathrm{gold}} \approx 0.72$); Codex and Claude Code typecheck all generated files, yet *estimated* theorem–gold coverage remains at ≤ 0.156 across the evaluated agents (with the strongest agents on the headline ranking, Codex/-Claude Code/Baseline, capped at ≤ 0.105)—a *theorem-coverage gap* estimated by an LLM coverage judge whose held-out task-split agreement with five independent human raters reaches Pearson $r = 0.70$ under leakage-safe isotonic calibration (zero-shot $r = 0.52$; rank correlations weaker, $\rho{\approx}0.32$). VERIBENCH reframes code-verification evaluation from isolated proof search to end-to-end conjunctive

Preprint.

grounding, surfacing theorem formulation 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 (Amrollahi et al., 2026): finite testing routinely misses critical bugs. As AI coding agents graduate from isolated programming puzzles to complex software engineering, this structural limitation becomes a critical bottleneck. The cost of test-invisible bugs is most severe in security-sensitive and high-assurance domains, an acute risk given that LLM-generated code frequently harbors high-risk weaknesses (Pearce et al., 2022; Perry et al., 2023). Trustworthy evaluation in these critical applications demands a stronger signal than test passing: safety properties quantify over unbounded, partly adversarial inputs, meaning finite testing can expose bugs but never rule them out.

Formal verification offers such a signal: a machine-checkable certificate that a program's intended behavior holds universally for all possible inputs (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 this direction, but they evaluate isolated slices of the pipeline rather than the agentic workflows that have rapidly become the standard paradigm for AI coding assistants (Qian et al., 2024; Wang et al., 2024b; Yang et al., 2024; Wang et al., 2025). VERINA (Ye et al., 2025) and CLEVER (Thakur et al., 2025) feed models natural-language descriptions or partial scaffolds to score subtasks in isolation—a setup that abstracts away the core challenge of modern agentic coding: autonomously formalizing properties directly from raw source code. Similarly, FVAPPS (Dougherty & Mehta, 2025) and DAFNYBENCH (Loughridge et al., 2024) score proof completion against pre-written specifications, removing the critical burden of formulating the right theorem in the first place. 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 generated theorems fail to 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, an 884-task end-to-end Python-to-Lean 4 autoformalization benchmark whose canonical core (602 tasks) spans HumanEval-style programs (Chen et al., 2021), classical algorithms, Python standard-library functions, introductory programming tasks, and a 460-task security-critical subset adapted from MIT 6.858 (CSAIL, 2014), complemented by a 282-task high-assurance-inspired expansion 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 generated Lean file typechecks, (ii) the stated theorems verify without `sorry`, (iii) these theorems semantically cover the gold reference, and (iv,v) gold-side benchmark-validity gates (D_1, D_2) measuring whether the gold reference itself passes its tests and proves cleanly before being used to score agents (Section 4). 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.

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.

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 structured Lean 4 artifact—implementation, tests, theorems, and proofs. Codex (GPT-5.4), Claude Code (Sonnet 4.6), and Leanstral v2 reach agent-skill scores $\widetilde{S}_{\text{skill}} = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1)^{1/3}$ of only 0.29, 0.22, and 0.10, respectively (corresponding five-factor evaluation-chain scores $\widetilde{S}_5$: 0.42, 0.36, 0.23); Codex and Claude Code typecheck all generated files, but this does not close the theorem-coverage gap. Strikingly, overall theorem–gold coverage stalls at ≤ 0.156 across the evaluated agents—and at ≤ 0.105 for the headline agents (Codex, Claude Code, Baseline): agents routinely produce Lean files that typecheck, yet the theorems they state systematically fail to cover the corresponding gold theorems written by a human curator for the same task. We name this regularity the *theorem-coverage gap* and validate it with an LLM judge whose held-out task-split agreement with five independent human raters reaches Pearson $r = 0.70$ under leakage-safe isotonic calibration (zero-shot $r = 0.52$; Appendix G). This held-out value is not zero-shot; it is post-hoc monotone calibration evaluated on disjoint test tasks. The gap appears across program families and persists under self-correction, indicating that theorem formulation is a bottleneck at least as severe as proof search.

VERIBENCH reframes code-verification evaluation from isolated proof search to end-to-end conjunctive grounding, surfacing theorem formulation 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: 884 tasks running from developer-written Python source to checked Lean 4 artifacts, with a 602-task canonical core (HumanEval-style, classical algorithms, Python standard-library code, introductory programming, and a 460-task MIT 6.858-derived security split) and a 282-task high-assurance-inspired expansion across 14 industrial domains (Section 3, Appendix D).
- **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 theorem-coverage gap.** Frontier agents reach $\widetilde{S}_{\text{skill}} \leq 0.29$ (five-factor $\widetilde{S}_5 \leq 0.42$) with overall theorem–gold coverage ≤ 0.156 (and ≤ 0.105 for the headline agents), surfacing theorem formulation as a bottleneck at least as severe as proof search (Section 5).
- **A human-calibrated coverage judge.** An LLM coverage judge with held-out task-split Pearson $r = 0.70$ vs. five independent human raters under leakage-safe isotonic calibration (zero-shot $r = 0.52$; rank correlations weaker, $\rho \approx 0.32$), enabling scalable evaluation that tracks expert assessment of theorem semantic coverage (Appendix G).

2 Related Work

Benchmarks for Code Verification. Loughridge et al. (2024) introduce DAFNYBENCH, a large benchmark for evaluating LLMs in formal software verification. It contains over 750 Dafny programs stripped of verification hints, requiring models to regenerate the missing annotations to pass the verifier. Their best-performing system reaches roughly 68% success, while performance varies with program size, hint complexity, and retry strategy.

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 into Lean models, using Python as a motivating example given its ubiquity (GitHub, 2024), we measure whether agents can choose representations, modular structure, and theorem boundaries without a scaffold. This aligns with the practical necessity of verifying existing codebases rather than enforcing native Lean development. Furthermore, we eschew purely test-based specification evaluation, which fails on non-computable theorems, in favor of a theorem-coverage check that uses prover-established entailment when feasible and a calibrated semantic judge otherwise.

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 specification and a correctness-proofed implementation. Its non-computable, specification-agnostic setup avoids test leakage and demands theorem-level reasoning; current 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 structured Lean 4 artifact—functional and imperative implementations, unit tests, correctness theorem statements, and proof attempts whose remaining `sorry` placeholders are reported explicitly—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 checks fully or accepts only with explicit `sorry` placeholders. We use "machine-checkable" to mean kernel-typechecked: theorem statements and closed proof bodies are checked by Lean, while proof bodies 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 contains 884 tasks organized into six task families that progress from introductory reasoning through classical algorithms, security-critical code, and production utility code, to industry-elicited high-assurance domains.

- **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**: 460 tasks comprising a curated paired `security_6858` subset of 227 safe/vulnerable Lean pairs (454 tasks) drawn from the MIT 6.858 *Computer Systems Security* corpus, plus 6 standalone `security_python` examples covering shell injection, command injection, and privilege-escalation patterns. The paired structure exposes 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; Appendix 3 lists the module/function families and adaptation type.
- **VeriBench-IndustrySet**: 282 tasks across 14 industry-elicited high-assurance domains (crypto, RFC, memory safety, aerospace, regulated medical/automotive, finance, embedded OS, hardware, concurrent, hypervisor, compilers, gov-cert, critical infrastructure), built by a license-verified real-OSS hunt (pyca/cryptography, cpython stdlib, OpenAPS, NASA F Prime, rust-vmm) plus domain-prototypical synthetic kernels where canonical sources are GPL/proprietary; taxonomy and per-domain provenance in Appendix D.10.

The SCSC results in Section 5 evaluate the canonical core's five splits (HumanEval, EasySet, CSSet, SecuritySet, RealCodeSet) across 176 Lean evaluation units after multi-formalization; VeriBench-IndustrySet (282 tasks) is evaluated separately at compile-rate granularity in Appendix I.

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.

Numeric and termination semantics. Float-using and `partial def`-using gold files would otherwise conflate agent skill with library-choice artifacts: Lean's primitive `Float` is opaque to kernel-level reasoning, the ecosystem lacks a single Mathlib-endorsed IEEE-754 formalization (Rute & Parker, 2025), and `partial def` implementations are similarly opaque. VERIBENCH therefore makes numeric semantics explicit through layered specifications (a runtime `FloatLayer` for ground evaluation, mathematical-intent `RealLayer`/`RatLayer`, and optional `FloatSpecLayer`/`FleanLayer`/`IntervalLayer` for tasks marked as floating-point-essential) declared in-file as a coverage-judge-readable policy comment, and replaces `partial def` with terminating definitions where feasible. The full layer taxonomy, the audit that motivated the repair, and the per-task layer policy template are in Appendix J.

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:

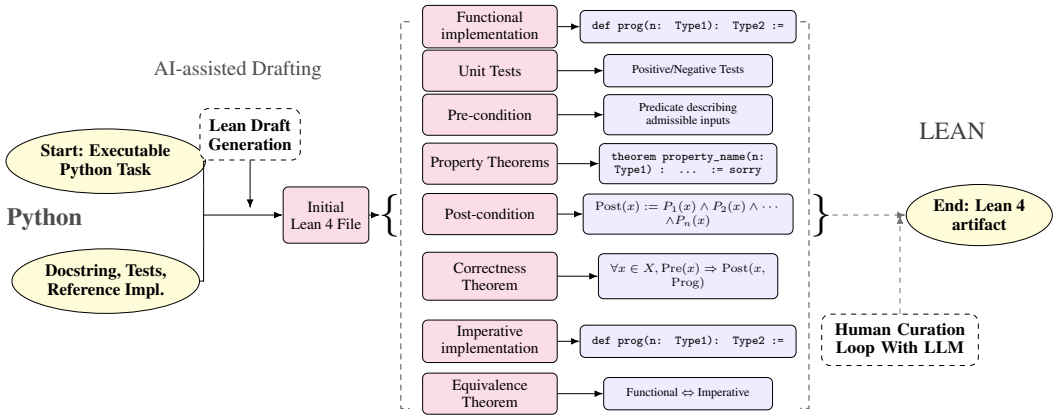


Figure 1: VeriBench construction pipeline: executable Python tasks are curated into standardized Lean 4 gold artifacts with implementations, tests, specifications, and proof attempts.

curators identify the central properties for each task and make them explicit. **Curation pipeline.** A single written rubric—a gold-reference template (implementation, tests, pre/post-conditions, property propositions, correctness, optional imperative + equivalence) and a review checklist—guides both the AI drafting pass (`o3`) and a second-annotator human review pass; the rubric is released with the benchmark. The reviewer (paper author or Lean/PL/formal-methods collaborator, working independently from the drafter) checks that theorem statements characterize behavior rather than restate the implementation, pre/post-conditions are non-trivial, tests cover positive/negative/edge cases, and every proof either closes or carries an explicit `sorry` counted by D_2. Disagreements are resolved in the GitHub issue thread; *final merges to `main` are always performed by a human reviewer*, never by an AI agent. These edits often change types, theorem statements, or proof strategies; provenance is recorded in repository history, with Appendix D.2 documenting the rubric and the current limits of external auditing.

Validity. VeriBench mitigates common risks in LLM-assisted benchmark construction through human curation, Lean typechecking, and execution-based validation. Every Python reference file has passing unit tests. Gold Lean files are expected to compile and discharge their proofs; remaining failures are exposed transparently through D_1 and D_2 rather than hidden. When a gold file compiles, the Lean kernel checks the implementations, tests, theorem statements, and any proof bodies that do not contain `sorry`. 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 D gives the construction pipeline, gold-file schema, Python input standard, and additional validity discussion.

Illustrative example. Appendix D gives the full task schema and examples; the main text focuses on benchmark design, metric semantics, and measured failure modes.

4 Evaluation Metric: The Smooth Conjunctive Score for Code Verification

VERIBENCH evaluates whether an agent can turn a Python task into a Lean 4 artifact that both typechecks and states theorem obligations covering the gold reference. Let L^* denote the human-curated gold Lean file, with implementation $L^*.f$, theorem set $L^*.\mathrm{Thms}$, and tests $L^*.T$. Let $\hat{L}$ denote the agent-generated Lean file, with analogous components. The score combines five factors:

$$\mathrm{SCSC}(\hat{L}, L^*) = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1 \cdot D_1 \cdot D_2)^{1/5}.$$

Equivalently, for positive factors f_i, it can be computed as

$$\mathrm{SCSC} = \exp\left(\frac{1}{5}\sum_{i=1}^{5} \log f_i\right).$$

If any factor is zero, the product convention sets $\mathrm{SCSC} = 0$; the implementation does not take $\log 0$. Table 2 reports SCSC of per-factor task-averages, $\mathrm{SCSC}(\bar{\mathrm{IC}}_1, \bar{\mathrm{IC}}_2, \bar{\mathrm{TC}}_1, \bar{D}_1, \bar{D}_2)$, which by Jensen's inequality differs from the macro-average $\overline{\mathrm{SCSC}}(\hat{L}_t, L_t^*)$; the released per-task CSV reports both readings. Figure 2 (Appendix F) gives the colored evaluation topology: which artifact each factor compares ($\hat{L}$ vs L^*), and how the agent-side ($\mathrm{IC}_1, \mathrm{IC}_2, \mathrm{TC}_1$) and gold-side ($D_1, D_2$) factors split.

Agent-side factors. IC_1 is the typechecking factor: it is 1 when the generated file resolves imports and Lean accepts it as a well-typed file, and 0 otherwise. IC_2 is the sorry-free proof factor. For a typechecking generated file with at least one theorem or lemma declaration,

$$\mathrm{IC}_2 = \frac{\#\{\text{stated theorem/lemma declarations without } \texttt{sorry/admit}\}}{\#\{\text{stated theorem/lemma declarations}\}}.$$

If the file does not typecheck or states no theorem/lemma declarations, $\mathrm{IC}_2 = 0$. Thus an agent can improve IC_2 only by proving the claims it states; deleting or weakening claims may raise IC_2 locally but lowers TC_1 if the gold obligations are no longer covered.

TC_1 is theorem coverage: it measures whether the agent's theorem set semantically covers the gold theorem obligations. Coverage is one-directional: a stronger agent theorem may cover a weaker gold obligation, but vacuous or narrower statements do not. The scorer uses three modes, in decreasing rigor: Lean cross-implication when feasible, syntactic/structural matching for alpha-equivalent or trivially rewritten statements, and a calibrated LLM semantic judge otherwise. The deployed v3 TC_1 judge in Section 5 is a Claude Sonnet 4.6 LLM judge: per item, k=3 independent calls produce 0–10 ratings whose median is normalized to $[0, 1]$. As post-hoc validation that LLM-as-coverage-judge tracks human labels in this regime, an *independent* GPT-5.3-Codex 0–5 rubric judge reaches held-out task-split Pearson $r = 0.7033$ against five independent human raters under leakage-safe **CV2 task-split isotonic** calibration (zero-shot $r = 0.5154$; rank correlations are weaker, $\rho = 0.3211$, $\tau = 0.2651$; Appendix G); this is calibration of the rubric judge, not direct calibration of the deployed Sonnet judge. Self-evaluation between the deployed Sonnet 4.6 judge and the same-family agent (Claude Code) is an additional confound we cannot rule out. Headline TC_1 values should therefore be read as a *Pearson-coverage estimate*—scale-matched alignment with human raters via the rubric judge above—not as rank-preserving equivalence or kernel-checked entailment.

Gold-side gates. D_1 and D_2 are benchmark-validity gates, not agent-skill factors. D_1 checks whether the gold tests pass on the gold implementation. D_2 checks whether the gold theorem declarations prove the gold implementation without `sorry` or `admit`. Low D_1 or D_2 therefore indicates benchmark item immaturity or incomplete gold proofs and is reported transparently rather than hidden.

Decomposition: agent-side skill vs. gold-side quality. Because the five-factor SCSC mixes agent-side factors ($\mathrm{IC}_1, \mathrm{IC}_2, \mathrm{TC}_1$) with gold-side gates ($D_1, D_2$), we report two principled subscores alongside it so the agent contribution is not entangled with benchmark-item maturity:

$$\widetilde{S}_{\mathrm{skill}}(\hat{L}, L^*) = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1)^{1/3}, \qquad \widetilde{Q}_{\mathrm{gold}}(L^*) = (D_1 \cdot D_2)^{1/2}.$$

$\widetilde{S}_{\mathrm{skill}}$ is the headline *agent-skill* number: it depends only on the agent's output, so a stronger gold reference cannot inflate it. $\widetilde{Q}_{\mathrm{gold}}$ is a property of the benchmark item, not of the agent, and is approximately constant across agents evaluated against the same gold set. The five-factor SCSC is therefore best read as an *evaluation-chain* score reflecting end-to-end conjunctive grounding (agent skill *and* benchmark validity), not as a pure agent-skill estimate; we report it for chain-health diagnostics, with $\widetilde{S}_{\mathrm{skill}}$ and $\widetilde{Q}_{\mathrm{gold}}$ carrying the load of the headline agent ranking. A *Verified Core* sensitivity view restricted to items with $D_1 = D_2 = 1$ is presented in Appendix F so that the strongest formal-verification claims rest only on items whose gold reference is fully closed.

Conjunctive aggregation. The distinction between arithmetic and geometric aggregation is semantic, not merely quantitative. An arithmetic mean is compensatory: a high typechecking score could offset a near-zero coverage score. That is misaligned with formal verification, where one broken stage can invalidate the artifact. The product, reported as a geometric mean to preserve the 0–1 scale, implements a smooth conjunction: any zero factor collapses the score, while nonzero partial progress remains distinguishable.

Numeric specification layers. For files with a numeric specification-layer header, TC_1 is evaluated modulo the declared accepted layers. The agent need not match the gold namespace syntactically: a theorem over $\mathbb{Q}$, $\mathbb{R}$, an IEEE-754 library, or interval semantics may receive full coverage credit if it states an equivalent or stronger obligation under the declared layer semantics. Conversely, a theorem stated only in primitive `FloatLayer` does not automatically cover a mathematical or IEEE obligation; because `FloatLayer` is runtime-only by default, the agent must either prove the claim in an accepted proof layer or provide a proved bridge from its chosen numeric model to the gold obligation. Optional numeric layers that are not part of the scored gold layer are included as comments or fully proved declarations only, so experimental floating-point stubs do not depress D_2.

Theorem coverage operationalized. TC_1 runs in one of three modes, in order of decreasing rigor: prover-established entailment, syntactic/structural matching, and an LLM-judge fallback (calibration above; full mode definitions in Appendix F.3). In our current evaluation, the LLM-judge mode dominates reported TC_1 values, so headline TC_1 should be read as "human-aligned semantic coverage estimate," not formal entailment. For multi-layer files (Section 3), the judge's cross-layer rules give the same TC_1 credit to semantically equivalent theorems stated over $\mathbb{R}$, $\mathbb{Q}$, or FloatSpec; an agent that states a `Float` theorem and leaves it `sorry` receives zero IC_2 credit for it.

Coverage, not equivalence; no cross-apply. Coverage is one-directional: stronger agent theorems entail the gold obligation and are credited, while strict equivalence would punish additional guarantees. We also drop earlier designs that cross-applied gold tests/theorems to agent code, because signature/representation mismatches caused systematic false negatives unrelated to correctness; $\mathrm{IC}_2 \wedge \mathrm{TC}_1$ is the empirical proxy. Floating-point caveats are audited in Appendix J.

Threats to validity. We name five: *judge dominance* of TC_1 (mode 3 carries the bulk of reported values), *single-run point estimates* for closed-source agents, the *coverage proxy* ($\mathrm{IC}_2 \wedge \mathrm{TC}_1$ is empirical, not a kernel-checked transitivity proof), *no human-as-agent ceiling*, and *gold comprehensiveness* bounded by the released theorem set; each is detailed in Appendix F.2.

5 Evaluation

We evaluate five agents on the canonical VERIBENCH core (HumanEval-style, EasySet, CSSet, SecuritySet, RealCodeSet), expanded to 176 Lean evaluation units after multi-formalization; the 282-task VeriBench-IndustrySet is evaluated separately at compile-rate granularity in Appendix I. 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; **Leanstral v2**, a Lean-specialized model; and **Gemini 3-flash-preview**, run through an AlphaApollo iterative scaffold. Each agent operates autonomously: it receives only the Python file and must generate the structured Lean formalization end-to-end.

Experimental setup. All agents operate in the same evaluation harness against Lean 4 toolchain `leanprover/lean4:v4.22.0` (matched to the project CI; see `.github/workflows/ci.yml`) with Mathlib `v4.22.0`. Each task runs in a fresh Harbor (Harbor Framework Team, 2026) Docker container with the Python input, a writable Lean workspace, internet access enabled for package/documentation lookup, and no mounted gold Lean reference. The per-task budget is one hour for the agent, one hour for verification, and one hour for environment build; the main runs use 3–5 concurrent task containers depending on agent cost. Codex and Claude Code use their tool-native verifier-feedback loops within that one-hour budget, Leanstral v2 uses a fixed iterative Lean-feedback adapter, and Baseline is a single LLM call without tool use. Outputs are parsed by extracting the last Lean fenced block from the agent's transcript, written to a single Lean file, and compiled with `lake env lean` under the same toolchain; failure to compile counts as $\mathrm{IC}_1 = 0$ and forces $\mathrm{IC}_2 = 0$. Generated proofs are allowed to use `sorry`, and generated artifacts are scanned for `axiom` declarations and unsafe escape hatches. The exact prompts, Docker settings, timeout policy, and result-table source paths are released with the artifact; Appendix D.8 gives the smoke-test and reproduction commands.

Agent	IC_1	IC_2	TC_1	D_1	D_2	n	$\widetilde{S}_{\mathrm{skill}}$	$\widetilde{Q}_{\mathrm{gold}}$	$\widetilde{S}_5$
Codex (GPT-5.4)	1.000	0.237	0.102	0.921	0.570	165	**0.289**	**0.725**	**0.417**
Claude Code (Sonnet 4.6)	1.000	0.114	0.098	0.921	0.568	165	0.224	0.723	0.358
Baseline (Sonnet 4.6)	0.310	0.282	0.105	0.923	0.567	168	0.209	0.723	0.344
Gemini 3-flash-preview[†]	0.206	0.043	0.156	0.979	0.558	141	0.111	0.739	0.237
Leanstral (v2)	0.292	0.065	0.058	0.923	0.567	168	0.103	0.723	0.225

Table 2: Agent-skill and benchmark-quality scores on the canonical VERIBENCH core (176 evaluation units across HumanEval-style, EasySet, CSSet, SecuritySet, RealCodeSet). **Per-factor definitions (Section 4):** $\mathrm{IC}_1 \in \{0,1\}$ — agent's Lean file typechecks; IC_2 — fraction of agent-stated theorems closed without `sorry`/`admit`; TC_1 — LLM-coverage-judge estimate of how well agent theorems cover the gold reference; deployed judge is Claude Sonnet 4.6 (median of k=3 0–10 ratings, normalized); validation against five human raters via an independent GPT-5.3-Codex rubric judge reaches held-out task-split Pearson $r = 0.7033$ under leakage-safe isotonic calibration (zero-shot $r = 0.5154$; rank correlations weaker, $\rho \approx 0.32$; Appendix G); $D_1 \in \{0,1\}$ — gold tests pass on the gold implementation; D_2 — fraction of gold theorems closed without `sorry`/`admit`; n — number of tasks the agent produced output for. **Aggregate scores:** $\widetilde{S}_{\mathrm{skill}} = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1)^{1/3}$ — agent-only headline ranking, invariant to gold-side gates; $\widetilde{Q}_{\mathrm{gold}} = (D_1 \cdot D_2)^{1/2}$ — benchmark-item gold-quality, approximately constant across agents (the gold reference is fixed); $\widetilde{S}_5 = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1 \cdot D_1 \cdot D_2)^{1/5}$ — conditional five-factor evaluation-chain score (full-benchmark variant $\widetilde{S}_{5,\mathrm{full}}$ in Appendix F). The vertical line separates per-factor inputs (left) from aggregate scores (right); the leader of each aggregate column is **bold**. [†]Gemini 3-flash-preview was scored under the same SCSC pipeline but on the new Harbor task expansion ($n = 141$ unique-task trials), versus the multi-formalization trial set ($n \in \{165, 168\}$) used by the other agents; auxiliary 3-flash/3-pro/3.1-pro runs on the full 884-task benchmark are in Appendix I.3.

5.1 Overall Results

Reporting policy. Table 2 reports overall scores across 176 gold Lean files spanning the canonical core's five splits. Per the decomposition in Section 4, the headline agent-skill score is $\widetilde{S}_{\mathrm{skill}} = (\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1)^{1/3}$, reported alongside the gold-side quality $\widetilde{Q}_{\mathrm{gold}} = (D_1 \cdot D_2)^{1/2}$ and the five-factor evaluation-chain score $\widetilde{S}_5$; per-agent denominators range $n \in [141, 176]$, and we report both conditional and full-benchmark (no-output $\to 0$, $N = 176$) variants of $\widetilde{S}_5$.

First, on the directly-comparable set ($n \in [165, 168]$), the ranking is Codex at $\widetilde{S}_{\mathrm{skill}} = 0.289$, Claude Code at 0.224, Baseline at 0.209, Leanstral v2 at 0.103; Gemini 3-flash (0.111, $n = 141$, different task set per dagger) enters for breadth and is not directly comparable. $\widetilde{Q}_{\mathrm{gold}} \approx 0.72$–$0.74$ holds across rows, so the ordering is not contaminated by gold-set immaturity. Second, theorem coverage (TC_1) is uniformly low: Codex/Claude Code/Baseline score ≤ 0.105 and Gemini 3-flash reaches 0.156, but Gemini's higher TC_1 does not lift its $\widetilde{S}_{\mathrm{skill}}$ because $\mathrm{IC}_1 = 0.21$ and $\mathrm{IC}_2 = 0.04$ are far weaker than the leaders'. Third, internal consistency splits sharply—Codex and Claude Code achieve $\mathrm{IC}_1 = 1.0$ but $\mathrm{IC}_2 \leq 0.237$; against the Sonnet 4.6 baseline, Codex improves $\widetilde{S}_{\mathrm{skill}}$ by 38.3% and Claude Code by 7.2%, so verifier feedback helps but does not close the theorem-coverage gap. Per-split breakdown, Verified-Core sensitivity ($D_1 = D_2 = 1$), and the auxiliary 884-task expansion run are in Appendix I.

6 Discussion and Limitations

A detailed limitations and trade-offs discussion—scalable-oversight framing, shallow embedding vs. full Python semantics, TC_1 as Pearson-coverage estimate, single-run point estimates, canonical-core SCSC scope, partly-synthetic VeriBench-IndustrySet, and the no-human-as-agent-ceiling decision—is in Appendix B.

What the benchmark measures. VERIBENCH evaluates Python-to-Lean autoformalization under a shallow embedding: agents produce Lean 4 artifacts that typecheck, close stated theorems without `sorry`, and cover the human-curated gold obligations—not a Python-semantics-equivalence proof.

Acknowledgments

BM acknowledges support by Schmidt Sciences, Stanford EDGE Scholar Fellowship, NSF 2046795 and 2205329, the MacArthur Foundation, Stanford HAI, OpenAI and Google Inc.

RS acknowledges support from Stanford Data Science and from the OpenAI Superalignment Fast Grant.

SK acknowledges support by NSF 2046795 and 2205329, the MacArthur Foundation, Stanford HAI, OpenAI and Google Inc.

Author Emails

Brando Miranda	brando9@stanford.edu
Srivatsava Daruru	srivatsavad@stanford.edu
Ethan S. Hersch	ehersch@stanford.edu
Zhanke Zhou	cszkzhou@comp.hkbu.edu.hk
Allen Nie	allennie@google.com
Daneshvar Amrollahi	daneshvar@cs.stanford.edu
Leni Aniva	aniva@stanford.edu
Iddah Mlauzi	iddah@stanford.edu
Kirill Acharya	kacharya@stanford.edu
Elyas Obbad	eobbad@stanford.edu
Dilara Soylu	soylu@cs.stanford.edu
Weston Kirk	wkirk@stanford.edu
Zixiao Jolene Wang	zixiaowang@g.harvard.edu
Kai Fronsdal	kaif@stanford.edu
Ying Li	louisely@stanford.edu
Donald Poindexter Jr.	poin@stanford.edu
Rakshit Kaushik	rakshit.kaushik2772@gmail.com
Shurui Liu	srliu3264@gmail.com
Yegor Denisov-Blanch	ydebl@stanford.edu
Steven Dillmann	stevendi@stanford.edu
Simon Obstbaum	simon.obstbaum@obstbaum.de
Santiago Cuellar	scuellar@galois.com
John Sarracino	sarracino2@llnl.gov
Rylan Schaeffer	rschaef@cs.stanford.edu
Mo Tiwari	motiwari@google.com
Donghyun Lee	donghyunlee@berkeley.edu
Bo Han	bhanml@comp.hkbu.edu.hk
Sanmi Koyejo	sanmi@cs.stanford.edu

References

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

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

Amrollahi, D., Karimi, M., Miranda, B., Aniva, L., Sun, C., Barrett, C., and Koyejo, S. AI coding benchmarks need proofs, not just tests. https://cs.stanford.edu/~daneshva/publications/ai-coding-benchmarks-need-proofs-not-just-tests.pdf, 2026. Daneshvar Amrollahi (Stanford); Mahyar Karimi (TU Wien); Brando Miranda, Leni Aniva, Chuyue Sun, Clark Barrett, Sanmi Koyejo (Stanford).

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. Asking the right question: Formal methods as scalable oversight. Author blog post (URL suppressed for blind review), April 2026. Blog post.

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, 2014. 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.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, January 2026. URL https://github.com/harbor-framework/harbor.

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. URL https://arxiv.org/abs/2310.06770.

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. *arXiv preprint arXiv:2305.01210*, 2023. URL https://arxiv.org/abs/2305.01210.

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. https://openai.com/index/introducing-swe-bench-verified/, 2024. Published August 13, 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.

Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z., and Sun, M. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

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.

Rute, J. and Parker, C. What is the most accepted library for floats in lean4? https://proofassistants.stackexchange.com/questions/5311/what-is-the-most-accepted-library-for-floats-in-lean4, 2025. Proof Assistants Stack Exchange discussion of Lean 4 floating-point formalization options.

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*, 2024a.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*, 2024b.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. OpenHands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. R., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

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 Limitations and Discussion (Extended)

This section expands the brief Limitations in main-body Section 6 into the full version, framed as the substantive trade-offs we made and what each one bought us. Each paragraph names a real limitation honestly and the corresponding design pro that motivated it.

Formal methods as scalable oversight (motivating frame). A central concern with LLM-assisted code generation is that models produce text faster than humans can audit, so trust in the output erodes even when the surface looks correct. The concern treats verification as a single unstructured activity: one human eye against a wall of generated tokens. With a proof assistant the activity decomposes. The Lean kernel deterministically checks every step of a proof, so what the human must verify is not the proof but a much smaller artifact—the formal statement, its definitions, and its assumptions—and whether they faithfully express the informal claim. This is the structural move that makes oversight scalable: *the verifier checks the answer; the human checks the question*. VERIBENCH operationalizes this view by separating gold-side curation (*what* to prove, encoded in $\widetilde{Q}_{\mathrm{gold}}$) from agent-side execution (*how*, encoded in $\widetilde{S}_{\mathrm{skill}}$), so the agent contribution is not entangled with curator-set difficulty. Theoretical limits remain—Rice's theorem, the halting problem, and Gödelian incompleteness bound what any countable formal system can decide—but they constrain what the verifier can settle, not the central oversight claim that humans are best deployed on *asking the right question*. The trade-offs named below are the practical price of operating in this regime today, and each is paired with the design pro it bought us. For an extended informal discussion of this view, see (Anonymous, 2026).

Shallow embedding vs. full Python semantics. The gold Lean artifact captures source-level intent (signature, contract, central invariants) rather than re-deriving Python's full operational semantics. *Trade-off pro:* shallow embedding is the regime working formal-methods practitioners actually use, and it is what made an 884-task benchmark possible today—a full-Python-semantics requirement would either reduce VERIBENCH to $\sim$10 tasks or postpone it indefinitely; readers should therefore interpret VERIBENCH scores as "Lean artifact faithful to source-level intent and curator-stated theorems," not as "provably equivalent to the Python operational semantics."

TC_1 as a Pearson-coverage estimate. The deployed TC_1 is dominated by an LLM judge whose held-out Pearson is $r{=}0.7033$ against five independent raters under leakage-safe CV2 task-split isotonic calibration; rank correlations are weaker ($\rho{=}0.32$, $\tau{=}0.27$). *Trade-off pro:* a fully prover-based TC_1 would require cross-implication search to terminate within a tractable budget on most non-trivial pairs, which it does not in our current evaluation. We therefore expose the rigor stack explicitly—kernel cross-implication $\succ$ syntactic match $\succ$ LLM judge—and disclose which mode dominates the reported numbers, rather than collapsing the three into one opaque score. The held-out calibration evidence ($n{=}75$ items, $p{<}10^{-11}$) lets readers calibrate downstream trust to the actual scale-agreement we measured.

Single-run point estimates and the empirical coverage proxy. Headline scores for closed-source agents are single-run evaluations, and we report $\mathrm{IC}_2 \wedge \mathrm{TC}_1$ as an empirical proxy for "agent statement covers gold obligation *and* agent has a sorry-free proof." *Trade-off pro:* the within-agent variance is dominated by judge stochasticity (already characterized via $k{=}3$ judge calls per item, with the deployed item score the repeat mean) rather than agent stochasticity; the Verified-Core sensitivity view ($D_1{=}D_2{=}1$, Appendix I.2) and the near-constant $\widetilde{Q}_{\mathrm{gold}}{\approx}0.72$ across agents indicate that the headline ranking is robust. A stronger empirical proxy lets the agent get coverage credit for theorems that strictly subsume the gold obligation, which a strict-equivalence metric would punish even when the agent statement is logically stronger.

Canonical-core SCSC scope and partly-synthetic VeriBench-IndustrySet. SCSC headline numbers come from the canonical core's five splits, evaluated as 176 Lean evaluation units after multi-formalization; VeriBench-IndustrySet (282 tasks: 106 real-OSS adaptations + 176 domain-prototypical synthetic kernels) is evaluated separately at compile-rate granularity in Appendix I. *Trade-off pro:* the SCSC evaluation set is fixed at release, so the public ranking is reproducible and

not subject to silent expansion. For VeriBench-IndustrySet the synthetic share is determined by license—domains where canonical sources are GPL or proprietary (e.g., KVM/Xen for hypervisors, Linux scheduler internals, Jane Street finance code) get domain-prototypical synthetic kernels, while permissive-license domains (`pyca/cryptography`, OpenAPS `oref0`, NASA F Prime, rust-vmm, cpython stdlib, Cloudflare workerd) get real ports; the released `MANIFEST.csv` per-domain records license, upstream commit SHA, and real/synthetic tag for every task, so the boundary is auditable at the per-task level rather than only per-category.

No human-as-agent ceiling. We do not include a separate "human writing the Lean artifact under the same Harbor harness" baseline. *Trade-off pro:* the gold-validity gates ($D_1 \approx 0.92$, $D_2 \approx 0.57$) already report what careful human curation achieves on the same task pool, exposing the natural ceiling without hiding curator effort behind a perfect-human idealization—an agent below $\widetilde{Q}_{\text{gold}}$ is below careful curators on the same items, not below an unattainable ideal. This framing sharpens, rather than dissolves, the concern motivating human oversight: shallow reading and unjustified confidence remain failure modes, but the target of human attention shifts from hundreds of lines of generated argument to a Lean statement together with its definitions and assumptions—a much smaller object, but the one that actually encodes intent.

Imperative and effectful code: monad-layered verification. A natural reviewer concern is how an end-to-end provable benchmark handles imperative and effectful code (mutation, file I/O, system calls) when those domains naturally involve side effects. Our gold artifacts use a monad-layered approach: imperative code uses Lean's `Id` or `StateM` monad—`do`-notation with `let mut`/`for` desugars to a pure, fully kernel-provable computation, so algorithmic code never needs `IO`. For security and real-code tasks that genuinely require effects, the gold separates provable pure logic (sanitization, parsing, algorithmic kernels) from the `IO` interface, treated as axiomatic at the system boundary—the same trust split used by seL4, CompCert, and CakeML. *Trade-off pro:* this is the standard verified-systems division of labor; pretending to prove OS or hardware behavior would misclassify where the trust actually lives. *Honest caveat:* we prove that the code dispatched to system primitives is well-formed and satisfies its stated invariants; we do not prove that primitive `IO` or system calls themselves behave as documented, and full closed-form composition of agent-written `IO` wrappers with axiomatized system calls is left as a per-task curation choice rather than a benchmark-wide guarantee.

C 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; a future benchmark-validity block could handle this symmetrically by discounting 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. (2024a) 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 a structured Lean artifact (code, specification, and docstring) by choosing a formal model for 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 acceptance structure used in formal code verification.

Module family	Functions / kernels	Adaptation type
`bisect.py`	`bisect_left/right`, `insort_left/right`	Simplified single-function adaptation
`heapq.py`	`heappush`, `heappop`	Simplified single-function adaptation
`queue.py`	simple queue, thread-safe queue kernels	Distilled invariant kernel
`collections`	`Counter`, `ChainMap`	Simplified interface adaptation
`statistics.py`	mean, median, variance, correlation, covariance, quantiles	Simplified numeric adaptation
`fractions.py`	fraction normalization/arithmetic kernel	Distilled invariant kernel
`functools.py`	`reduce`, `lru_cache`, `partial`	Simplified interface adaptation
`textwrap.py`	`indent`, `dedent`	Simplified string adaptation
`html.py`, `shlex.py`	escape, quote, join	Simplified string adaptation
`base64.py`, `json`, `urllib.parse`, `secrets.py`, `graphlib.py`	base32, decoder, parse, token, topological-sort kernels	Distilled contract kernel

Table 3: RealCodeSet provenance. All 32 items are simplified or distilled adaptations used to preserve the source-level contract and central invariant under Lean-friendly types; none is claimed to be a verbatim CPython port.

D 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.

D.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-sensitive 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 simplified single-function adaptations of Python standard-library routines, 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.

Domain selection. The 14 domain families span the established *high-assurance software* target areas in the formal-methods literature—cryptography, OS / hardware / compilers, hypervisors, real-time and aerospace control—and the emerging high-stakes ML-adjacent areas: medical software,

Expansion domain	Real	Synthetic	Total
01_crypto	16	4	20
02_rfc	19	1	20
03_memory_bugs	5	15	20
04_aerospace	3	17	20
05_regulated	5	15	20
06_medical_oss	12	8	20
07_finance	8	12	20
08_os_embedded	7	13	20
09_hardware	5	16	21
10_concurrent	5	15	20
11_hypervisor	5	16	21
12_compilers	5	15	20
13_gov_cert	5	15	20
14_critical_infra	6	14	20
IndustrySet total	**106**	**176**	**282**

Table 4: **VeriBench-IndustrySet** composition. Real tasks are license-checked adaptations from permissively licensed or public-domain sources; synthetic tasks are domain-prototypical kernels designed to exercise representative verification patterns.

regulated automotive/medical control, financial settlement, and government certification. The real-vs-synthetic mix in Table 4 reflects what is available under permissive licenses: where domain-canonical open-source software is licensed permissively (e.g., NASA/JPL F Prime, OpenAPS, rust-vmm, Spike/RISC-V), we favor real ports; where canonical sources are GPL or proprietary (e.g., KVM/Xen, Linux scheduler internals), we use domain-prototypical synthetic kernels designed to exercise representative verification patterns under the same VeriBench schema.

Per-domain verification flavor. Each domain probes a different cluster of verification properties:

- `01_crypto` — HMAC, HKDF, AES-GCM nonce-counter freshness, PKCS#7 padding, Fernet header/TTL guards, ERC-20 transfer/allowance arithmetic, Schnorr response, and Merkle pair canonicalization.
- `02_rfc` — Internet-RFC parsing and protocol-state-machine conformance (HTTP status classes, real CPython-sourced RFC kernels).
- `03_memory_bugs` — Bounds, aliasing, use-after-free, and double-free freedom; CPython-derived real ports plus synthetic memory-safety bookkeeping kernels.
- `04_aerospace` — Real-time control invariants from Apache-2.0 F Prime ports plus synthetic aerospace kernels (deadlines, redundancy, bounded numerical drift).
- `05_regulated` — FDA-/FAA-/ISO-style safety pre/post-conditions for regulated medical, automotive, and aviation software.
- `06_medical_oss` — OpenAPS-derived insulin-control kernels: dosage bounds, safety interlocks, dose-rate envelopes.
- `07_finance` — Order matching, settlement, margin, FIFO inventory, transfer, spread, tick, slippage, Kelly cap, and double-spend invariants.
- `08_os_embedded` — Scheduler liveness, interrupt handlers, and embedded-OS memory-safety kernels.
- `09_hardware` — Spike/RISC-V decoder ports and synthetic RTL-style kernels (instruction-decode invariants, memory-ordering proxies).
- `10_concurrent` — Race-freedom, deadlock-freedom, and atomicity invariants for concurrent kernels.
- `11_hypervisor` — rust-vmm `vm-memory`, `seccompiler`, and `vm-virtio` ports plus synthetic guest-isolation kernels.
- `12_compilers` — CPython-derived compiler kernels and toy-IR semantics-preservation tasks.
- `13_gov_cert` — Apache-2.0 government-certification kernels (Common-Criteria-style refinement and access-control patterns).

- `14_critical_infra` — Fail-safe defaults and fault-tolerance bounds for critical-infrastructure control kernels.

The `REPORT.md` per-domain notes released alongside the artifact list the specific real-source provenance (license, upstream commit, files), so the boundary between real and synthetic in each row of Table 4 is auditable.

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.

D.2 Curation Rubric and Provenance

Reviewer roles. Task curation was performed by paper authors and close collaborators with Lean 4, programming-languages, or formal-methods experience. For anonymized review we describe qualifications by role rather than name: each accepted task has an issue-opening curator and a second reviewer, and the second reviewer is not the same person who opened the issue.

Written review checklist (shared by AI and human reviewers). The same written rubric—the gold-reference template (`my_add.lean` canonical example) plus the review checklist below—is used both as the prompt to AI drafting agents (`o3` and curation-time tools) and as the explicit checklist for human reviewers, so the same standard governs every artifact regardless of who wrote the patch. The second human reviewer checks that the Python input runs, the Lean file follows the benchmark schema, theorem statements characterize behavior rather than merely restating the implementation, pre/post-conditions expose meaningful admissibility and semantic claims, proof placeholders are explicit `sorry`/`admit` occurrences counted by D_2, and positive/negative/edge tests agree between the Python and Lean sides. The reviewer may edit the patch directly or request changes in the issue/PR thread; *the final merge to `main` is always performed by a human reviewer* after open correctness concerns are resolved. The released artifact ships the rubric prompts alongside the gold examples (`veribench_dataset/instruction_creating_examples_and_curating.md` for all splits, plus `veribench_dataset/instruction_curating_security_examples.md` for the security-specific extensions) so the curation guidance is auditable. The general rubric prefaces itself with the policy *"Automation (Opus critic, LLMs) can flag issues and propose fixes, but a human must read the final file and agree before it enters the dataset. Every data point defines 'correctness' for a benchmark—the final inclusion decision must be human."*

Substantive edits. We classify a curation edit as substantive when it changes a type, implementation, theorem statement, pre/post-condition, proof strategy, or test oracle. Formatting-only edits, comments, and typo fixes are not substantive. The repository history records these changes at per-task granularity, but the current paper does not use an aggregate "substantive rewrite fraction" as an experimental result because that fraction has not yet been computed with a stable script.

External audit status. The current release uses internal two-reviewer curation plus the independent human-rater study for TC_1 calibration. It does not claim that an external formal-methods reviewer audited every gold task before submission; D_1, D_2, the float/partial audit, and the released provenance are reported so that remaining benchmark-validity work is visible.

D.3 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.

D.4 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.

D.5 Parallel Translations Across Additional Languages

Scope and intent. While the headline VERIBENCH evaluation in this paper is Python-to-Lean 4 autoformalization, the released dataset additionally ships parallel artifacts of the canonical-core splits (HumanEval, EasySet, CSSet, RealCodeSet, SecuritySet) in five additional source languages—C, C++, Haskell, OCaml, and Rust—and in Rocq (Coq) as an alternative formal target alongside the primary Lean 4 target. VeriBench-IndustrySet is not covered by these parallel translations. We include these artifacts to support two extensions that we leave to future work: (i) studying autoformalization across non-Python source languages, and (ii) studying agent behavior under alternative target proof assistants.

Source-language reference implementations. The C, C++, Haskell, OCaml, and Rust translations are executable reference implementations with assertion-based unit tests, mirroring the role of the Python reference files described above (Appendix D). They are *not* formal verifications: they do not include theorem statements, pre/post-condition predicates, or proof obligations. Coverage is consistent on the core programming splits—56 HumanEval, 41 EasySet, and 13 CSSet tasks per language—and partial on RealCodeSet (16–32 tasks per language) and SecuritySet (9–460 tasks per language; the C SecuritySet is substantially larger because the underlying MIT 6.858 source ships multiple variants per task). We did not run an SCSC-style evaluation against these source-language references for this paper; doing so requires committing to a target proof assistant for each source language and is left to follow-up work.

Rocq (Coq) target. Rocq is provided as an alternative formal target on a subset of the core problems: 56 HumanEval, 41 EasySet, 13 CSSet, 16 RealCodeSet, and 8 SecuritySet tasks. We include this partial Rocq formalization to demonstrate that the gold-file structure used for Lean 4 (implementation, tests, properties, post-condition, correctness theorem, optional imperative implementation with equivalence) ports to a second proof assistant without redesign, and to enable cross-prover comparisons in future work. Rocq evaluations are *not* included in our headline SCSC results, and we do not claim equivalent task coverage or evaluation depth between Lean 4 and Rocq.

Provenance and limitations. The multi-language artifacts were drafted with LLM assistance and human-curated, following the same two-stage process as the Lean 4 gold files. Non-Lean implementations have been compiled and exercised against their assertion-based tests but have not been independently audited at the same depth as the Lean 4 gold reference. We therefore present them as auxiliary research artifacts that broaden the dataset's reach, rather than as additional primary evaluation targets in this paper.

D.6 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.

D.7 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. Gold Lean files are expected to compile and discharge their proofs; current gaps are reported through D_1, D_2, and the oracle row rather than hidden. When a gold file compiles, the Lean kernel establishes that the implementation, tests, theorem statements, and any proof bodies without `sorry` 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. Compilation checks that implementations, unit tests, theorem statements, and proof terms are well-typed; when a theorem body contains no `sorry` or `admit`, the proof term is kernel checked. 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.8 Artifact Access and Smoke Test

The anonymous supplemental artifact contains the benchmark data, Lean project, evaluation scripts, prompts, and v3 result files used for the paper tables. The expected smoke test is:

```
cd veribench
bash veribench_setup.sh
cd veribench_dataset/lean_src
lake build
```

The full agent evaluation is run through Harbor with one Docker container per task; the main result tables are generated from `experiments/38_scsc_full_metric_evals/expt_v3/results/`. Closed-model agent runs require the corresponding vendor CLI/API credentials, but the benchmark data, gold Lean files, metric scripts, and stored result summaries can be inspected without those credentials. For the anonymous submission, public repository URLs are omitted or anonymized. The artifact bundle is the submission-time reproduction package; permanent repository and metadata links are intentionally omitted for anonymity.

D.9 Representative Benchmark Examples

To make the task structure concrete, we include one simplified example per canonical-core split. Each example pairs an executable Python reference (with docstring + tests) with the corresponding gold Lean 4 artifact (functional implementation, unit tests, pre-condition, property theorems, post-condition, correctness theorem, and—when applicable—imperative implementation with equivalence theorem). Full-length variants are released alongside the artifact.

Algorithm 1 An exemplar input Python code of VeriBench-EasySet (simplified).

```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, f"expected 7 from (7,3) but got {candidate(7, 3)}"
    # ... (other tests) ...
    return True

if __name__ == "__main__":
    assert check(my_max), f"Failed: {__file__}"
    print(f'All tests passed: {__file__}!')
```

Algorithm 2 An exemplar golden output Lean 4 code of VeriBench-EasySet (simplified).

```
namespace MyMax

-- Implementation
def myMax (a b : Nat) : Nat :=
  if _ : a \le b then b else a

infixl:70 " \sqcup " => myMax    -- left-associative, precedence 70

-- 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
```

Algorithm 3 An exemplar input Python code of VeriBench-CSSet (simplified for showcase).

```python
# Implementation
from typing import List, Optional
def binary_search(arr: List[int], target: int) -> Optional[int]:
    """
    Binary search implementation that searches for a target value in a sorted list.
    Returns the index if found, None if not found.
    """
    if not arr:
        return None

    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        mid_val = arr[mid]

        if mid_val == target:
            return mid
        elif mid_val < target:
            left = mid + 1
        else:
            right = mid - 1

    return None

# Tests
from typing import Callable
def check(candidate: Callable[[List[int], int], Optional[int]]) -> bool:
    assert candidate([1, 2, 3, 4, 5], 1) == 0
    assert candidate([1, 2, 3, 4, 5], 3) == 2
    # ... (more tests) ...

    print("Pass: all correct!")
    return True

if __name__ == "__main__":
    assert check(binary_search), f"Failed: {__file__}"
```

Algorithm 4 An exemplar golden output Lean 4 code of VeriBench-CSSet (simplified for showcase).

```
import Mathlib.Data.List.Sort
import Mathlib.Data.List.Basic

namespace BinarySearch
open List

-- Implementation
partial def binarySearchAux (arr : List Nat) (target : Nat) (left right : Nat) :
    Option Nat :=
  if left > right then
    none
  else
    let mid := (left + right) / 2
    if mid >= arr.length then
      none
    else
      let midVal := arr.get ⟨mid, by sorry⟩
      if midVal = target then
        some mid
      else if midVal < target then
        binarySearchAux arr target (mid + 1) right
      else
        binarySearchAux arr target left (mid - 1)

def binarySearch (arr : List Nat) (target : Nat) : Option Nat :=
  if arr.isEmpty then
    none
  else
    binarySearchAux arr target 0 (arr.length - 1)

/-- Linear search for comparison and verification -/
def linearSearch (arr : List Nat) (target : Nat) : Option Nat :=
  arr.findIdx? (· = target)

-- Theorem: If binarySearch returns Some i, then arr[i] = target
theorem correctness_binarySearch (arr : List Nat) (target : Nat) (i : Nat) :
  binarySearch arr target = some i → arr[i]? = some target := by
  sorry

-- Theorem: If target is in the sorted array, then binarySearch finds it
theorem completeness_binarySearch (arr : List Nat) (target : Nat) :
  List.Sorted (fun x y => x \le y) arr → target \in arr →
  \exists i, binarySearch arr target = some i := by
  sorry

-- ... (more theorems) ...

end BinarySearch
```

Algorithm 5 An exemplar input Python code of VeriBench-HumanEval.

```python
# -- Implementation --
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, any two numbers are closer
    to each other than the given threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False

# -- Tests --
from typing import Callable
def check(candidate: Callable[[List[float], float], bool]) -> bool:
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    return True

if __name__ == "__main__":
    assert check(has_close_elements), f"Failed: {__file__}"
```

Algorithm 6 An exemplar golden output Lean 4 code of VeriBench-HumanEval.

```lean
namespace HasCloseElements
open List

def hasCloseElements (numbers : List Float) (threshold : Float) : Bool :=
  match numbers with
  | []      => false
  | x :: xs =>
    if xs.any (fun y => Float.abs (x - y) < threshold) then
      true
    else
      hasCloseElements xs threshold

-- Theorems
/-- Monotone in threshold: enlarging the tolerance preserves truth. -/
@[simp] theorem threshold_mono
    {numbers : List Float} {t_1 t_2 : Float}
    (hle : t_1 \le t_2)
    (h : hasCloseElements numbers t_1 = true) :
  hasCloseElements numbers t_2 = true := by
  sorry

end HasCloseElements
```

Algorithm 7: An exemplar input Python code of VeriBench-SecuritySet.

```python
# Implementation
def unsafe_copy(dst: bytearray, src: bytearray) -> None:
    """
    Copy bytes from `src` into `dst` at the same indices, without any bounds checking.
    If `len(src) > len(dst)`, this will raise an IndexError (buffer overflow).
    """
    for i, b in enumerate(src):
        dst[i] = b

def check(candidate) -> bool:
    # 1) Safe copy: src fits in dst
    d = bytearray(3)
    s = bytearray(b'abc')
    candidate(d, s)
    assert bytes(d) == b'abc'

    # ... (other tests) ...
    return True

assert check(unsafe_copy), "Candidate failed buffer-overflow tests"
print("Pass!")
```

Algorithm 8: An exemplar golden output Lean 4 code of VeriBench-SecuritySet.

```lean
namespace BufferOverflow

/--
`unsafeCopy dst src` attempts to overwrite the first `src.length` bytes of `dst`
with those from `src`.
Returns `some newDst` if `src.length \le dst.length`,
otherwise `none`, modeling a buffer overflow.
-/
def unsafeCopy (dst src : List UInt8) : Option (List UInt8) :=
  let n := dst.length
  src.enum.foldl (fun o (i, b) =>
    o.bind fun acc =>
    if h : i < n then some (acc.set i b) else none
  ) (some dst)

-- Tests
example : unsafeCopy [0, 0, 0] [1,2] = some [1,2,0] := by rfl
example : unsafeCopy [0, 0] [1,2,3] = none := by rfl

-- Theorem: safety precondition
theorem copy_safe {dst src : List UInt8} (h : src.length \le dst.length) :
  \exists newDst, unsafeCopy dst src = some newDst := by
  unfold unsafeCopy
  admit

end BufferOverflow
```

Algorithm 9 An exemplar input Python code of VeriBench-RealCode.

```python
# -- Implementation --
# source: https://github.com/python/cpython/blob/3.13/Lib/heapq.py

def _siftdown(heap, startpos, pos):
    newitem = heap[pos]
    while pos > startpos:
        parentpos = (pos - 1) >> 1
        parent = heap[parentpos]
        if newitem < parent:
            heap[pos] = parent
            pos = parentpos
            continue
        break
    heap[pos] = newitem

def heappush(heap, item):
    """Push item onto heap, maintaining the heap invariant."""
    heap.append(item)
    _siftdown(heap, 0, len(heap) - 1)

# -- Tests --
from typing import Callable
import random

def check(candidate: Callable[[list, int], None]) -> bool:
    # Basic unit tests
    h = []
    candidate(h, 3)
    # ... (tests omitted for brevity) ...
    return True

if __name__ == "__main__":
    assert check(heappush), f"Failed: {__file__}"
```

Algorithm 10 An exemplar golden output Lean 4 code of VeriBench-RealCode.

```
namespace HeapPush

/-- `_siftdown`: Restores the heap invariant by moving parents down. -/
def _siftdown (heap : Array Int) (startpos pos : Nat) : Array Int :=
  let newitem := heap[pos]!
  let rec loop (h : Array Int) (pos : Nat) : Array Int :=
    if pos > startpos then
      let parentpos := (pos - 1) >>> 1
      let parent := h[parentpos]!
      if newitem < parent then
        loop (h.set! pos parent) parentpos
      else
        h.set! pos newitem
    else
      h.set! pos newitem
  loop heap pos

def heappush (heap : Array Int) (item : Int) : Array Int :=
  let h := heap.push item
  _siftdown h 0 (h.size - 1)

-- Theorems
/-- Invariant theorem: `heappush` preserves the heap invariant. -/
@[simp] theorem invariant_thm (heap : Array Int) (item : Int) (hPre : Pre heap) :
  prop_invariant heap item := by sorry

end HeapPush
```

D.10 VeriBench-IndustrySet: Taxonomy Provenance and Real-vs-Synthetic Methodology

Taxonomy provenance. The 14 domain families in **VeriBench-IndustrySet** are derived from a taxonomy of *"what code do people actually want formally verified"* elicited from formal-methods experts and industry-adjacent practitioners during the benchmark's design phase. The taxonomy organizes verification targets by domain-of-impact and by the distinct invariant families each domain probes (e.g., constant-time arithmetic for cryptography, message-format conformance for RFC parsing, bounded-numerical-drift for aerospace, dose-rate envelopes for medical control, race/deadlock-freedom for concurrent systems, semantics-preservation for compilers, capability isolation for hypervisors). Names of consultants are anonymized for blind review; the taxonomy itself is included verbatim in the released artifact.

Real-vs-synthetic methodology. For each of the 14 domains we apply a fixed two-stage construction:

1. **License-verified real-OSS hunt.** For each category, we spend a fixed budget on a `web_fetch`-driven search for permissively licensed open-source software (MIT, Apache-2.0, BSD, ISC, PSF, public-domain, NOSA, 0BSD, Unlicense, CC0). Each candidate's license is verified before adaptation; AGPL and proprietary sources are explicitly rejected.
2. **Domain-prototypical synthetic kernels.** Where canonical sources are GPL or proprietary (e.g., KVM/Xen for hypervisors, Linux scheduler internals, Jane Street finance code), we author synthetic kernels designed to exercise representative verification patterns under the same VeriBench schema.

The real/synthetic split per category is therefore *whatever the license-verified hunt yields*, not a quota. This produces honest provenance (the released `MANIFEST.csv` per domain records license, upstream commit SHA, file hashes, and source URL for every real port; synthetic items are tagged explicitly).

Category	Prior lean	Representative real OSS / source
01_crypto	Real-heavy	pyca/cryptography, PyNaCl, pycryptodome (MIT/BSD); HACL* primitives; ERC-20 ports
02_rfc	Real-heavy	cpython stdlib (`urllib.parse`, `ipaddress`, `email.parser`, `http.cookies`); h11
03_memory_bugs	Synth-heavy	C/C++ memory bugs don't port cleanly; synthesize bytearray/ring-buffer/refcount kernels
04_aerospace	Hybrid	Apache-2.0 NASA F Prime ports, poliastro (MIT) for orbital math
05_regulated	Hybrid	FDA/FAA/ISO-style synthetic compliance kernels; some real OpenAPS-adjacent dose math
06_medical_oss	Real-heavy	OpenAPS `oref0` (MIT) IOB / basal / bolus pure functions; Loop Swift→Python ports
07_finance	Hybrid	QuantLib (BSD), open matching engines, ta-lib; synthetic order-book/settlement/margin
08_os_embedded	Hybrid	Zephyr (Apache) + FreeRTOS (MIT) ring-buffer/scheduler kernels; seL4 capability synth
09_hardware	Synth-heavy	Ibex/CVA6 RISC-V (Apache) decoder ports; CRC/ALU/FIFO synthetic kernels
10_concurrent	Hybrid	Lock-free/CRDT primitives; linearizability/ABA-safety synthetic invariants
11_hypervisor	Synth-heavy	rust-vmm `vm-memory`/`seccompiler`/`vm-virtio` (Apache); seL4-style synth
12_compilers	Hybrid	CompCert→Python ports for constant-folding/dead-code-elim; cpython parser fragments
13_gov_cert	Synth-heavy	NASA-CR-2014-218244 numerical excerpts; ACL2 verified arithmetic ports; audit-log synth
14_critical_infra	Hybrid	Cloudflare workerd/quiche/pingora (Rust→Python ports); rate-limit/circuit-breaker synth

Table 5: Per-category priors on whether each domain leans real OSS, synthetic kernels, or hybrid, and representative real-source candidates the hunt targets first. These are *priors*, not quotas; the realized real/synthetic split (Table 4) is determined by what the license-verified hunt actually yields.

Per-category real-vs-synthetic priors. Table 5 summarizes our *prior expectation* for each category before the hunt; the actual real/synthetic counts in Table 4 are determined post-hoc by what the licensed hunt yields.

Auditable provenance. Each of the 14 domain subdirectories in the released artifact ships with a `MANIFEST.csv` (one row per task: license, upstream URL, commit SHA, file SHA-256, line counts, real/synthetic tag, comprehensiveness note) and a `REPORT.md` (TLDR + per-task curation notes). This makes the boundary between real and synthetic auditable per-task rather than only per-category.

D.11 Broader Impacts (Extended)

Positive societal impact. Stronger formal-verification benchmarks help shift AI-coding evaluation away from test-only correctness and toward checked specifications. Our shallow-embedding setup makes formal verification accessible without a full Python semantics, and our public theorem-coverage judge calibration (Appendix G) lets downstream users build on our calibration data rather than re-litigate it. The released gold-validity gates (D_1, D_2) make the boundary between agent skill and benchmark-item maturity visible, reducing the risk that improved scores are misread as fully verified artifacts.

Negative societal risks and mitigations. The main risks are (i) over-trust: users reading high SCSC values as a deployment guarantee, (ii) evaluation-driven over-fitting: agents stating only easy obligations to inflate IC_2 at the expense of TC_1, and (iii) misuse of the security examples. We mitigate (i) by explicitly reporting factor-level failures, sorry-counts, and judge-calibration limits (held-out $r = 0.7033$; zero-shot $r = 0.5154$); we mitigate (ii) by separating coverage and proof factors and reporting both; we mitigate (iii) by framing the security split as benchmark tasks (gold theorems characterizing the safe behavior of vulnerable code patterns), not as exploit-generation prompts. The release contains benchmark tasks and evaluation code, not model weights or any privileged inputs.

E 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.

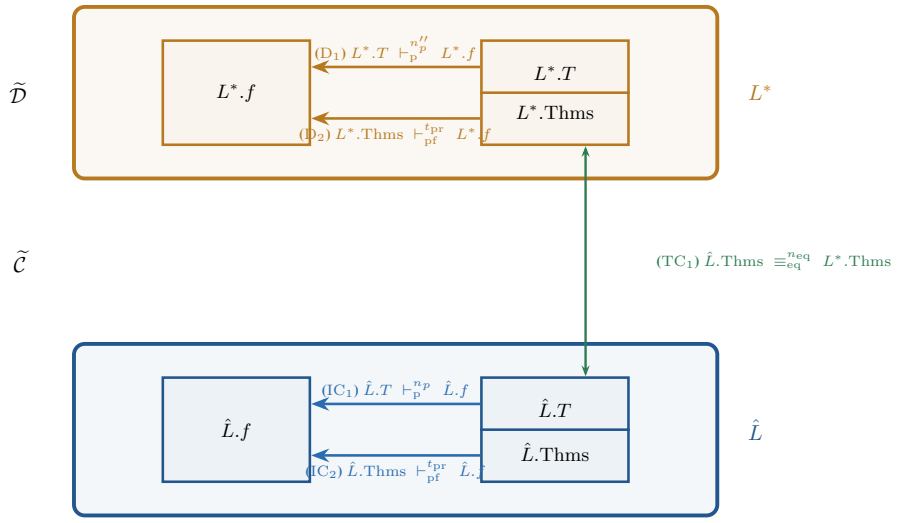
The hypercube volume intuition. The five soft truth values $\{f_i\}_{i=1}^5$ define the edge lengths of a 5-dimensional box with volume $\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}} = \prod f_i$. $\widetilde{S} = \left(\widetilde{\mathcal{C}} \cdot \widetilde{\mathcal{D}}\right)^{1/5}$ is the 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.



$\widetilde{\mathcal{C}}$ **Correctness:** **Internal-Consistency (IC)** ∧ **Theorem-Cov (TC)**

$\widetilde{\mathcal{D}}$ **Data Quality:** **Gold-Lean-Correctness**

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

VERIBENCH *evaluation topology. Each arrow is one multiplicative factor in* $\widetilde{S}$*.*
Double arrow ($\leftrightarrow$) *= coverage* ($\equiv_{\text{eq}}$)*; single arrows* ($\rightarrow$) *= property satisfaction* ($\vdash$)*.*

Figure 2: **VERIBENCH evaluation topology.** Each solid arrow is one multiplicative factor in the five-factor evaluation-chain score $\widetilde{S}_5 = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1 \cdot D_1 \cdot D_2)^{1/5}$. **Blue arrows (IC)**: agent's tests and theorems evaluated against agent's own code ($\hat{L}$, blue box) — the agent-side internal-consistency factors IC_1, IC_2. **Green double-arrow (TC)**: agent's theorems compared against the gold theorems $L^*.\text{Thms}$ (gold box) — the agent-side coverage factor TC_1. **Gold arrows ($\mathcal{D}$)**: gold tests and theorems evaluated against the gold code L^* — the gold-side benchmark-validity gates D_1, D_2. Agent-skill score $\widetilde{S}_{\text{skill}} = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1)^{1/3}$ uses only the blue and green arrows; gold-side quality $\widetilde{Q}_{\text{gold}} = (D_1 \cdot D_2)^{1/2}$ uses only the gold arrows.

F Comprehensive SCSC Results

Table 6 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.

Why the scorer does not cross-apply gold artifacts. Early SCSC drafts included factors that applied gold tests and theorem statements directly to the agent implementation, as well as a Jaccard-style test-equivalence factor between gold and agent test suites. We removed these factors because independently generated Lean files often use different types, namespaces, currying conventions, or data representations. Directly importing a gold theorem over one representation and applying it to an agent implementation over another therefore produces type errors rather than evidence of semantic failure. Test-suite Jaccard overlap has the same problem: two valid test suites need not share literal tests. The released metric instead combines IC_2, which asks whether the agent proves its own

Agent	Model	IC_1	IC_2	TC_1	D_1	D_2	$\widetilde{S}$	n
Frontier agents (2026 models)								
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 6: 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; IC_2 is the fraction of agent theorems without `sorry`; TC_1 is the LLM-judge theorem coverage between agent and gold theorems under the same 0–5 coverage rubric calibrated in Appendix G, normalised to $[0, 1]$; D_1, D_2 are gold-side benchmark-validity gates for gold tests and gold proofs. Runs without completed SCSC summaries are omitted. n = number of tasks with agent output. Source: `experiments/38_scsc_full_metric_evals/expt_v3/results/`.

Agent	$\widetilde{S}_{\mathrm{skill}}$	$\widetilde{Q}_{\mathrm{gold}}$	$\widetilde{S}_{\mathrm{cond}}$	$\widetilde{S}_{\mathrm{full}}$
Codex (GPT-5.4)	0.289	0.724	0.417	0.391
Claude Code (Sonnet 4.6)	0.224	0.724	0.358	0.335
Baseline (Sonnet 4.6)	0.210	0.723	0.344	0.328
Leanstral (v2)	0.103	0.723	0.225	0.215

Table 7: Sensitivity view separating agent-side skill from gold-side benchmark validity for the main agents. $\widetilde{S}_{\mathrm{skill}} = (\mathrm{IC}_1\mathrm{IC}_2\mathrm{TC}_1)^{1/3}$, $\widetilde{Q}_{\mathrm{gold}} = (D_1 D_2)^{1/2}$, $\widetilde{S}_{\mathrm{cond}}$ is the headline conditional SCSC, and $\widetilde{S}_{\mathrm{full}}$ scores no-output tasks as zero on $N = 176$.

theorem statements, with TC_1, which asks whether those statements cover the gold obligations. This is an empirical semantic-coverage proxy, not a kernel proof that gold theorems hold of the agent's code; this limitation is why the TC_1 judge is calibrated against humans and reported separately.

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.

- TC_1 **is consistently low.** 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 structured Lean artifact (implementation + tests + theorems) that compiles end-to-end.

F.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

$$\mathrm{IC}_2(\hat{L}) \;=\; \begin{cases} \dfrac{\#\{\text{closed theorems in } \hat{L}\}}{|\mathrm{Thms}(\hat{L})|}, & \text{if } \hat{L} \text{ compiles under } \texttt{lake build} \text{ and } |\mathrm{Thms}(\hat{L})| > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

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 file typechecks.** Let $\hat{L}$ denote the single Lean file extracted from the agent transcript. Then $\mathrm{IC}_1(\hat{L}) = 1$ when $\hat{L}$ resolves imports and is accepted by Lean as a well-typed file under the evaluation toolchain, and $\mathrm{IC}_1(\hat{L}) = 0$ otherwise.

D_1, D_2 **— gold-side validity gates.** D_1 measures whether the gold tests pass on the gold implementation, and D_2 applies the same sorry-free theorem-closure rule as IC_2 to the gold theorem set. 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 gold-side gate values $\mathrm{D}_1 = 0.898$, $\mathrm{D}_2 = 0.604$ summarize the current gold-set typecheck/proof maturity.

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 overcount 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.2 Threats to Validity (Detailed)

We expand the five threats named briefly in Section 4.

(1) Judge dominance. Prover and structural modes close only a small fraction of pairs within budget, so mode 3 effectively determines the bulk of reported TC_1 values; we calibrate the prompt against five human raters (held-out task-split Pearson $r = 0.7033$ under leakage-safe isotonic calibration; zero-shot $r = 0.5154$; both moderate by convention, with rank correlations weaker; per-split correlations and inter-rater reliability in Appendix G), but cross-model judge variance is uncontrolled.

(2) Single-run point estimates. Closed-source agent runs are not multi-seeded due to compute cost, so the noise in $\widetilde{S}$ is judge-stochasticity-dominated rather than agent-stochasticity-controlled; deterministic factors (IC_1, IC_2, D_1, D_2) are reproducible exactly.

(3) Coverage proxy. $\mathrm{IC}_2 \wedge \mathrm{TC}_1$ is an empirical proxy for cross-proves correctness, not a kernel-checked transitivity proof.

(4) No human-as-agent ceiling. The human-judge study (Appendix G) calibrates the LLM judge on *rating* agent outputs, not on human performance at *generating* them; we do not report a human-author baseline.

(5) Gold comprehensiveness and Float opacity. Uncovered behaviors are mitigated by the D_1/D_2 gates and the Verified-Core sensitivity in Appendix F; universal `Float` properties enter via the multi-layer convention (Section 3; Appendix J).

F.3 TC_1 Coverage Modes (Detailed)

The coverage check $\hat{L}.\mathrm{Thms} \equiv_{\mathrm{eq}}^{n_{\mathrm{eq}}} L^*.\mathrm{Thms}$ runs in one of three modes, in order of decreasing rigor; 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 whose held-out task-split agreement with five independent human raters reaches Pearson $r = 0.7033$ under leakage-safe isotonic calibration (zero-shot $r = 0.5154$; Appendix G). This mode produces an empirical proxy for coverage, not a formal proof of entailment.

Formal cross-implication (mode 1) times out for most non-trivial pairs in our current evaluation, so the LLM-judge mode (mode 3) effectively determines the bulk of reported TC_1 values. This is why we report the calibration framing (held-out + leakage-safe + zero-shot) explicitly at every TC_1 mention in the body.

F.4 Extended Discussion (Gold Validity, Judge Validity, Reproducibility, Scope)

Gold validity and judge validity. The released gold set still has imperfect typecheck, test, and proof-validity gates (especially incomplete proofs), exposed through D_1, D_2 rather than hidden; the headline ranking therefore uses $\widetilde{S}_{\mathrm{skill}}$ (which depends only on agent output), with $\widetilde{Q}_{\mathrm{gold}}$ reported separately. TC_1 is a human-calibrated semantic coverage estimate, not a kernel-checked equivalence proof, whenever the LLM-judge fallback is used.

Reproducibility and scope. The main results are single-run evaluations of expensive closed-model agents, so they should be interpreted as point estimates rather than variance-controlled training curves. We report confidence intervals and p-values for the human–judge alignment study, but not multi-seed uncertainty for every agent run. The benchmark spans 884 tasks (602-task canonical core plus VeriBench-IndustrySet's 282-task industry-elicited high-assurance set), but it does not claim to cover arbitrary Python, C, or systems-code semantics. The artifact release, Docker harness, Lean toolchain

pin, and smoke-test commands in Appendix D.8 are intended to make the benchmark executable and auditable.

G 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 $(\texttt{gold\_file}, \texttt{candidate\_file})$ pair with a human label. The calibration pool contains 25 tasks (75 item keys) from the human-label collection used in Appendix H. 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. This calibration protocol is task-disjoint within the 25-task pool; it is reported separately from the canonical-core SCSC benchmark ranking.

Judge/model/prompt setup. Important: deployed judge $\neq$ calibration judge. The deployed v3 TC_1 values reported in Section 5 (Table 2) are produced by a `claude-sonnet-4-6` LLM judge: per item, k=3 independent calls produce 0–10 theorem-equivalence ratings whose median is normalized to $[0, 1]$ (see `experiments/38_scsc_full_metric_evals/expt_v3/run_real_eval.py`). The calibration analyses in this appendix use a *separate*, independent `gpt-5.3-codex` judge under a 0–5 theorem-coverage rubric (`Human Rating Prompt`; `P4_CHECKLIST` also reported); the held-out r=0.7033 result calibrates the rubric-prompted GPT judge against humans, not the deployed Sonnet judge directly. We treat the GPT-rubric calibration as post-hoc validation evidence that LLM-as-coverage-judge tracks human labels in this regime; we do not claim it transfers numerically to the deployed Sonnet judge. Calibrating the deployed Sonnet judge directly to humans on the same items is left to future work. A practical confound: the deployed Sonnet judge is the same model family as one of the evaluated agents (Claude Code, Sonnet 4.6); Sonnet judging Sonnet outputs is a self-evaluation effect we cannot rule out. 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.

Why these small p-values occur. The null tested for Pearson is $H_0 : r = 0$, so with held-out $r = 0.7033$ at $n = 75$, the test statistic

$$t = r\sqrt{\frac{n-2}{1-r^2}}$$

is large ($t \approx 8.5$, $df = 73$), which yields a two-sided p-value on the order of 10^{-11}, consistent with Table 9. Thus, the small p-value is a consequence of strong nonzero linear association, not by itself evidence of perfect agreement; this is why we also report rank correlations and confidence intervals.

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 8: **Human–judge alignment under task-level held-out evaluation for calibration runs.** Zero-shot row is raw judge score; isotonic rows are post-hoc monotone remappings fit on train tasks only. 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 9: **p-values for Table 8.**

The TC1 judge is calibrated against five independent human raters, reaching held-out Pearson $r = 0.7033$ under leakage-safe task-level isotonic remapping; raw zero-shot alignment is moderate ($r = 0.5154$), and rank correlations remain lower.

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. The negative result is preserved: rank correlations do not uniformly improve under calibration (e.g., CV2 isotonic $\rho = 0.3211$, $\tau = 0.2651$). Therefore, gains are mostly realized for Pearson correlation.

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:

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

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

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

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).

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 10: **95% confidence intervals for correlation estimates.** These intervals quantify uncertainty in the estimated correlations, not run-to-run judge score variation.

```
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.

H Human Judgement Collection

This appendix documents how human labels were collected, grouped, and aggregated for the analyses in Appendix G. 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

and were instructed not to use LLMs or automated judges during labeling. 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 G.

H.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`.

Collection constraints. Raters are instructed to work independently and not use LLMs or automated judges during labeling. 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 raters were expert/collaborator volunteers rather than paid crowdsourcing workers, and the study did not collect personal or sensitive information.

H.2 Distribution of Human Scores

Distribution differences and relevance. Figure 3 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 11 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 12 reports mean and median dispersion.

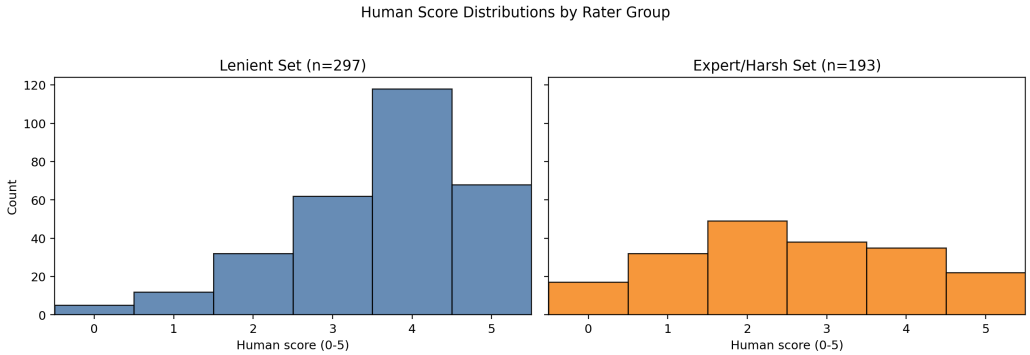


Figure 3: Human score distributions (0–5): lenient set ($n = 297$, left) and expert/harsh set ($n = 193$, right).

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 11: Cross-set disagreement between the lenient and expert/harsh grouped human label sets on shared examples. Item-level metrics are computed on overlapping (`gold_file`, `candidate_file`) keys ($n = 75$); task-level metrics first average scores within each task, then compare task means on overlapping tasks ($n = 25$).

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 13 summarizes pairwise agreement across all rater pairs in the current collection.

These dispersion levels are consistent with a difficult semantic-labeling task, so we interpret judge–human calibration as alignment to a noisy expert signal rather than a noise-free ground truth. This is due to the disagreement between human ratings for item-level comparisons.

Strict-set alignment summary. Table 14 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 15 reports the corresponding aggregation analyses over strict and lenient label sources.

Aggregation experiments and what is averaged. Table 15 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.

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 15). 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.

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 12: Within-group human score dispersion on the 0–5 scale, computed per item and then summarized across items. "Items" is the number of $(\texttt{gold\_file}, \texttt{candidate\_file})$ 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 13: 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 $(\texttt{gold\_file}, \texttt{candidate\_file})$ keys ranges from 44 to 75), and the table reports the mean and median across all such rater pairs.

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.

I Additional Evaluation Tables and Numeric Audit

This appendix keeps the paper-facing evaluation appendix focused on current, reproducible artifacts. The v3 SCSC run is the source of truth for Section 5 and Appendix F; earlier exploratory result summaries remain in the repository but are not used for headline claims.

I.1 Per-Split Breakdown (Canonical Core)

The headline overall SCSC numbers reported in Section 5 (Table 2) are decomposed by dataset split in Table 16 below. Both tables refer to the same canonical-core SCSC evaluation (176 Lean evaluation units after multi-formalization expansion); the full 282-task industrial expansion is reported separately in §I.3.

The CS Set proves hardest: Baseline achieves $\widetilde{S} = 0.000$ due to zero-collapse on classical algorithm tasks, and even Codex reaches only 0.321. The Easy Set is the most accessible ($\widetilde{S}$ up to 0.443), while Security, Real Code, and HumanEval cluster in the 0.3–0.41 range for the best agent. Across all splits, no agent's per-split $\widetilde{S}$ exceeds 0.45 (best is Codex on Easy at 0.443), confirming that end-to-end formal code verification from Python remains a wide-open challenge.

I.2 Verified-Core Sensitivity ($D_1 = D_2 = 1$)

The headline agent-skill score $\widetilde{S}_{\text{skill}} = (\text{IC}_1 \cdot \text{IC}_2 \cdot \text{TC}_1)^{1/3}$ used in Section 5 is constructed precisely so that benchmark-item maturity (D_1, D_2) cannot inflate the agent ranking: the gold-side gates appear only in $\widetilde{Q}_{\text{gold}}$ and in the five-factor evaluation-chain score $\widetilde{S}_5$. A Verified-Core view restricted to canonical-core items with $D_1 = D_2 = 1$ (per-task gold-side passing both tests and proofs without `sorry`/`admit`) is therefore mathematically equivalent, for $\widetilde{S}_{\text{skill}}$, to the all-items view modulo a smaller denominator: items with $D_2 < 1$ enter the average through their agent-side factors only, since D_1, D_2 do not appear in $\widetilde{S}_{\text{skill}}$ at all. We therefore expect the Verified-Core $\widetilde{S}_{\text{skill}}$ ranking to track the all-items ranking reported in Table 2, with item-count uncertainty rather than agent-skill movement; the released CSV un-

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

Table 14: 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 15: **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 8) 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.

der `experiments/38_scsc_full_metric_evals/expt_v3/results/detailed_table.csv` includes the per-task D_1, D_2 disaggregation needed for downstream users to compute restricted high-gold-quality slices. For five-factor $\widetilde{S}_5$, restricting to $D_1 = D_2 = 1$ items collapses the gold-side contribution to 1 on those items, so $\widetilde{S}_5$ on the Verified Core reduces to $(\mathrm{IC}_1 \cdot \mathrm{IC}_2 \cdot \mathrm{TC}_1)^{1/5}$ rather than $\widetilde{S}_{\text{skill}}$; we report $\widetilde{S}_{\text{skill}}$ as the primary number to keep the agent-side and chain views algebraically consistent.

I.3 Auxiliary 884-Task Expansion Run (Compile-Rate Diagnostic)

In parallel with the canonical-core SCSC evaluation reported in Section 5, we ran six agents—three production agents (Claude Code (Sonnet 4.6), Codex (GPT-5.4), Leanstral v2) plus three preview-tier Gemini variants (3-flash, 3-pro, 3.1-pro) accessed through an AlphaApollo iterative scaffold—on the full released 884-task benchmark (602-task canonical core, including the full 460-task SecuritySet—454 paired `security_6858` tasks plus 6 standalone `security_python`—plus the 282-task high-assurance industrial expansion across 14 domains). This run reports *compile rate only*—the fraction of submitted Lean files that pass the Lean 4 kernel typecheck—so it is a strict diagnostic, not an SCSC measurement; full TC_1 judging on the 884-task superset is out of scope for the present paper, and we do not use this run for any headline claim or for any claim about specification coverage (which is the gap our paper actually argues about). The compile-rate ranking on the 884-task superset (Claude Code $>$ Codex $>$ Gemini 3.1-pro subset $>$ Leanstral) tracks the canonical-core $\widetilde{S}$ ranking, suggesting that the dataset expansion does not change the relative ordering of the strongest agents. Table 17 reports per-agent counts including the two partial Gemini Pro variants.

I.4 SCSC v3 Per-Split Detail

I.5 Evaluation Harness Provenance

Every main run uses the Harbor sandbox with a fresh Docker container per task, the Lean 4.22.0 toolchain, Mathlib 4.22.0, and a task-level one-hour agent timeout. The task configuration files set `allow_internet = true` and `build_timeout_sec = 3600`; the evaluation scripts run 3–5 containers concurrently depending on agent cost. Gold Lean files are not mounted inside the agent container. Stored result files in `experiments/38_scsc_full_metric_evals/expt_v3/results/` are sufficient to reproduce the paper tables without rerunning closed-model agents.

Agent	Easy	CS	Real	HE	Sec.	$\widetilde{S}$
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 16: Per-split SCSC scores ($\widetilde{S}$) on the canonical VERIBENCH core (HumanEval, EasySet, CSSet, SecuritySet, RealCodeSet). The CS Set is hardest for agents, with Baseline scoring 0.000 due to zero-collapse. Even the Easy Set saturates below 0.45.

Agent	N_{eval}	with sol.	compile rate ↑
Claude Code (Sonnet 4.6)	884	882	0.992
Codex (GPT-5.4)	884	884	0.971
Leanstral v2	884	882	0.694
Gemini 3-flash-preview †	884	884	0.476
Gemini 3-pro-preview †	289	288	0.153
Gemini 3.1-pro-preview †	884	114	0.886

Table 17: Full per-variant compile rates on the released 884-task VERIBENCH dataset (the canonical-core SCSC evaluation reported in Section 5 is contained within this 884-task superset). †Gemini 3-pro-preview and 3.1-pro-preview are partial: Pro-tier eval runs were chunked due to long per-task reasoning latency, and their reported solution counts reflect only completed chunks within the eval window. 3-flash-preview reflects all 884 tasks attempted (with full per-chunk recovery). Compile rate is reported only over tasks with a submitted solution. This auxiliary run is a compile-rate diagnostic only and is not used for any headline or specification-coverage claim; full TC_1 judging on the 884-task superset is out of scope for the present paper.

J Floating-Point and Partial-Definition Audit

This audit motivated the numeric specification-layer policy in Section 3. It classifies remaining `sorry` obligations by whether they are ordinary proof-engineering targets or blocked by default modeling choices that are poor benchmark targets. Counts below are the audit at submission time; the released artifact ships the scanner and the categorized output so the same classification is reproducible on any future revision of the gold corpus.

Lean's primitive `Float` is suitable for execution but not for default universal proof obligations: its arithmetic operations are not exposed as a transparent theorem-rich IEEE-754 model to the kernel. Similarly, `partial def` uses unsafe recursion internally, so the kernel cannot reduce it in the way needed for general correctness proofs. These are not merely hard proof obligations; they are poor default targets for a benchmark that aims to measure agent skill rather than ecosystem opacity.

Accepted layers define what the coverage judge may credit. Populated layers define what the gold file actually proves. A layer can be accepted without being populated: for example, `FleanLayer` and `IntervalLayer` are accepted in v1 if the agent's generated file compiles and the theorem semantically covers the obligation, but the gold files do not generally populate those layers. An accepted external layer only helps an agent if the generated Lean file compiles in the evaluation environment. In v1, FloatSpec may be added as an evaluation dependency for floating-point-essential tasks. Flean and Interval are accepted semantically only when the agent provides self-contained definitions or the corresponding dependency is available in the evaluation environment.

```
/-
spec_layers_accepted:
  - FloatLayer       -- runtime-only primitive Float tests
  - RealLayer        -- mathematical intent over Real
  - RatLayer         -- exact rational semantics when appropriate
  - FloatSpecLayer   -- IEEE-style finite precision, when available
```

Agent	Split	IC1	IC2	TC1	D1	D2	IC	TC	D	S_tilde	n
Codex (GPT-5.4)	Easy	1.000	0.226	0.102	0.976	0.710	0.475	0.102	0.832	0.443	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	0.963	0.463	0.476	0.111	0.668	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	0.407	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	0.435	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 (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 18: SCSC v3 detailed table. Source: the per-agent v3 JSON summaries in `experiments/38_scsc_full_metric_evals/expt_v3/results/`.

Category	Count	Interpretation
`FLOAT_DEPENDENT`	44	Universal primitive-`Float` proof obligations
`PARTIAL_DEPENDENT`	38	Proofs blocked by `partial def` opacity
`NATIVE_DECIDE_LEAK`	0	Invalid use of `native_decide` for non-ground goals
`OTHER`	494	Ordinary proof-engineering targets

Table 19: Audit of remaining `sorry` obligations at submission time. The scanner and its category definitions are released with the artifact so the classification can be re-derived on any subsequent revision of the gold corpus.

```
  - FleanLayer       -- accepted if dependency or self-contained definitions compile
  - IntervalLayer    -- accepted for certified numerical bounds

spec_layers_populated:
  - RealLayer

intent:
  State the numeric behavior this task is meant to test.
  Mark floating_point_essential only when rounding, NaN, infinities,
  signed zero, overflow, or underflow are part of the specification.

judge_instructions:
  TC1 should credit any accepted proof layer that states an equivalent
  or stronger obligation under the declared intent.
  Primitive FloatLayer claims are runtime-only unless the agent provides
  a proved bridge to an accepted proof layer.
  Theorem or lemma declarations containing sorry or admit are unproved.
-/
```

Listing 1: Standard numeric specification-layer policy comment template.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes] .

   Justification: The abstract, Introduction, metric section, results section, and Discussion state the benchmark scope, SCSC factors, gold-validity gates, judge calibration, and limitations.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes] .

   Justification: Section 6 discusses benchmark scope, incomplete gold proofs, LLM-judge limitations, floating-point semantics, and single-run agent evaluation limits.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA] .

Justification: The paper defines a benchmark and metric but does not present a new mathematical theorem with a proof obligation beyond the stated metric equations.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes] .

   Justification: Sections 5, 6, and Appendix D.8 specify the Lean version, harness, timeout policy, output parsing, failure counting, and result source paths.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes] .

Justification: The anonymized supplemental artifact is described in Appendix D.8 and includes the dataset, Lean project, metric scripts, prompts, and stored v3 result files.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

Answer: [Yes] .

Justification: Section 5 describes agent identities, toolchain, sandboxing, timeouts, allowed tools, output parsing, and failure-counting policy; appendix tables give result provenance.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No] .

Justification: Human–judge calibration reports correlations, confidence intervals, and p-values, but the closed-model agent evaluations are single expensive runs and do not yet include multi-seed error bars.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes] .

   Justification: Section 5 and Appendix I describe the Docker/Harbor execution model, one-hour task budgets, concurrent workers, and closed-model credential requirements.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes] .

   Justification: The work conforms to the NeurIPS Code of Ethics to the authors' knowledge; the submission is anonymized and the artifact description avoids deanonymizing repository URLs.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes] .

    Justification: Appendix D.11 discusses positive impacts for verification evaluation and negative risks (over-trusting benchmark scores, evaluation-driven over-fitting, misuse of security examples) along with mitigations.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [Yes] .

    Justification: The release contains benchmark tasks and evaluation code rather than model weights; security examples are framed as verification tasks, and limitations/gold-validity gates are reported to reduce misuse as a deployment guarantee.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes] .

    Justification: The paper cites the source benchmarks and libraries used to construct tasks, including HumanEval, MIT 6.858 materials, Lean/Mathlib, and Python standard-library-derived routines; final release metadata should include full license files.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes] .

    Justification: VERIBENCH is a new benchmark artifact, documented in Sections 3–5 and the appendices, with artifact setup and smoke-test commands in Appendix D.8.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [Yes] .

    Justification: Appendix H includes rater instructions, rating scale, item counts, input visibility, independence constraints, and aggregation details; the raters were expert/collaborator volunteers rather than crowdsourced workers.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA] .

Justification: The human-judgement component consists of expert ratings of non-sensitive code artifacts and does not collect personal data; the anonymous submission does not include institution-identifying IRB details.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes] .

Justification: LLMs are central to the benchmark construction, evaluated agents, and TC1 judge; these roles are described in Sections 3, 4, and 5.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2026/LLM`) for what should or should not be described.