ISOMORPHISM IS NOT ENOUGH: AN AGENTIC AUDIT OF VERIFIED CODE GENERATION IN LEAN

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic systems have recently emerged as state-of-the-art approaches for automated theorem proving in formal mathematics. To assess how far these capabilities extend to program verification and how reliably existing benchmarks can measure them, we audit CLEVER, a Lean 4 benchmark for verifiable code generation, using three independent agentic stacks based on Claude Opus 4.6, GPT-5.4, and Kimi K2.5. Our audit finds arguably erroneous reference specifications in 80 of 161 entries, each accepted by the benchmark's authors, distills a taxonomy of benchmark errors, and exposes structural limitations of isomorphism-based scoring, whose signal breaks down when generated and reference specifications formalize different yet defensible readings. In the process, the stacks generate arguably valid specifications for up to 98.8% of entries, certify implementations against sound reference specifications for up to 92.3%, and reach up to 96.2% end-to-end success over entries with sound premises, suggesting that CLEVER no longer stresses modern agentic provers as much as its design intended. Looking ahead, we translate our findings into recommendations for future work, with our error analysis doubling as a repair list for CLEVER and a heuristic checklist for the next generation of benchmarks.

1 INTRODUCTION

Large Language Models (LLMs) are known to hallucinate, and even recent "reasoning" models may produce arguments that appear coherent but are logically invalid. In contrast, formal logic offers a long-standing and rigorous alternative: mathematical arguments can be expressed in formal languages and mechanically checked for correctness. This approach is embodied in interactive proof assistants based on type theory (e.g., Rocq (Rocq Development Team, 2026), Lean (Moura & Ullrich, 2021), Agda (Norell, 2007)), higher-order logic (e.g., Isabelle (Nipkow et al., 2002), HOL4 (Slind & Norrish, 2008), HOL-Light (Harrison, 1996)), and set theory (e.g., Mizar (Bancerek et al., 2015), MetaMath (Carneiro et al., 2023)). We focus on the first category, in particular the Lean 4 theorem prover.

Recent years have seen striking progress in applying LLMs to automated proof construction in type-theoretic settings. Notable systems achieve strong performance on formalized mathematics benchmarks in Lean 4 (Hubert et al., 2025; Achim et al., 2025; Chen et al., 2025), as well as in Isabelle (Lin et al., 2024; Xu et al., 2025) and Rocq (Bayazıt et al., 2025; Viennot et al., 2025). These systems employ a range of architectural strategies, including whole-proof generation, proof-state search, iterative self-correction on prover feedback (Lin et al., 2025; Leang et al., 2025), and increasingly, agentic designs that integrate planning, tool use, and iterative refinement.

We study how these advances translate to *verified code generation*, or *vericoding* for short, a setting both practically relevant and structurally challenging. It is also far less explored: Lean 4 hosts many benchmarks for formal mathematics but only a handful for verified code generation (Section 2.1), and strong results on the former need not carry over to the latter. Unlike mathematics, program verification requires reasoning about executable artifacts, explicit specifications, and subtle side conditions, all under strict syntactic and semantic constraints. To this end, we conduct a systematic *audit* of CLEVER (Thakur et al., 2025), one of the few benchmarks for end-to-end code generation and verification in Lean 4, and of its evaluation methodology. We evaluate three agentic stacks (Claude Code with Opus 4.6, Codex CLI with GPT-5.4, and OpenHands with Kimi K2.5) on its

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

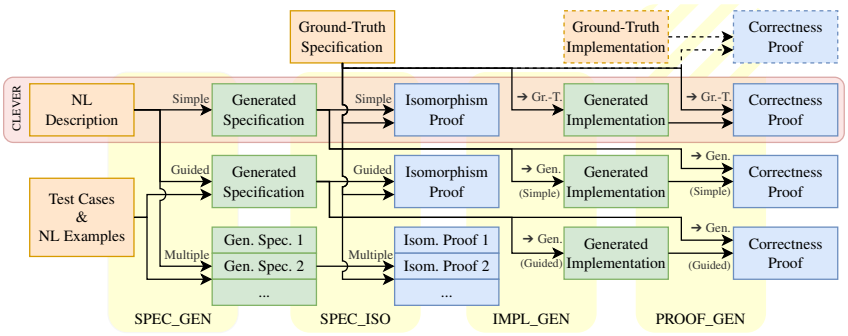

Figure 1: Schematic overview of our experimental pipeline. The four generation and proof tasks are identified in yellow: CLEVER's original intended setup spans horizontally in red across the top row, while other pathways represent our custom variations of the setup. The dashed portion in the top-right corner is the preliminary experimentation setup of Appendix A.

full pipeline, across successive prompt regimes and task variations; the results show what agentic systems can certify, and where isomorphism-based scoring obscures that capability. Figure 1 provides a schematic overview of our experimental pipeline.

Our contributions are as follows:

Benchmark audit. We run the full CLEVER pipeline, in multiple variations of its 4 tasks, with three agentic stacks based on Claude Opus 4.6, GPT-5.4, and Kimi K2.5, on all 161 entries. The audit finds arguably erroneous reference specifications in 80 entries, defects in the benchmark's fixed components that make 4 entries infeasible, and 13 further entries whose NL descriptions admit valid readings that differ from the reference, which isomorphism-based scoring then rejects. The verdicts are stable across prompts, tasks, and stacks, and were reviewed and accepted, entry by entry, by the CLEVER authors. We deliver them as a per-entry annotation and as taxonomies of errors and failure modes, intended as a repair list for CLEVER and a heuristic checklist for future benchmarks.

Methodological insight. Our findings expose structural limitations of isomorphism-against-a-ground-truth as an evaluation method for autoformalization: its signal breaks down when generated and reference specifications formalize different yet defensible readings, and it accepts vacuous equivalences between unsatisfiable specifications. They motivate recommendations for the next generation of verification benchmarks: less buggy datasets, more principled evaluation, and more challenging tasks.

Saturation. The three stacks generate arguably valid specifications for 87.0–98.8% of all entries, certify implementations against 67.9–92.3% of the sound references, and reach 75.2–96.2% end-to-end success on sound premises, with the two frontier stacks near the top of each range. Under CLEVER's own end-to-end criterion they succeed on up to 59 of 161 entries, where the original evaluation reached 1/161.

2 METHODOLOGY

2.1 AUDIT SCOPE

The CLEVER benchmark (Thakur et al., 2025) is a curated dataset of 161 problems adapted from HUMANEVAL (Chen et al., 2021), testing end-to-end automatic code generation and verification in Lean. Each problem is provided as an annotated Lean file containing multiple sections: (1) the natural language (NL) specification of a function (formatted as a Python-style docstring in a comment), complete with the function signature and example usages; (2) the Lean 4 signature of the formal specification `generated_spec`, with a `sorry` placeholder to be replaced by the generated specification; (3) a human-authored ground-truth specification `problem_spec` with the same signature; (4) an *isomorphism* theorem stating the semantical equivalence between `generated_spec` and `problem_spec`, with placeholder `sorry`; (5) the Lean 4 signature of the implementation, with placeholder `sorry`; and (6) a *correctness* theorem stating that the implementation satisfies the `problem_spec` specification, with placeholder `sorry`. Helper definitions for problems that require them are also provided, as well as some formal test cases for the implementation.

To evaluate the models, the benchmark employs a staged pipeline, defining 4 different tasks grouped in two stages: *Specification certification* consists of generating the `generated_spec` specification (SPEC_GEN), then proving it is isomorphic to the ground-truth `problem_spec` (SPEC_ISO); *Implementation certification* consists of implementing the function (IMPL_GEN), then proving it satisfies the `problem_spec` specification (PROOF_GEN). The provided testing harness selectively retrieves the sections related to each task; solutions are then evaluated by formatting a new Lean script with the generated elements and testing it for successful compilation and absence of `sorry` keywords.

Benchmark emendation. A custom fork of the original repository tracks our fixes in the benchmark and the minor edits to the evaluation scripts required by our setup. Emendations to the benchmark entries were made in order to fix errors and inconsistencies in formatting, typos and erroneous test cases, as experimentation progressed and issues of this kind emerged. All other elements such as ground-truth specifications, helper functions, and theorem signatures were left untouched.

Prior work. The original paper (Thakur et al., 2025) reported an end-to-end ceiling of $1/161$ ($\approx 0.62\%$) under a pass@k-seconds protocol with a 600 s budget, reached by several few-shot and COPRA-based (Thakur et al., 2024) configurations; the best per-stage figures were $14/161$ (8.7%) for implementation certification and $3/161$ (1.86%) for specification certification. To our knowledge, no later work has exceeded the end-to-end ceiling under the same protocol. Follow-ups report partial pipelines only: Goedel-Code-Prover (Li et al., 2026) reaches 54.0% on the correctness proofs alone, given implementations generated by GPT-5.2 and manually verified, while frontier reasoning models reach at most 23.6%; WybeCoder (Gloeckle et al., 2026) evaluates an imperative translation of CLEVER, not the benchmark itself. Specification-level property-based testing (Feng et al., 2026) flags 18 entries (11.2%) as defective, 16 of them for underspecified postconditions, and Goedel-Code-Prover's quickcheck finds counterexamples to the correctness theorem, specification and implementation combined, for 10 entries.

Other benchmarks. Most benchmarks for verifiable code generation supply the formal specification and test only the downstream stages: FVAPPS (Dougherty & Mehta, 2025) casts APPS problems as Lean 4 signatures and theorems to implement and prove, the vericoding benchmark (Bursuc et al., 2025) and AlgoVeri (Zhao et al., 2026) do the same across Dafny, Verus, and Lean, and NTP4VC (Xu et al., 2026) isolates the proof stage, discharging verification conditions from existing verification projects. Those that also require a generated specification start from other sources or target other languages: VeriBench (Miranda et al., 2026) autoformalizes Python reference programs into Lean 4, VeriContest (Xie et al., 2026) targets Rust with Verus, and VeriEquivBench (Zeng et al., 2025) Dafny.

Vericoding in Lean. To our knowledge, at the time of writing only CLEVER and VERINA (Ye et al., 2025) cover the full Lean 4 pipeline from an NL description to a certified implementation, scoring the generated specification against a reference within Lean itself. They differ in how: VERINA checks soundness and completeness with an LLM-based prover, which rarely settles the check, and otherwise with test suites and property-based testing, whereas CLEVER requires a machine-checked isomorphism proof.

Preliminary experiments. An exploratory study on both benchmarks, restricted to proof generation from the reference specifications and implementations, compared leading agentic and non-agentic provers (Appendix A). Agentic systems based on Claude Code already performed best on the verified-correct portions of both benchmarks, with Aristotle (Achim et al., 2025) close behind, while specialized whole-proof generation models and symbolic tactics trailed substantially. The strongest agents also consistently identified errors in the benchmarks' specifications and implementations, in several cases proposing fixes and proving the corrected statements. These errors point to the weak spot of CLEVER's scoring. By accepting a generated specification only through a formal proof against a single reference, it is the stricter check of the two, but also the most exposed to flaws in that reference. The present work therefore audits this mechanism, evaluating agentic systems on the full CLEVER pipeline with its native evaluation infrastructure.

2.2 EXPERIMENTAL SETUP

We evaluated three agentic stacks on all 161 CLEVER entries: the Claude Agent SDK running Claude Opus 4.6, the OpenAI Codex CLI running GPT-5.4, and the OpenHands SDK running the open-weight Kimi K2.5. All three used the same benchmark tasks, prompt regimes, and Lean tooling: *lean-lsp-mcp* (Dressler, 2026) exposes via MCP the Lean LSP and provides search tools over the

project and Mathlib, while *lean4-skills* (Freer, 2026) injects Lean-specific instructions and commands, tactic best practices and workflow patterns into the model's context.

For each attempt, the system initializes a temporary Lean project, creates a file containing the relevant parts of a benchmark entry, and prompts a fresh agent instance to edit it. After completion or timeout, we retain the Lean file, conversation and tool traces, and run metadata. Each agent receives the task description as system prompt: objective, input and output formats, behavior rules, and any guidelines to follow. A short NL request then asks it to fill a specification or implementation (SPEC_GEN, IMPL_GEN); theorem-proving tasks (SPEC_ISO, PROOF_GEN) instead invoke the `/lean4:autoprove` command from the *lean4-skills* package, starting a multi-cycle theorem-proving routine. The stacks differ in agent framework, model serving, time budget, and the way skills and commands are delivered; Kimi also ran under an in-run compliance check. Appendix B details these differences and the status-flag protocol, summarized below.

We successively developed multiple prompt regimes to test whether more structured guidance could overcome the benchmark limitations that emerged. An initial *simple* prompt provided only the task description and a request to fill the relevant section. The *guided* prompt added a short set of instructions for each task, derived by Claude, the highest-performing stack, from an analysis of its own initial run. A further *multiple-specification* prompt was run only with Claude, on entries without a successful guided SPEC_ISO run, asking it to generate multiple candidate specifications at once.

Status reporting. After each task, the agent reports OK for a completed, compiling solution, FAIL for an incomplete attempt, or ISSUE when it argues that a bug or incorrect assumption makes the task impossible (Appendix B.2). For SPEC_ISO, MISMATCH instead denotes two different but defensible readings of the NL description, with ISSUE reserved for an erroneous specification (one contradicting the NL description or test cases). For PROOF_GEN, the issue may lie in the specification, implementation, or fixed benchmark components. The agents were asked to support these diagnoses with arguments or counterexamples. A handful of flags were amended for technical reasons such as a prose-only answer, with each amendment tracked in the released solution files.

To extract deeper diagnostic insights, solutions and traces in the ISSUE category were further analyzed with Claude to identify where the bug originated: in the ground-truth `problem_spec`, the generated `generated_spec` or both for SPEC_ISO; and in the specification, implementation, or fixed benchmark components for PROOF_GEN. Likewise, MISMATCH entries were analyzed to determine whether the generated specification was stronger or weaker than the reference (i.e. one implied the other), or neither.

Cross-checking of results. Reported flags of all tasks directly supported by the benchmark (all SPEC_ISO runs and PROOF_GEN from ground-truth specs) were cross-checked against the benchmark harness: all and only OK solutions passed, apart from one correct solution obstructed by a test-case bug and one successful run reported as a timeout. For all other tasks (SPEC_GEN, IMPL_GEN, and PROOF_GEN from generated specs), OK solutions were cross-checked against the Lean typechecker and the absence of `sorry` and `axiom` in the output, with discrepancies in only 3 PROOF_GEN runs due to bugs in test cases, all already flagged as ISSUE at the earlier IMPL_GEN stage.

Merged annotation protocol. As a sanity check, the solutions and traces of the top-performing stack, Claude Code, were systematically inspected for soundness and underlying structural bugs by the authors. Results of the SPEC_ISO runs were then merged to produce a single merged annotation for each entry, following fixed rules (no valid specification produced $\rightarrow$ FAIL; a ground-truth bug in any run $\rightarrow$ ISSUE; otherwise, a successful run $\rightarrow$ OK; otherwise MISMATCH), discussed in Section 3.1.

Independent validation. We shared the complete audit with the CLEVER authors: the merged annotation and the per-entry findings of Appendix I, each as a short summary and a long-form description, covering the 80 flagged references, the 4 entries infeasible because of their fixed components, and the 13 MISMATCH classifications. They reviewed every finding and accepted all of them; the corrections are being incorporated into the next release of the benchmark, and we will link the upstream issue in the camera-ready. This confirmation covers every verdict that defines the sound subsets C_{spec}, C_{gt}, and C_{gen} used throughout Section 3; the validity of generated specifications remains the agents' own judgment, inspected by the authors as described in Section 3.1.

Table 1: Specification certification: SPEC_ISO outcomes on each stack's SPEC_GEN output, one attempt per entry and run; cells give count (%). *Valid*, *Accepted*, and C_{spec} as in Section 3.1; *Any*: at least one run succeeds (S, G, M: Simple, Guided, Multiple). Multiple: Claude only, on the 105 entries without a Guided OK (24 in C_{spec}), succeeding if any of up to four candidates does. Claude's Any (S+G+M) row matches its merged annotation: all entries but the 2 FAIL are *Valid*, and exactly the 65 OK entries are *Accepted*. Union: any stack, any run.

count (%)		Valid		Accepted	
Stack	Run	/161	C_{spec} /80	/161	C_{spec} /80
Claude Code + Opus 4.6	Simple	147 (91.3%)	70 (87.5%)	59 (36.6%)	58 (72.5%)
	Guided	157 (97.5%)	76 (95.0%)	56 (34.8%)	56 (70.0%)
	Any (S+G)	*158 (98.1%)*	*77 (96.3%)*	*62 (38.5%)*	*61 (76.3%)*
	Multiple	101	21	11	9
	Any (S+G+M)	*159 (98.8%)*	*78 (97.5%)*	*67 (41.6%)*	*65 (81.3%)*
Codex CLI + GPT-5.4	Simple	136 (84.5%)	65 (81.3%)	49 (30.4%)	48 (60.0%)
	Guided	140 (87.0%)	67 (83.8%)	56 (34.8%)	55 (68.8%)
	Any	*150 (93.2%)*	*73 (91.3%)*	*60 (37.3%)*	*58 (72.5%)*
OpenHands + Kimi K2.5	Simple	121 (75.2%)	52 (65.0%)	34 (21.1%)	33 (41.3%)
	Guided	123 (76.4%)	57 (71.3%)	42 (26.1%)	41 (51.3%)
	Any	*140 (87.0%)*	*63 (78.8%)*	*46 (28.6%)*	*44 (55.0%)*
Union (any stack, any run)		160 (99.4%)	79 (98.8%)	73 (45.3%)	68 (85.0%)

3 RESULTS

We evaluated the three stacks on all four tasks, with a single attempt per entry and run. This already produced specifications, implementations, and proofs of consistently high quality, often not credited by the benchmark harness, which led us to design the sequence of prompts described below. Appendix C gives per-entry results.

3.1 SPECIFICATION CERTIFICATION

We score generated specifications in two ways. A specification is *valid* when its SPEC_ISO run finds it an admissible reading of the NL description (flag OK, MISMATCH, or ISSUE with the bug in `problem_spec` alone), and *accepted* when its isomorphism to `problem_spec` is proved (OK, confirmed by the harness), CLEVER's own criterion. Table 1 reports both over all entries and over C_{spec}, the 80 entries whose `problem_spec` and helper definitions the merged annotation found sound.

Simple prompt. The first minimal prompt gave only the task instructions and the input/output format, without further guidance; as in the original protocol, SPEC_GEN saw the NL description alone, without example usages or test cases. All stacks produced compiling specifications for every entry at the SPEC_GEN stage, but only 59, 49, and 34 were *accepted* for Claude, GPT-5.4, and Kimi at the SPEC_ISO stage. Proof synthesis was rarely the obstacle: besides 4 timeouts, Claude flagged 66 non-isomorphic entries as ISSUE in the reference, 22 as MISMATCH, only 9 as ISSUE in its own specification, and 1 in both, leaving 147 *valid* specifications. GPT-5.4 and Kimi follow the same pattern, with more failed proofs and own-specification bugs (136 and 121 *valid*). Claude's MISMATCH specifications were mostly stronger than the reference (15 against 1, with 6 incomparable), consistent with the references' tendency towards under-restrictive formulations (e.g. leaving behavior outside the valid input domain unconstrained); Kimi shares this skew (22 against 7), whereas GPT-5.4 reverses it (4 against 12), so the direction also reflects each stack's authoring style.

Guided prompt. A second prompt sought to align generated specifications with the reference without revealing it. A new Claude instance derived candidate guidelines from the outputs and traces of Claude's own first run, which we refined by hand and stripped of benchmark-specific examples; they covered the structure of the specification, handling of edge cases and degenerate inputs, preference for relational formulations, and Lean 4 syntax. The prompt also fixed a format for the specification body, and the setup now supplied the example usages and formal test cases, aiming to clarify the reading the reference adopted. Guidance raised the number of *valid* specifications for every stack (to 157, 140, and 123), but not that of *accepted* ones: these rose for GPT-5.4 (49 to 56)

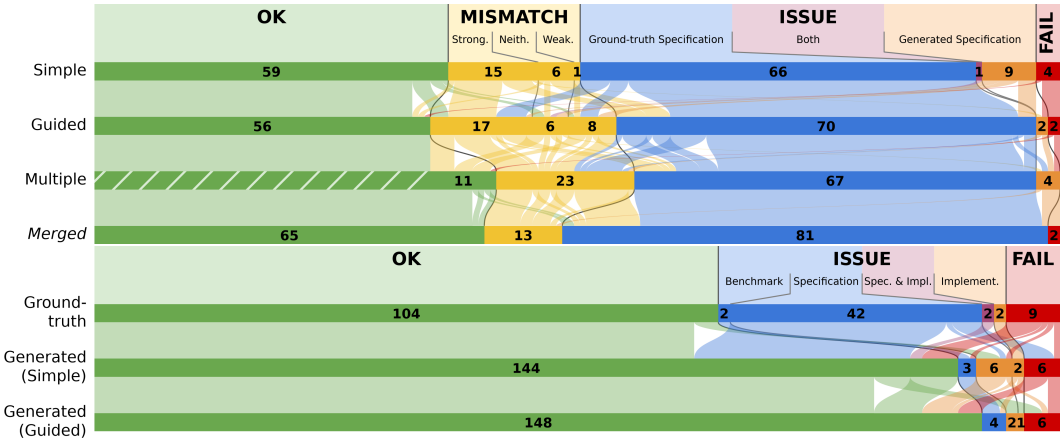

Figure 2: Distribution of Claude's output flags for the SPEC_ISO runs (top) and for PROOF_GEN against the reference and against its simple and guided generated specifications (bottom). The color scheme, used throughout the paper, is green for OK, red for FAIL, yellow for MISMATCH, blue for ISSUE in benchmark components, orange for ISSUE in generated components, and purple for issues in both. In the top panel, the hatched segment of the Multiple specifications row marks the 56 entries already OK in the guided run, which were not re-tested; in the bottom panel, the color of a specification ISSUE follows its source: reference in the first row, generated in the others.

and Kimi (34 to 42) and fell for Claude (59 to 56), whose own-specification ISSUE flags dropped from 10 to 2 while its MISMATCH flags rose from 22 to 31. The gap between *valid* and *accepted* persists: on C_{spec}, Claude's two runs yield a *valid* specification for 77 entries but an *accepted* one for only 61, and the three stacks together reach 65.

Multiple specifications prompt. A MISMATCH fails only because the generated specification reads the NL description differently from the reference. To close this gap, a third prompt, run only with Claude, the strongest stack, asked for "all and only" the admissible readings of each entry, varying preconditions, edge-case handling, quantifier scope, operation semantics, and property strength, under the same guidelines and test cases. On the 105 entries without a guided OK, Claude produced 290 candidates (at most four per entry), of which only 15, spanning 11 entries, proved isomorphic to the reference. Enumerating readings thus recovered few matches: on C_{spec}, Claude's *accepted* entries rose only from 61 to 65.

Merged annotation. As a final step, we merged Claude's three runs into a single per-entry annotation and, as a sanity check, manually inspected its outputs and traces for evident errors and inconsistencies. This pass resolved disagreements between runs, caught a lingering structural bug invisible through the flag system (problem 160, where isomorphism succeeded thanks to a flawed helper), and found Claude's classifications and supporting arguments reliable and well-reasoned in all cases; the resulting verdicts on the benchmark were then reviewed and accepted by the CLEVER authors (Section 2.2, Independent validation). The annotation counts 81 ISSUE entries, 80 with an arguably flawed `problem_spec` and one (problem 160) with the flawed helper, which form the complement of C_{spec}; 65 OK entries, with at least one *accepted* specification; 13 MISMATCH entries, whose sound reference no generated specification ever matched; and 2 FAIL entries.

3.2 IMPLEMENTATION CERTIFICATION

An implementation is *certified* when PROOF_GEN proves it correct (OK) against the given specification: against the reference, this is CLEVER's own criterion, scored by the harness; against a generated specification, which the harness cannot natively check, we recompiled every proof independently. A certification holds only relative to its target, so an end-to-end success is *valid* or *accepted* according to its specification. The set of sound entries follows the target: against the reference, C_{gt} (78 entries) also excludes entries 32 and 123, whose fixed components make the task infeasible; against generated specifications, C_{gen} (157 entries) excludes only the 4 entries with such defects.

Against reference specifications. Following CLEVER's protocol, the first run generated implementations from the reference specifications and certified them: Claude, GPT-5.4, and Kimi certified 104,

Table 2: Implementation certification, one attempt per entry and run; cells give count (%). *Top*: against the reference specification (CLEVER's protocol), one run per stack; *Certified* is PROOF_GEN OK against `problem_spec`, and end-to-end success also requires a *Valid* or *Accepted* Simple or Guided specification, *Accepted* being CLEVER's end-to-end criterion (1/161 in the original evaluation; with Claude's Multiple run, *Valid* 102 and 68, *Accepted* 59 and 59). *Bottom*: against each stack's own generated specification, per SPEC_GEN chain (*Any*: either chain); proofs are independently recompiled and make no claim on the specification's validity, and end-to-end success requires a *Valid* or *Accepted* specification from the same chain, whose isomorphism transfers the proof to `problem_spec`. Terms as in Sections 3.1 and 3.2.

count (%)	Certified		End-to-end Valid		End-to-end Accepted	
Stack	/161	C_{gt} /78	/161	C_{gt} /78	/161	C_{gt} /78
Claude Code + Opus 4.6	104 (64.6%)	70 (89.7%)	101 (62.7%)	67 (85.9%)	55 (34.2%)	55 (70.5%)
Codex CLI + GPT-5.4	105 (65.2%)	72 (92.3%)	100 (62.1%)	68 (87.2%)	56 (34.8%)	55 (70.5%)
OpenHands + Kimi K2.5	77 (47.8%)	53 (67.9%)	67 (41.6%)	44 (56.4%)	35 (21.7%)	35 (44.9%)
Union (any stack)	109 (67.7%)	74 (94.9%)	108 (67.1%)	73 (93.6%)	63 (39.1%)	62 (79.5%)

count (%)		Certified		End-to-end Valid		End-to-end Accepted	
Stack	Chain	/161	C_{gen} /157	/161	C_{gen} /157	/161	C_{gen} /157
Claude Code	Simple chain	144 (89.4%)	144 (91.7%)	135 (83.9%)	135 (86.0%)	56 (34.8%)	56 (35.7%)
+	Guided chain	148 (91.9%)	148 (94.3%)	145 (90.1%)	145 (92.4%)	56 (34.8%)	56 (35.7%)
Opus 4.6	*Any*	*154 (95.7%)*	*154 (98.1%)*	*151 (93.8%)*	*151 (96.2%)*	*60 (37.3%)*	*60 (38.2%)*
Codex CLI	Simple chain	142 (88.2%)	141 (89.8%)	125 (77.6%)	125 (79.6%)	48 (29.8%)	48 (30.6%)
+	Guided chain	142 (88.2%)	141 (89.8%)	128 (79.5%)	127 (80.9%)	54 (33.5%)	54 (34.4%)
GPT-5.4	*Any*	*155 (96.3%)*	*153 (97.5%)*	*143 (88.8%)*	*142 (90.4%)*	*57 (35.4%)*	*57 (36.3%)*
OpenHands	Simple chain	123 (76.4%)	123 (78.3%)	98 (60.9%)	98 (62.4%)	30 (18.6%)	30 (19.1%)
+	Guided chain	119 (73.9%)	119 (75.8%)	95 (59.0%)	95 (60.5%)	34 (21.1%)	34 (21.7%)
Kimi K2.5	*Any*	*140 (87.0%)*	*140 (89.2%)*	*118 (73.3%)*	*118 (75.2%)*	*38 (23.6%)*	*38 (24.2%)*
Union (any stack, any chain)		158 (98.1%)	156 (99.4%)	156 (96.9%)	155 (98.7%)	65 (40.4%)	65 (41.4%)

105, and 77 entries (Table 2). At the IMPL_GEN stage, Claude's few ISSUE flags pointed to bugs in the provided test cases, which we then fixed. Most ISSUE flags blame the specification (42 of Claude's 48), and every such flag, for all three stacks, falls on an entry the merged annotation labels ISSUE. A flawed reference can reject a correct implementation, while an underspecified one can still be met. On C_{gt}, the frontier stacks are on par (70 and 72 of 78, 89.7% and 92.3%), while Kimi trails (53), mostly through failures and timeouts. Requiring also an *accepted* specification yields CLEVER's own end-to-end criterion, met by 55, 56, and 35 entries (59 for Claude with the multiple specifications run), against 1 of 161 in the original evaluation, obtained, however, with earlier models, other tooling, and a 600 s budget.

Against generated specifications. Two further runs implemented and certified each stack's own simple and guided specifications; at the IMPL_GEN stage, Claude's three ISSUE flags in the simple run pointed to test cases inconsistent with the generated specification, which had been written without access to them. Claude certified 144 and 148 entries, GPT-5.4 142 and 142, and Kimi 123 and 119: unlike at the specification stage, guidance brought no consistent gain. The remaining ISSUE flags stem from generated implementations (e.g. finite fuel that does not cover all inputs), generated specifications (e.g. mishandled borderline cases), and occasionally the benchmark's fixed parts (Section 4). Over both runs, the stacks certify 154, 155, and 140 entries, and at least one stack certifies 158. These proofs, however, hold only relative to each stack's own specification. Requiring that specification to be *valid*, Claude reaches end-to-end success on 151 entries, 96.2% of C_{gen}, against 90.4% for GPT-5.4 and 75.2% for Kimi; requiring it to be *accepted* lowers the counts to 60, 57, and 38, the same gap between *valid* and *accepted* found at the specification stage.

4 DISCUSSION

4.1 CONSISTENCY

The audit rests on the claim that a reference specification is flawed, so we first check how consistently this verdict recurs across runs, tasks, and stacks (full analysis in Appendix D).

- **Across runs.** The verdict is largely stable across prompts: between the simple and guided SPEC_ISO runs, Cohen's κ is 0.73 for Claude, 0.81 for GPT-5.4, and 0.55 for Kimi. Claude flags 73 of the 81 ISSUE entries in at least two of its three runs; its MISMATCH verdicts recur as well, in at least two runs for 12 of the 13 entries.
- **Across tasks.** Every reference bug raised during PROOF_GEN falls on an ISSUE entry, for all three stacks (44, 41, and 39 flags). The proof stage alone, however, surfaces only about half of the flawed references: on most of the others, the stacks certify an implementation against the flawed specification (34, 33, and 24 entries), more often when the bug weakens it. Specification-level comparison, rather than implementation proof, exposes most reference bugs.
- **Across provers.** GPT-5.4 and Kimi, whose flags the merged annotation does not use, attribute a bug to the reference on 76 and 62 of the 81 ISSUE entries, and jointly on 77 (95.1%; $\kappa = 0.85$ against the merged annotation, one-sided Fisher $p = 3.4 \cdot 10^{-31}$). Their flags on C_{spec} fall on 8 entries, all flagged by GPT-5.4 (one also by Kimi) and 5 of them MISMATCH entries, suggesting that GPT-5.4 is quick to call a defensible reading a bug. No stack ever proves a flawed reference valid: the few isomorphisms accepted on ISSUE entries reproduce the same bug, reuse the same flawed helper, or are vacuous, with both specifications unsatisfiable on some input. Capability differences surface mainly as failures (on 27, 59, and 170 of 805 certification attempts) and as MISMATCH readings in place of bug claims, and only rarely as contradictions. The ISSUE / MISMATCH boundary is where the stacks diverge most: only 8 of the 13 MISMATCH entries receive a MISMATCH reading from all three.

4.2 TAXONOMY

We now turn from how often the stacks succeed to why they fail. For this analysis we restrict ourselves to Claude, the strongest-performing stack. We had Claude classify the cause behind each ISSUE and MISMATCH flag of its runs, and grouped the diagnoses by the component at fault — ground-truth specifications, generated specifications, implementations, and the benchmark's fixed parts — and, for MISMATCH entries, by the authoring choice that separates the two readings. The resulting taxonomies (Appendix E) serve both as a diagnosis of CLEVER and as a heuristic checklist of pitfalls for the authors of future benchmarks.

A faithful specification must have the right strength and commit to one reading of an often ambiguous NL description: it must formalize the *intent* behind the prose (Lahiri, 2026). The taxonomies show how each component falls short of this goal:

- **Ground-truth specifications** fail mostly through Lean-encoding pitfalls (48 of the 80 flawed references), led by *conjunct-as-guard* patterns, $(P \wedge Q)$ in place of $(P \to Q)$ (15 problems), and *precedence bugs* (12); the other errors are semantic, most often a wrong interpretation of the NL description (21).
- **Claude's generated specifications** show the inverse profile: their bugs are almost only semantic, mostly missing constraints. Guidance cut them from 10 to 2 per run, but they resurfaced when the multiple specifications prompt pushed Claude to explore the space of readings.
- **Mismatches** reflect authoring choices rather than errors, most often on the boundary of the input domain (73 of 142 assignments).
- **Implementations** were rarely at fault (7 cases in 5 problems), for instance through bounded fuel that does not cover all inputs.
- **Fixed benchmark components** make four entries infeasible: two hinge on open conjectures (problems 123 and 39, on the Collatz sequence and on Fibonacci primes), one ships a faulty helper (160), and one lacks a precondition (32, Appendix H).

4.3 RESOURCES AND CONTAMINATION

Appendix F reports statistics on token usage, runtimes, and costs. Appendix G discusses training-data contamination: why the audit's findings do not depend on it, and an exploratory control with GPT-5.1, a model trained before CLEVER became public.

4.4 RECOMMENDATIONS

The same findings also expose structural limitations in how CLEVER is designed and evaluated, and suggest three axes along which the next generation of vericoding benchmarks should improve.

- **Less buggy datasets.** The error analysis of Section 4.2 and Appendix E doubles as a repair list for CLEVER and an authoring checklist for new datasets. Of the 80 bugged ground-truth specifications, 48 exhibit recurring Lean-encoding pitfalls of exactly the kind an authoring-time "linting" pass (e.g. property-based testing plus an agentic judge) is designed to catch.
- **More principled evaluation.** For isomorphism-based scoring to remain a meaningful signal, the generated and reference specifications must come from a shared intent: more precise NL descriptions, explicit formalization hints (e.g. pre-committing to a type signature), or an iterative disambiguation pass in which an LLM judge surfaces and resolves multiple admissible readings until none remains. Scoring must also rule out vacuous successes: CLEVER's isomorphism theorem, of the form $\forall f.\,(\forall x.\,P(f,x)) \leftrightarrow (\forall x.\,Q(f,x))$, holds for any two specifications that no implementation can satisfy, as in two isomorphisms Claude proved on the flawed entries 67 and 160 (Appendix D.3); a pointwise statement, or a witness implementation certified against each specification, would close this gap. Where formal isomorphism is intractable, the reviewed diagnoses and the attribution differences across stacks suggest combining evaluators, LLM-based and deterministic: agentic judges cross-checked by property-based testing of the specifications themselves, and end-to-end synthesis against the model's own specification, which surfaced vacuous, contradictory, or under-constrained specifications that the equivalence check had missed.
- **More challenging tasks.** Once the evaluation is trustworthy, benchmarks should turn to harder sources and longer horizons — multi-function or repository-level verification, richer data structures and side conditions — beyond HUMANEVAL-style single functions.

These changes would restore the reliability of the evaluation signal; making it discriminative again for modern agentic provers is the task these three axes leave open.

5 CONCLUSIONS

We audited CLEVER, one of the few benchmarks that score the full pipeline from an NL description to a certified Lean 4 implementation, with three independent agentic stacks on all 161 entries. Under the benchmark's own scoring, the stacks appear far from solving it: no stack passes the isomorphism test on more than 67 entries, or end to end against the reference on more than 59. Under inspection, most of the gap belongs to the benchmark: 80 reference specifications are arguably erroneous, 4 entries are infeasible because of their fixed components, and 13 more admit valid readings the reference does not share. These verdicts recur across prompts, tasks, and stacks, and were independently confirmed by the benchmark's authors. On sound premises, the stacks certify implementations for up to 92.3% of entries and reach up to 96.2% end to end, with the two frontier stacks within two entries of each other on implementation certification: the benchmark no longer separates them, and much of its remaining difficulty is an artifact of its evaluation.

The structural lesson concerns isomorphism against a single reference as a scoring method for autoformalization. Its signal presumes that the reference formalizes the only defensible reading of the prose, which our mismatch entries show it does not always do; the gap between valid and accepted specifications persisted for every stack under guidance, and for Claude even when enumerating readings; and its theorem form accepts vacuous equivalences between unsatisfiable specifications. Future benchmarks should catch the Lean-encoding pitfalls of our taxonomy at authoring time; restore a shared intent before scoring, or combine evaluators where isomorphism cannot decide; and, once the signal is trustworthy, raise the difficulty beyond single functions. Our per-entry annotation and taxonomies are released as a repair list for CLEVER and a checklist for that next generation.

AI USE STATEMENT

The agentic systems studied in this work generated the specifications, implementations, and proofs we evaluate. In addition, we used generative AI tools to derive the guided prompt's guidelines from early runs, implement the experimental harness and analysis scripts, and classify and interpret the agents' diagnoses. We also used them to aid plotting, literature search, data analysis, and editing for readability. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Our repository (`https://anonymous.4open.science/r/APPV-iclr27`) releases the agent harnesses and prompts, our fork of CLEVER (Section 2.1), all solutions and agent traces, and the scripts behind every table, figure, and statistic. Section 2.2 and Appendix B describe the configurations and pinned tool versions; Appendices C–F and I report per-entry outcomes and further statistics. Since the models are served through endpoints that may change, re-runs may differ on individual entries, but every reported outcome can be re-verified with Lean from the released solutions.

REFERENCES

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level Automated Theorem Proving, October 2025.

Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Paąk, and Josef Urban. Mizar: State-of-the-art and beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge (eds.), *Intelligent Computer Mathematics*, pp. 261–279, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20615-8. doi: 10.1007/978-3-319-20615-8_17.

Barış Bayazıt, Yao Li, and Xujie Si. A case study on the effectiveness of llms in verification with proof assistants, 2025.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A benchmark for vericoding: Formally verified program synthesis, 2025. URL `https://arxiv.org/abs/2509.22908`. Dafny Workshop at POPL 2026.

Mario Carneiro, Chad E. Brown, and Josef Urban. Automated theorem proving for metamath. In Adam Naumowicz and René Thiemann (eds.), *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 9:1–9:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.ITP.2023.9.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021.

Wenhu Chen, Shuo Wei, Hao Wang, et al. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025.

Joshua Clune. LeanHammer, 2026.

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2502.05714. LLM4Code Workshop at ICSE 2025.

Oliver Dressler. Lean-lsp-mcp, 2026.

Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pîrlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. Certified program synthesis with a multi-modal verifier, 2026. URL https://arxiv.org/abs/2604.16584.

Cameron Freer. Lean4-skills, 2026. (preliminary experiments of Appendix A on the earlier commit 5d6bd03).

Fabian Gloeckle, Mantas Bakšys, Darius Feher, Kunhao Zheng, Amaury Hayat, Sean B. Holden, Gabriel Synnaeve, and Peter O'Hearn. WybeCoder: Verified imperative code generation, 2026. URL https://arxiv.org/abs/2603.29088.

John Harrison. Hol light: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD)*, 1996.

Thomas Hubert, Harsh Mehta, Laurent Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y.

Shuvendu K. Lahiri. Intent Formalization: A Grand Challenge for Reliable Coding in the Age of AI Agents, March 2026.

Joshua Ong Jun Leang, Giwon Hong, Wenda Li, and Shay B. Cohen. Theorem prover as a judge for synthetic data generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 29941–29977, Vienna, Austria, 2025. Association for Computational Linguistics.

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical proof search for open state-of-the-art code verification, 2026. URL https://arxiv.org/abs/2603.19329. Conference on Language Modeling (COLM 2026).

Jannis Limperg and Asta Halkjær From. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, pp. 253–266, New York, NY, USA, January 2023. Association for Computing Machinery. ISBN 979-8-4007-0026-2. doi: 10.1145/3573105.3575671.

Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. Fvel: Interactive formal verification environment with large language models via theorem proving, 2024.

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction, August 2025.

Junqi Liu, Zihao Zhou, Zekai Zhu, Marco Dos Santos, Weikun He, Jiawei Liu, Ran Wang, Yunzhou Xie, Junqiao Zhao, Qiufeng Wang, Lihong Zhi, Jia Li, and Wenda Li. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics, January 2026.

Brando Miranda, Srivatsava Daruru, Ethan S. Hersch, Zhanke Zhou, Allen Nie, Daneshvar Amrollahi, Leni Aniva, Iddah Mlauzi, Kirill Acharya, Elyas Obbad, Dilara Soylu, Weston Kirk, Zixiao Jolene Wang, Kai Fronsdal, Ying Li, Donald Poindexter, Jr., Rakshit Kaushik, Shurui Liu, Yegor Denisov-Blanch, Steven Dillmann, Simon Obstbaum, Santiago Cuellar, John Sarracino, Rylan Schaeffer, Mo Tiwari, Donghyun Lee, Bo Han, and Sanmi Koyejo. VeriBench: End-to-end formal verification benchmark for AI coding agents in Lean 4, 2026. URL https://cs.stanford.edu/people/brando9/professional_documents/papers/NeurIPS_2026_VeriBench.pdf. Preprint.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, pp. 625–635, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-79875-8. doi: 10.1007/978-3-030-79876-5_37.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

Chase Norman and Jeremy Avigad. Canonical for Automated Theorem Proving in Lean, April 2025.

Project Numina. Numina-lean-agent, 2026.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition, April 2025.

Rocq Development Team. The rocq prover. *Project website*, 2026. `https://rocq-prover.org`.

Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *Lecture Notes in Computer Science*, pp. 28–32. Springer, 2008. doi: 10.1007/978-3-540-71067-7_6.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024. URL `https://arxiv.org/abs/2310.04353`.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. CLEVER: A Curated Benchmark for Formally Verified Code Generation, October 2025. Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track, 2025.

The Lean Developers. *The Lean Language Reference: The grind tactic*. Lean FRO, 2024. URL `https://lean-lang.org/doc/reference/latest/The--grind--tactic/`. Lean version 4.15.0 or later.

Jules Viennot, Guillaume Baudart, Emilio Jesùs Gallego Arias, and Marc Lelarge. Minif2f in rocq: Automatic translation between proof assistants – a case study, 2025.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning, April 2025.

Zichen Xie, Mrigank Pawagi, Yuxin Liu, Aaditi Rai, Lize Shao, John Berberian Jr., Sicong Che, and Wenxi Wang. VeriContest: A competitive-programming benchmark for verifiable code generation, 2026. URL `https://arxiv.org/abs/2605.08553`.

Qiyuan Xu, Renxi Wang, Peixin Wang, Haonan Li, and Conrad Watt. A minimalist proof language for neural theorem proving over isabelle/hol, 2025.

Qiyuan Xu, Xiaokun Luan, Renxi Wang, Joshua Ong Jun Leang, Peixin Wang, Haonan Li, Wenda Li, and Conrad Watt. Neural theorem proving for verification conditions: A real-world benchmark. In *International Conference on Learning Representations (ICLR)*, 2026. URL `https://arxiv.org/abs/2601.18944`.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation, October 2025. International Conference on Learning Representations (ICLR 2026).

Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. VeriEquivBench: An equivalence score for ground-truth-free evaluation of formally verifiable code, 2025. URL https://arxiv.org/abs/2510.06296. International Conference on Learning Representations (ICLR 2026).

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. AlgoVeri: An aligned benchmark for verified code generation on classical algorithms, 2026. URL https://arxiv.org/abs/2602.09464. International Conference on Machine Learning (ICML 2026).

A Preliminary Experiments

In prior experiments, the authors evaluated a suite of automated provers (agentic, specialized, and symbolic) on proof generation on the CLEVER (Thakur et al., 2025) and VERINA (Ye et al., 2025) datasets, finding that, in that setup, agentic systems substantially outperformed the alternatives. We summarise its findings here, both because it directly motivates the present work and because some of its observations reappear, refined, in our current results.

Setup. The earlier study used a different experimental setup than the one of Section 2.2. It focused exclusively on proof generation starting from ground-truth specifications and implementations, a task outside the scope of either benchmark's provided harness (in Figure 1, the setup illustrated in the dashed portion of the diagram in the top-right corner). For this reason it relied on a custom distillation of the two datasets rather than on their provided evaluation infrastructure: derivative versions (denoted CLEVER-P, VERINA-P) were obtained by reusing the ground-truth specifications and implementations, with other variants distilled to meet requirements for specific provers; NL descriptions were not provided. This restriction also limited the CLEVER subset to the 43 entries (out of 161) for which an implementation was available, plus the benchmark's 6 example entries. The agentic configurations were further built on Claude Opus 4.5 and on an earlier release of *lean4-skills* (Freer, 2026) that predated the current `/autoprove` command, relying instead on the `/fill-sorry` command, and with three independent one-shot attempts per problem. The benchmark releases evaluated were CLEVER v1.4.0 and a contemporary VERINA build, both of which have since been partially patched in light of the bugs reported by those experiments.

Provers. The evaluation covered three families. The *agentic* family included Claude Code with the earlier version of *lean4-skills* and *lean-lsp-mcp* (Dressler, 2026), and the *Numina-Lean-Agent* (Liu et al., 2026; Project Numina, 2026), a wrapper around Claude Code combining it with a fork of *lean-lsp-mcp* extended with semantic search, an informal-prover and a discussion-partner tool, run with `max_rounds=3`. The *whole-proof generation* family — specialized models trained or fine-tuned for formal verification, that interleave chain-of-thought reasoning with formal code generation — was represented by Kimina-Prover-72B (Wang et al., 2025), the highest-performing open model of its kind on MINIF2F, with DeepSeek-Prover-V2 (Ren et al., 2025) and Goedel-Prover-V2 (Lin et al., 2025) in the same class. The *symbolic* family covered Grind (The Lean Developers, 2024), LeanHammer (Clune, 2026), Aesop (Limperg & From, 2023), and Canonical (Norman & Avigad, 2025). Aristotle (Achim et al., 2025), Harmonic's high-level proof-state-aware Monte Carlo Graph Search system, was tested alongside as a separate state-of-the-art baseline (most notably achieving gold-medal performance at IMO 2025).

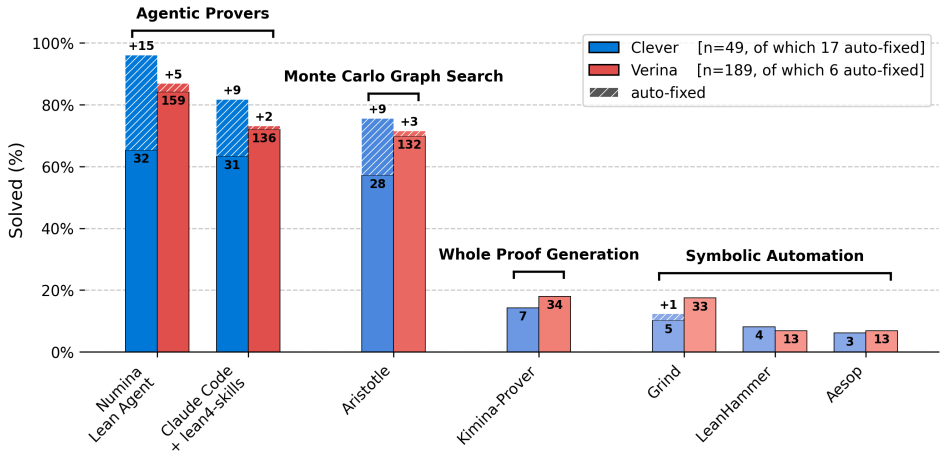

Figure 3: Overview of prover performance on the CLEVER and VERINA benchmarks. Solid bars indicate the number of tests solved on the verified-correct portions of each dataset, while the patterned bar segments above show additional solved instances obtained after correcting erroneous benchmark entries. Numbers annotated with a '+' denote the corresponding count of these additional solves.

Results. The evaluation revealed a clear performance hierarchy, as displayed by Figure 3: agentic systems based on Claude Code performed best, with Aristotle trailing by a small margin; Kimina-Prover displayed pass rates much lower than reported on MINIF2F (7/32 on CLEVER-P, 34/183 on VERINA-P), while symbolic provers provided a stable but significantly lower baseline, consistent with their design as gap-fillers between assertions rather than full proof generators. Considering only the verified-correct portions of the datasets, Numina-Lean-Agent solved all 32 CLEVER-P entries, while Claude Code with *lean4-skills* and Aristotle solved 31 and 28 respectively; out of the 183 correct VERINA-P entries, the same three provers solved 159, 136 and 132. The high performance of the agentic systems was attributed largely to their direct access to compiler feedback through their search and tool-use loop, the main documented difference in implementation between the various approaches; conversely, the whole-proof-generation models were tested in a single one-shot attempt without successive refinement on compiler feedback, and their performance might improve substantially under iterative protocols.

Table 3: Number of solved tests by each prover over the correct entries in the tested benchmarks. Numbers in brackets represent the number of solved entries among the auto-fixed ones.

	CLEVER					VERINA				
Prover	examples		human_eval			basic		advanced		
Numina Lean Agent	5	(+1)	27	(+14)		102	(+2)	57	(+3)	
Claude Code + lean4-skills	5	(+1)	26	(+8)		96		40	(+2)	
Aristotle	5		23	(+9)		89	(+1)	43	(+2)	
Kimina-Prover	4	N/A	3	N/A		31	N/A	3	N/A	
Grind	2		3	(+1)		31		2		
LeanHammer	1		3			13		0		
Aesop	1		2			13		0		
Canonical	0		0			0		0		
number of entries		5	(+1)	27	(+16)		106	(+2)	77	(+4)

A second, less expected finding was that Claude Code and Aristotle consistently identified errors in the benchmarks themselves — Aristotle through its goal-negation search mechanism, Claude Code through its NL replies (sometimes already proposing fixes and certifying the fixed statements without being prompted to do so). Cumulatively, the provers identified 17 erroneous entries in CLEVER and 6 in VERINA. When subsequently given the NL descriptions and the previous proof attempts, Claude Code provided fixes for all of them and proved a substantial fraction of the corrected statements: Numina-Lean-Agent proved 15/17 and 5/6 of the fixed CLEVER and VERINA statements respectively, while Claude Code with *lean4-skills* proved 9/17 and 2/6, and Aristotle 9/17 and 3/6. Most of the fixes addressed wrong specification logic (9 cases), implementation bugs (6) and missing preconditions (5), with a few off-by-one and boundary errors (3).

Table 4: Number of fixes per benchmark by kind.

	CLEVER	VERINA	Total
Wrong Specification Logic	7	2	9
Implementation Bugs	4	2	6
Missing Preconditions	3	2	5
Off-by-One/Boundary Errors	3		3
Total	17	6	23

B SETUP DETAILS

This appendix complements Section 2.2 with the status-flag protocol and the differences between the three agentic stacks.

B.1 COMMON PROTOCOL

All stacks work on the same Lean project, pinned to Lean v4.24.0 and the matching Mathlib release, with the same *lean-lsp-mcp* server and the same commit of *lean4-skills*. Each attempt starts a fresh agent on a fresh copy of the project; each entry receives a single attempt per task (pass@1).

B.2 STATUS FLAGS

At the end of each task, the agent is asked to reply with a single status flag. The base prompt, offers three flags:

- `OK`: task completed, no `sorry` remains, code compiles;
- `FAIL`: unable to complete, `sorry` remains or code does not compile;
- `ISSUE`: task is impossible due to bugs or incorrect assumptions, and rigorous arguments in support have been provided.

For SPEC_ISO, the prompt splits `ISSUE` in two:

- `MISMATCH`: equivalence is unprovable because the specifications differ, yet both are valid interpretations of the NL description and test cases; rigorous arguments in support have been provided;
- `ISSUE`: equivalence is unprovable because at least one specification is erroneous (contradicts the NL description or test cases); rigorous arguments in support have been provided.

In the multiple specifications run, the agent reports one such flag per candidate file.

In later PROOF_GEN runs, a prompt that offered `MISMATCH` was used as well, although for homogeneity we merged these flags into `ISSUE`. This is consistent with the flags' meaning, since in implementation tasks the specification is provided, so an implementation that cannot meet it is an error, not an alternative reading. All changes are tracked in the released records.

B.3 DIFFERENCES BETWEEN STACKS

Table 5: Structural differences between the three agentic stacks. All share the benchmark, prompts, Lean project, MCP server, and *lean4-skills* commit.

	Claude	GPT-5.4	Kimi
Agent framework	Claude Agent SDK	Codex CLI	OpenHands SDK
Model serving	vendor	vendor	open weights, local cluster
Skills	native plugin	native skill directory	native plugin
`/lean4:autoprove`	native command	expanded into the prompt	keyword trigger
Time budget	1 h	1 h	4 h
Context management	framework default	framework default	condenser at 99% of the context window

Commands in Codex. The Codex CLI has no notion of user-defined commands. For the proof tasks, we therefore expand `/lean4:autoprove` into its full instructions and embed them in the prompt, while the skills themselves are exposed natively.

Compliance check for Kimi. In early runs, Kimi often edited parts of the Lean file it was not allowed to change, for instance adding a hypothesis to the theorem it had to prove or rewriting a fixed specification, and sometimes dropped the tags that delimit the file's sections, which the evaluation harness then scores as `sorry`. Some of these edits turned failed attempts into reported successes. The OpenHands harness therefore validates the file after the agent's main turn and before asking for the status flag:*Check A (tags)*: every section tag of the original file is still present; *Check B (fixed content)*: the non-editable sections and the preamble are unchanged, no axiom was added, and no fixed signature was redeclared; *Check C (flag)*: the status reply is exactly one of the allowed flags.

On a violation of Check B, the harness restores the original text itself; on a violation of Check A, it asks the agent to restore the tags. In both cases a corrective prompt explains the violation, and the agent continues its attempt. Each check allows up to three retries; exhausting them records `FAIL` without a status turn. Check C re-asks for the flag up to three times, again falling back to `FAIL`.

C Detailed Breakdown of Results

This appendix collects the per-entry outcomes behind Section 3. Figures 4 and 5 show how the flags of each run of GPT-5.4 and Kimi are distributed, in the same format as Claude's Figure 2. Figure 6 lists every run-level flag of the three stacks for all 161 entries, next to Claude's merged annotation, so that agreements and disagreements between stacks can be read entry by entry; Appendix D quantifies them. Figure 7 details Claude's outcomes, including each candidate of the multiple specifications run. In both grids, letters give the diagnosed source of an ISSUE (in SPEC_ISO, P, G, B for the reference, generated, or both specifications; in PROOF_GEN, S, I, B for the specification, the implementation, or both; $*$ for the fixed benchmark components), and symbols the direction of a MISMATCH (>, <, · for a generated specification stronger than, weaker than, or incomparable with the reference).

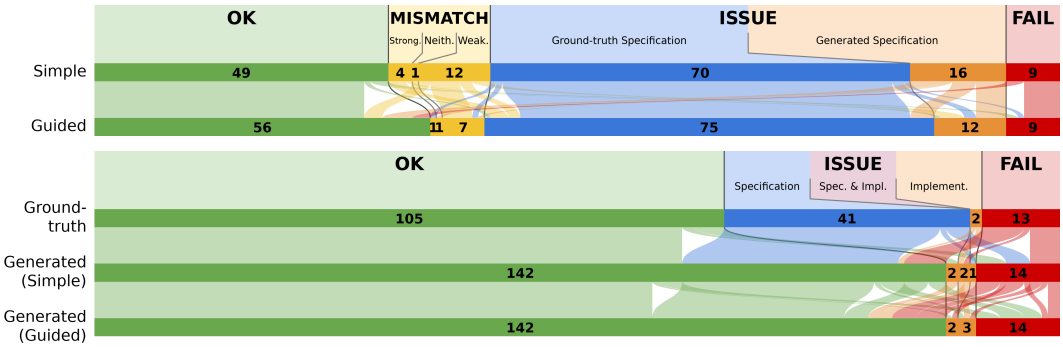

Figure 4: Distribution of GPT-5.4's output flags for the simple and guided SPEC_ISO runs (top) and for PROOF_GEN against the reference and against its simple and guided generated specifications (bottom); bands follow each entry from one run to the next. Colors as in Figure 2; FAIL includes timeouts. GPT-5.4 did not take part in the multiple specifications run, and its PROOF_GEN MISMATCH flags are merged into ISSUE (Appendix B.2).

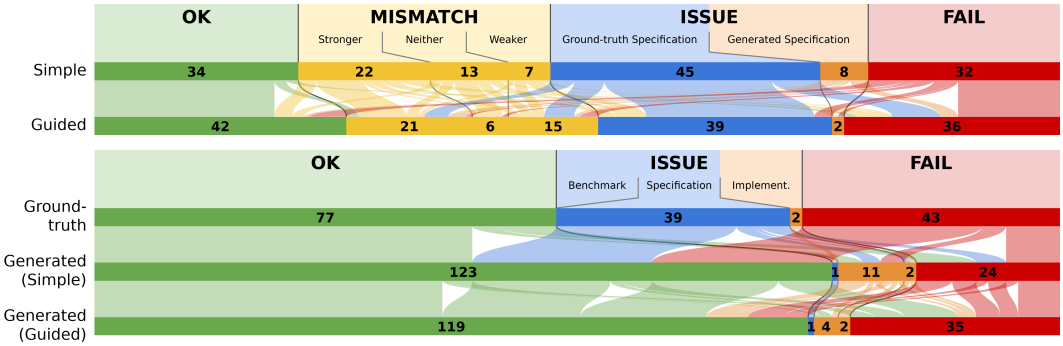

Figure 5: Distribution of Kimi's output flags for the simple and guided SPEC_ISO runs (top) and for PROOF_GEN against the reference and against its simple and guided generated specifications (bottom); bands follow each entry from one run to the next. Colors as in Figure 2; FAIL includes timeouts. Kimi did not take part in the multiple specifications run, and its PROOF_GEN MISMATCH flags are merged into ISSUE (Appendix B.2).

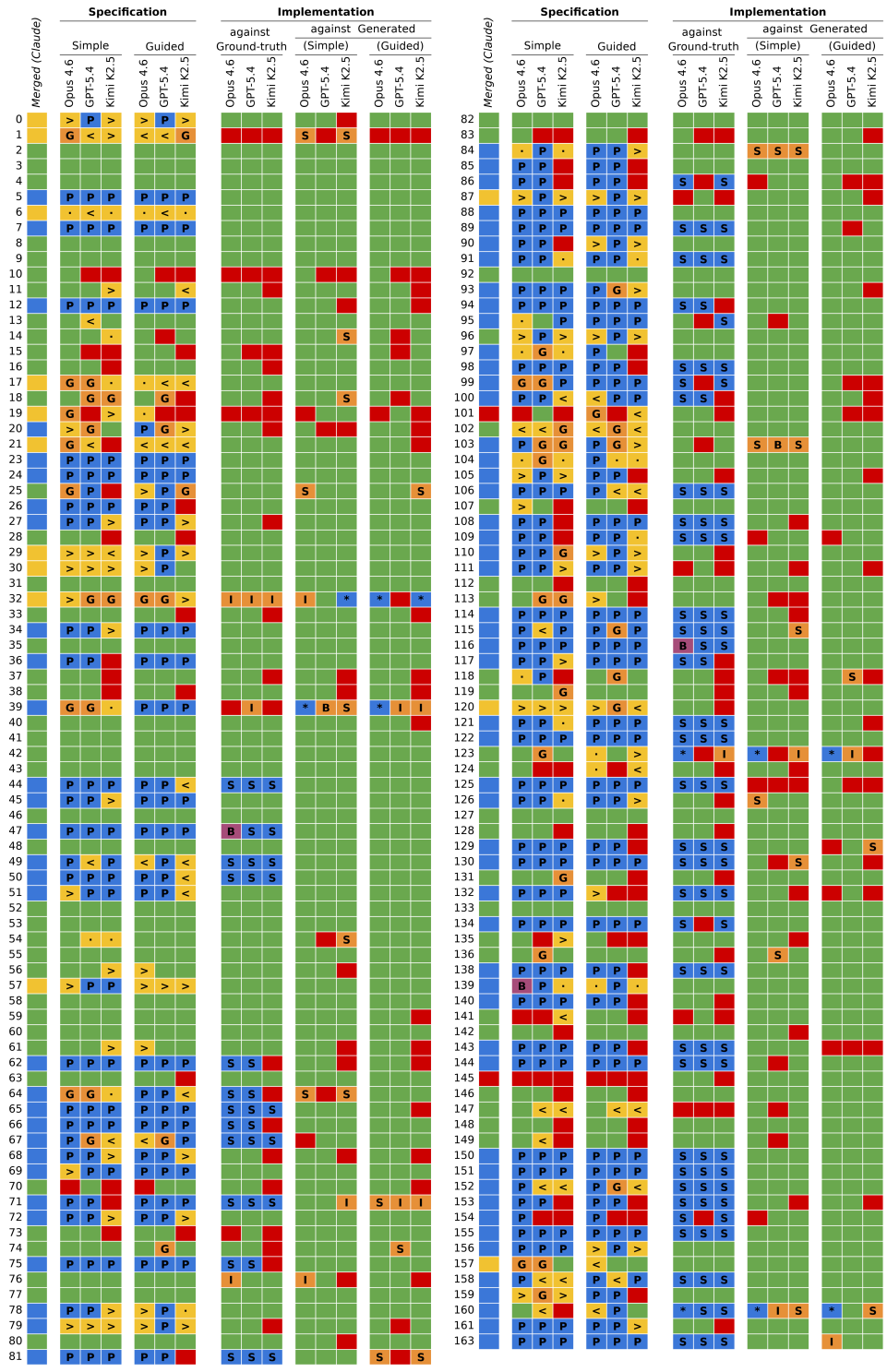

Figure 6: Per-entry flags of the three stacks over all 161 CLEVER entries: SPEC_ISO under the simple and guided prompts, and PROOF_GEN against the reference and against each stack's simple and guided generated specifications. The leftmost column is Claude's merged annotation. Colors as in Figure 2; red includes timeouts. Letters give the source of an ISSUE: P, G, or B for the reference, the generated, or both specifications in SPEC_ISO; S, I, or B for the specification, the implementation, or both in PROOF_GEN; an asterisk marks a defect in fixed benchmark components. Symbols give the direction of a MISMATCH: $>$ for a generated specification stronger than the reference, $<$ for weaker, $\cdot$ for neither.

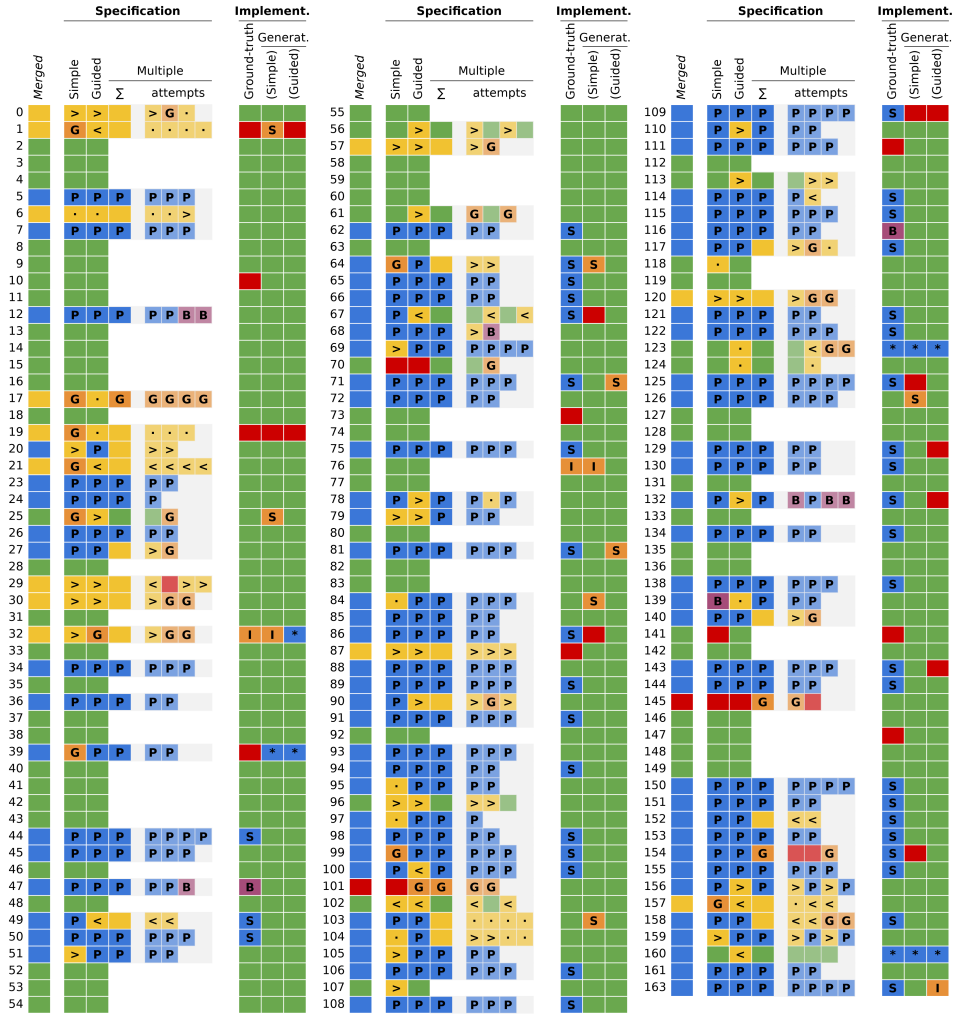

Figure 7: Claude's per-entry outcomes. Specification columns: SPEC_ISO under the simple and guided prompts, and the multiple specifications run, both aggregated per entry (Σ) and per candidate (lighter shades); entries already OK in an earlier run were not re-tested and are left blank. Implementation columns: PROOF_GEN against the reference and against the simple and guided generated specifications. The leftmost column is the merged annotation. Colors, letters, and symbols as in Figure 6.

D Consistency Analysis

This appendix checks how consistently the audit's central verdict, that a reference specification `problem_spec` is flawed, recurs across prompt runs, across tasks, and across the three stacks. We call *reference-bug flag* an ISSUE attributed to `problem_spec` (sources PROB or BOTH) in a SPEC_ISO run, or to the specification (SPEC or BOTH) in PROOF_GEN against the reference; proofs against generated specifications never count. The *flawed entries* are the 81 entries annotated ISSUE in the merged annotation (Section 3.1); the other 80 form C_{spec}. We write S and G for the simple and guided SPEC_ISO runs, M for Claude's multiple specifications run, where an entry counts as flagged if any of its candidates is, and PG for PROOF_GEN against the reference.

Since the merged annotation derives from Claude's runs, Claude's agreement with it is not independent evidence; we report it only where it describes the stability of those runs. GPT-5.4 and Kimi share prompts, benchmark, and Lean tooling with Claude, but neither model nor agent framework, and their flags do not contribute to the merged annotation. Agreement is measured with unweighted Cohen's κ, association with a one-sided exact Fisher test, and uncertainty with exact 95% Clopper–Pearson intervals; all p-values are nominal, without correction for multiple testing.

D.1 Across runs

Table 6: Reference-bug flags in the simple (S) and guided (G) SPEC_ISO runs, over all 161 entries. +/−: flagged or not in each run. Claude's either-run counts are omitted, since the annotation derives from them.

	S+G+	S+G−	S−G+	S−G−	agreement	κ	either run, on flawed / C_{spec}	
Claude	58	9	12	82	0.87	0.73	–	–
GPT-5.4	65	5	10	81	0.91	0.81	72	8
Kimi	28	17	11	105	0.83	0.55	55	1

Reference bugs. The reference-bug verdict is stable across prompts for Claude and GPT-5.4, and moderately so for Kimi (Table 6). Of the 81 flawed entries, Claude flags 73 in at least two of its three runs and 51 in all three.

Mismatches. Claude's MISMATCH verdicts are stable as well: 12 of the 13 MISMATCH entries receive that flag in at least two runs. The exception, entry 17, has a single MISMATCH run, its other runs blaming Claude's own specifications. When both S and G report a mismatch, its direction agrees for Claude (10 of 10 entries) and GPT-5.4 (5 of 5), but only on 14 of 22 entries for Kimi.

Full outcomes. The complete outcome of a run is less stable than the bug verdict alone. Over five classes (accepted, MISMATCH, reference bug, own-specification bug, failure), S and G agree with $\kappa = 0.64$, 0.68, and 0.48 for Claude, GPT-5.4, and Kimi; excluding entries with a failure in either run raises Kimi's to 0.60.

Contradictions. Within the SPEC_ISO runs, only two entries receive, from the same stack, both a proved isomorphism and a reference-bug flag. On entry 95, GPT-5.4's simple specification reproduces the reference bug, which the test cases of the guided run then exposed. On entry 67, Claude's multiple specifications run proved a vacuous isomorphism (Section D.3). A third case, Kimi on entry 160, pairs an isomorphism through the flawed helper with a reference-bug flag raised only at the proof stage.

D.2 Across tasks

Proof-stage flags. Reference bugs raised during PROOF_GEN never contradict the annotation: for every stack, each one falls on a flawed entry, including the few that no SPEC_ISO run had flagged (4 for GPT-5.4, 7 for Kimi).

Table 7: Reference-bug flags at the specification stage (S or G) and at the proof stage against the reference (PG).

	flagged entries (on flawed)		S/G flags	PG flags on	PROOF_GEN OK
	S or G	PG	also in PG	flawed entries	on flawed entries
Claude	79 (–)	44 (44)	44/79	44/44	34
GPT-5.4	80 (72)	41 (41)	37/80	41/41	33
Kimi	56 (55)	39 (39)	32/56	39/39	24

Coverage. The proof stage alone, however, surfaces only about half of the flawed references (Table 7). On most of the others, the stacks certify an implementation against the flawed specification: since the specification is visible at generation time, an implementation can satisfy it even when it misstates the intent. This happens more often when the bug weakens the specification. Taking as weakening the types *missing constraint*, *missing edge-case clause*, *vacuous disjunct*, and *trivial existential* (Appendix E), Claude certifies 13 of 17 such entries against 21 of the other 64 (one-sided Fisher $p = 0.0015$), GPT-5.4 13 against 20 ($p < 0.001$), and Kimi 8 against 16 ($p = 0.07$). This analysis is exploratory, as the direction is inferred from the bug type rather than annotated per entry. Specification-level comparison, rather than implementation proof, is thus what exposes most reference bugs.

D.3 Across provers

Table 8: Reference-bug flags (S, G, or PG) against the merged annotation, over all 161 entries. Sensitivity over the 81 flawed entries; false positives (FP) over the 80 entries of C_{spec}. Claude is omitted, since the annotation derives from its runs.

	sensitivity	95% CI	FP	κ	Fisher p
GPT-5.4	76/81	[86.2, 98.0]	8	0.84	$4.7 \cdot 10^{-30}$
Kimi	62/81	[65.8, 85.2]	1	0.75	$3.0 \cdot 10^{-26}$
GPT-5.4 ∪ Kimi	77/81	[87.8, 98.6]	8	0.85	$3.4 \cdot 10^{-31}$

Re-attribution. Counting flags from S, G, and PG, GPT-5.4 and Kimi flag 76 and 62 of the 81 flawed entries, and together 77 (95.1%, Table 8); from S and G alone, they flag 72 and 55 (Table 6). All three stacks flag 60 of the flawed entries. Measured between stacks rather than against the annotation, agreement on the bug verdict is $\kappa = 0.81$ between Claude and GPT-5.4, 0.75 between Claude and Kimi, and 0.72 between GPT-5.4 and Kimi.

False positives. All 8 flags on C_{spec} come from GPT-5.4, one of them also raised by Kimi (entry 57). Five fall on MISMATCH entries (0, 29, 30, 57, 87) and three on OK entries (25, 96, 118), and none is corroborated by the flagging stack's own PROOF_GEN, which certifies an implementation against the same reference. Most false positives thus sit at the boundary between ISSUE and MISMATCH. Since neither Claude nor, with one exception, Kimi flags these entries, they point to GPT-5.4 being quicker than the other stacks to call a defensible reading a bug.

Accepted isomorphisms on flawed entries. No stack proves a flawed reference valid. Seven runs have an isomorphism accepted on a flawed entry, and inspection of each proof shows why: three generated specifications reproduce the reference bug (GPT-5.4 on entries 95 and 97, Kimi on entry 20); on entry 160, the specifications of Claude and Kimi reuse the flawed helper; and Claude's multiple specifications run proved two *vacuous* isomorphisms, on entries 67 and 160, where both specifications are unsatisfiable on some input for every implementation. CLEVER's isomorphism theorem, of the form $\forall f.\,(\forall x.\,P(f,x)) \leftrightarrow (\forall x.\,Q(f,x))$, accepts any such pair, so an accepted specification is not, on its own, evidence of validity.

Misses. On the 4 flawed entries flagged by neither GPT-5.4 nor Kimi (20, 97, 103, 104), every SPEC_ISO run returns MISMATCH, blames the stack's own specification, fails, or reproduces the

bug. Of Kimi's 19 misses, 18 receive at least one MISMATCH reading instead. This boundary is also where the stacks diverge most: only 8 of the 13 MISMATCH entries receive a MISMATCH reading from all three, and GPT-5.4 flags 5 of them as reference bugs. Over the five outcome classes, pairwise κ between stacks ranges from 0.39 to 0.61, against 0.72–0.81 for the binary bug verdict.

Failures. Capability differences surface mainly as failures, and only rarely as contradictions: runs end with FAIL or a timeout in 27, 59, and 170 of the 805 certification attempts (161 entries times S, G, PG, and the two proofs against generated specifications) for Claude, GPT-5.4, and Kimi.

E TAXONOMY

Autoformalization is hard to evaluate. Beyond producing a Lean specification that compiles, two substantive issues dominate: choosing the right *strength* for the specification (loose enough not to over-constrain valid implementations, yet tight enough to rule out incorrect ones), and committing to a single *interpretation* of an inherently ambiguous NL description. Problem 124, for instance, requires dates in the `mm-dd-yyyy` format but leaves implicit whether single-digit months and days should be accepted; and the ground-truth specification of problem 27 only requires the output characters to have the opposite case of the corresponding input ones, an uncommon but defensible reading of the prose. In this sense, the central challenge in autoformalizing NL sources is that of *formalizing the intent* behind them, bridging the gap between the informal human specification and its formal counterpart (Lahiri, 2026): even direct human evaluation can leave subtle bugs unnoticed. Claude's downstream PROOF_GEN attempts sometimes surfaced generated-specification defects that SPEC_ISO had missed, because a successful equivalence check alone does not guarantee fidelity to the NL task. The distinction between passing a formal check and formalizing the intended behavior is the main focus of the audit.

Table 9: Taxonomy of specification ISSUE types.

Type	Description
Vacuous disjunct	$\vee$ (often a biconditional disjunct like `result = none` $\leftrightarrow$ `list = []`) trivially satisfied, admitting trivial implementations.
Precedence bug	Missing parentheses; $\wedge$/$\rightarrow$/$\leftrightarrow$ precedence makes the spec parse differently than intended (e.g. $\wedge$ binds tighter than $\rightarrow$, making the per-element constraint vacuous).
Conjunct used as guard	`(P` $\wedge$ `...)` used where `(P` $\rightarrow$ `...)` was intended, making the spec unsatisfiable for invalid inputs (or, vice versa, vacuous).
Trivial existential	$\exists$ or $\exists!$ trivially satisfied (e.g. `∃! S, S.card = 0` witnessed by $\emptyset$; `∃ v : ℚ, |v - n| = 0` by `v =` $\uparrow$`n`).
Universal over invalid domain	Universal quantifier ranges over inputs where the body is unsatisfiable (e.g. over all `Int n` requiring `length = n` for negative `n`; over all bases including 0).
Missing edge-case clause	Missing clause for a particular edge case (`n = 0`, empty list, single element, all-equal, all-odd, etc.).
Missing constraint	Missing a non-edge constraint required by the NL description or test cases (positivity, range, ordering, format, multiplicity, sublist, stability for ties, etc.).
Extra/over-restrictive constraint	Extra restriction not in the NL description or test cases (e.g. extra `result.data.all isLower` precondition, tolerance too tight).
Nat arithmetic truncation	Lean `Nat` truncation: subtraction floors at 0 and division floors, making the spec lossy.
Out-of-bounds default access	`getElem!`/`get!` returns a default value out of bounds, weakening order, equality or index constraints.
Type-system mismatch	`Nat`/`Int` coercion or other type-induced unsatisfiability.
Wrong semantics	Wrong formula, operation, predicate or filter, or structural mismatch (membership vs. count vs. sublist; midrange vs. median; backwards divisibility; off-by-one threshold; `String.toLower` vs. `Char.isLower`; etc.).

Ground-truth specifications. As stated in Section 3, 80 of the 161 ground-truth `problem_spec` were flagged at least once as having bugs. We then had Claude classify the bugs arising in each ISSUE entry across the three SPEC_ISO runs and the PROOF_GEN run from ground-truth specifications, obtaining a comprehensive taxonomy of 12 error types; for 5 of those problems (numbers 81, 99, 109, 130, 139), two types of issue were identified. Most issues cluster on *Lean-encoding* pitfalls: *conjunct-as-guard* patterns using $(P \wedge Q)$ in place of $(P \rightarrow Q)$ (15 problems); *precedence bugs* from omitted parentheses around $\wedge, \rightarrow, \leftrightarrow$ (12); universal quantifiers over inputs whose body is unsatisfiable (7); plus smaller counts of lossy `Nat` arithmetic, default-value access via `getElem!`/`get!`, type-system mismatches, vacuous disjuncts, and trivial existentials. Together, these Lean-language hazards affect 48 of the 80 problems; the remaining 34 carry errors in the specification semantics: *over-* or *under-*

restrictive constraints (10 problems), *missing edge-case clauses* (4), and, most prominently, overall wrong interpretations of the problem departing from the NL description (21). See Table 10.

Table 10: Frequencies by type of ground-truth specification issues identified across the Simple, Guided, Multiple runs of SPEC_ISO and the PROOF_GEN run from ground-truth specs. Asterisks count the entries that were flagged as ISSUE in both specs (SPEC_ISO) or in spec and impl (PROOF_GEN); superscripts mark entries classified under multiple types via the partner type's initial.

Type	Simp.	Guid.	Multi	PROOF	**Problem IDs**
Conjunct used as guard	14	13	29	$9{+}1^{O}$	24, 44, 49, 50, 72, 81, 84, 110, 115, 122, 140, 150, 152, 158, 163
Precedence bug	10	11	29^{***}	3	23, 45, 51, 69, 75, 85, 88, 93, 109, 126, 132, 154
Universal over invalid domain	7	6	16	5	7, 67, 111, 125, 130, 143, 155
Nat arithmetic truncation	3^{*}	2	8	2	36, 109, 139, 153
Out-of-bounds default access	2	3	7	$2{+}1^{C}$	26, 64, 66, 81
Type-system mismatch	3	2	8	3	100, 106, 151
Vacuous disjunct	3	3	7^{***}		12, 68, 161
Trivial existential	1	2	5		34, 39
Wrong semantics	18	19	38^{*}	15^{**}	27, 47, 65, 86, 89, 90, 91, 94, 95, 98, 99, 103, 108, 114, 116, 117, 121, 129, 130, 134, 144
Missing constraint	1	5	12	2	20, 79, 97, 99, 105, 138, 139, 159
Extra/over-restrictive constraint	2	1	5	1	71, 156
Missing edge-case clause	3	3	7	1	5, 62, 78, 104

Generated specifications. In contrast, Claude-generated specifications fail mainly in a *semantic* register, misstating the relation or under-specifying it, and rarely in a *syntactic* one (only two instances: a *trivial existential* and a *type-system mismatch*). Within the semantic class, *missing constraints* are more frequent than *wrong semantics*: a different failure profile from the ground-truth specifications. Per-run counts expose two phenomena. First, guidance is effective: the design guidelines instructed Claude on the policy for imposing constraints, producing a five-fold drop in issues (from 10 to 2) and the complete disappearance of *missing-constraint* and *over-/under-restrictive-constraint* errors. Second, those errors reappear in the multi-specification run despite the same guidelines: beyond the mechanical multiplication of surface area from more generated specifications per problem, this also reflects Claude's intentionally more "adventurous" generation, attempting to cover as much of the "interpretation space" as possible. See Table 11.

Table 11: Frequencies by type of Claude-generated specification *semantic* issues only across SPEC_ISO runs as in Table 10 and the PROOF_GEN runs from generated specs (Simple and Guided prompts). Asterisks and single-letter superscripts formatted similarly to Table 10.

Type	Simple	Guided	Multiple	PROOF ($\rightarrow$ Simple)	PROOF ($\rightarrow$ Guided)
Wrong semantics	2	2	$8^{**}{+}3^{M*}$	7^{**}	1
Missing constraint	5^{*}		$15^{****}{+}1^{E}{+}3^{W*}$	1	
Extra/over-restrictive constraint	2		$2{+}1^{M}$		

Mismatches. The distribution of MISMATCH entries, where neither specification is buggy but the two are not isomorphic, reflects *authoring choices* rather than errors. Claude's generated specifications more often impose stronger constraints than the references, though guidance narrows this skew. Across the reviewed runs, 73 taxonomy assignments concern the boundary of the input domain (gaps in preconditions, edge-case handling); 37 concern conditions imposed on the function output (ordering of lists vs. multisets, canonical-form impositions); 22 concern structural and definitional choices; and a minority of 10 are semantic in nature, e.g. different but mathematically defensible formulations, or quantification over indices instead of values. An entry can receive multiple assignments. See Table 12.

Table 12: Mismatch frequencies by type and direction across the SPEC_ISO runs. Direction symbols: $>$ generated stronger than reference, / neither, $<$ generated weaker. Single-letter superscripts are formatted similarly to Table 10.

	Simple			Guided			Multiple		
Type	$>$	/	$<$	$>$	/	$<$	$>$	/	$<$
Precondition gap	2	1^{O}	1	7	$2^{S}{+}1^{T}$	5	$11{+}1^{S}{+}1^{T}$	$2{+}3^{O}{+}2^{C}{+}2^{S}$	$7{+}1^{S}$
Edge-case behavior	4			5		2	$2{+}1^{A}$	1	9
Ordering vs multiset	5	1^{P}		4			10	$3^{P}{+}1^{S}$	2
Canonical form	1			1	1		3	$3{+}2^{P}$	
Structural-op choice	1	1			$1{+}2^{P}$		$1{+}1^{P}$	$6{+}2^{P}{+}1^{O}$	1^{P}
Type-system semantics		$2{+}1^{A}$			1^{P}		1^{P}		
Algorithmic variant	1	1^{T}			1		$1{+}1^{E}$		
Quantifier scope	1					1	2		
Adjacency		1							

Table 13: Taxonomy of specification MISMATCH types.

Type	**Description**
Edge-case behavior	The disagreement is on a small, finite or enumerable set of edge inputs (`n=0`; empty list; singleton; `p=0` in division; min = max; all-equal). The natural fix is to add 1–2 specific case clauses.
Precondition gap	The disagreement is on a broad or infinite class of inputs characterised by a property (`n < 0`; `length > 1000`; non-paren chars; non-digit chars; invalid operator; non-lowercase output). The natural fix is to remove or generalise a domain restriction.
Quantifier scope	Different quantifier domain: $\forall$ over distinct values vs. distinct indices; existential parsing vs. universal verification; $\exists!$ over a set vs. a sum over a multiset.
Ordering vs multiset	One spec requires a specific order, exact list, or `result = list.filter . . .` ; the other allows permutation, multiset, set, or membership semantics.
Canonical form	One spec requires a unique canonical representation; the other allows equivalents (no leading zeros, normalised form, fixed-width vs. variable-width formatting, single vs. multiple representations of the same value).
Structural-op choice	Different filter, split or recursion choices (`split` vs. `splitOn` + filter; with vs. without empty-token filtering; preserving vs. collapsing duplicates) that diverge on edge inputs.
Type-system semantics	Different choice of Lean type-system semantics: `Int.mod` Euclidean vs. `Nat.mod` vs. `natAbs`; `Rat`-field vs. `Int`-integer arithmetic; `Nat`-truncation vs. `Int`-exact; `String.toLower` vs. `Char.isLower`.
Algorithmic variant	Different but mathematically defensible formulation: binary digit sum vs. decimal digit sum; midrange vs. median; threshold `n` $\geq$ `2` vs. `n > 2`; different mathematical algorithm.
Adjacency	Adjacent or consecutive positions vs. arbitrary indices (consonant–vowel–consonant requiring `(i-1, i, i+1)` vs. `i < j < k`).

Implementations. Claude detected 7 issues in the generated implementations, on only 5 problems, across the three PROOF_GEN runs. Three involve hard-coded *bounded fuel* (e.g. the 200-iteration Newton–Raphson loop in problem 32, see Appendix H) that does not suffice for the arbitrarily large inputs that the specification quantifies over; the remaining four are off-by-one indexing errors (problems 47 and 163), a missing case (a stationary point for Newton's method in problem 32 again), and a single wrong-algorithm choice (problem 116: the implementation sorts numerically, while the specification requires sorting by binary digit count).

Benchmark. Four problems carry issues in the fixed, non-editable parts of the benchmark itself, which neither the generated specification nor the generated implementation can repair. Problem 123 asks for a `collatz` iterator: any proof of correctness against the formal specification essentially requires a proof of the Collatz conjecture itself; problem 39 (`prime_fib`) similarly hinges on the open question of the existence of infinitely many Fibonacci primes; and problem 160 (`do_algebra`) ships a fixed helper `applyOp` that returns a `none` on division by zero, which is not parsed by other

Table 14: Taxonomy of implementation ISSUE types.

Type	Description
Bounded fuel	Implementation uses a fixed iteration count or fuel insufficient for arbitrary inputs (e.g. 200 Newton iterations, 64 power checks, 1000 Collatz steps). Fixable by adding more fuel or unbounded recursion.
Off-by-one	Index or range off-by-one error.
Nat truncation	`Nat` subtraction floors at 0 or division truncates, causing wrong implementation behavior.
Out-of-bounds default access	Implementation uses `get!`/`getElem!` returning a default value outside bounds.
Wrong algorithm	Implementation encodes a fundamentally different algorithm than the spec demands.
Missing case	Implementation does not handle some edge case (e.g. stationary point in Newton's method).
Wrong formula	Wrong arithmetic or formula in the implementation.

parts of the script. A notable example, Problem 32, exhibits a missing rational-root precondition, and highlights other issues and phenomena observed in the results; it is discussed in detail in Appendix H.

F Resources

This appendix reports how the three stacks spent time, tokens, and money. Formalization tasks are SPEC_GEN and IMPL_GEN; proving tasks are SPEC_ISO and PROOF_GEN.

Time. Wall-clock time was spent mostly on model inference, not on client-side work such as tool calls, Lean builds, and LSP queries (Table 15). On proving tasks, tool execution took 20% of the total wall-clock time for Claude and GPT-5.4, and 6% for Kimi; on formalization tasks, 12%, 29%, and 22%. For GPT-5.4 and Kimi, whose traces time each tool call, nearly all of it is Lean tooling. Inference speed thus set the pace, and it differed with the serving: Claude and GPT-5.4 ran through their vendors' APIs, Kimi on a local cluster. On proving tasks, a median Kimi turn took 27 s against 12 s for Claude and 7 s for GPT-5.4, and Kimi also needed more turns. Its median proving attempt lasted 47 minutes, against 7 and 6 minutes for the other two, which is why it received a 4-hour budget instead of 1 hour; even so, 9.6% of its proving attempts reached the timeout. Formalization tasks took about a minute and a half on every stack.

Table 15: Time per attempt, by stack and task family: median wall-clock time, share of wall-clock time spent executing tools (summed over attempts), and, for proving tasks, median turns, median inference seconds per turn, and share of attempts reaching the timeout (1 h for Claude and GPT-5.4, 4 h for Kimi).

	median wall-clock		tool share		proving		
	formal.	proving	formal.	proving	turns	s/turn	timeouts
Claude	83 s	6.7 min	12%	20%	27	11.7	2.8%
GPT-5.4	85 s	5.6 min	29%	20%	36	6.9	0%
Kimi	91 s	47 min	22%	6%	83	26.6	9.6%

Tokens. Token usage mirrors time (Figure 8). A typical attempt of Claude or GPT-5.4 reads between 0.6 and 1.7 million input tokens, mostly from the prompt cache, and writes 7 to 28 thousand output tokens (medians over the shared prompt regimes), with the generation stage accounting for a small share of both. Kimi's medians are several times higher on the specification runs and on proofs against the reference (5.8 and 5.3 million input, 350 and 235 thousand output tokens), but the gap comes from its failures, not its successes. Its successful attempts use about as many tokens as those of the other stacks (0.3 to 1.9 million in total, depending on the task), while its failed attempts, which run until the 4-hour budget and make up 15–27% of its runs, consume 17 to 20 million each. Failed attempts are the most token-intensive outcome for every stack.

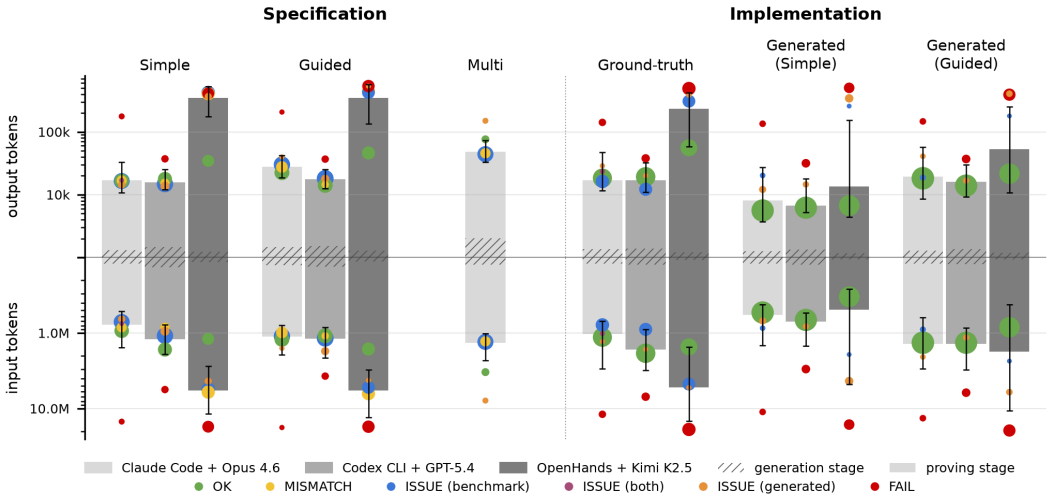

Figure 8: Median tokens per attempt, by stack and prompt regime, for specification certification (SPEC_GEN and SPEC_ISO) and implementation certification (IMPL_GEN and PROOF_GEN, against the reference or against the stack's simple or guided generated specification). Output tokens grow upward and input tokens, including cached ones, downward, each on its own logarithmic scale; error bars span the 25th to 75th percentiles. The hatched segment near the axis shows the generation stage's share of the total, not its absolute count. Dots mark the median of each outcome class, sized by its frequency. The multiple specifications run (Claude only) covers 105 entries. Kimi's endpoint does not report cached tokens; nine Claude records without complete token counts are excluded.

Costs. Only the Claude runs report costs: the GPT-5.4 harness did not provide them, and the local cluster serving Kimi is not billed per token. As reported by the Claude Agent SDK, costs per attempt grew with runtime (Figure 9) and, more tightly, with token usage: a median of $0.25 for formalization tasks ($0.49 for the multiple specifications run) and $1.17 for proving tasks, whose mean of $2.26 reflects a long tail of slow attempts. Attempts that reached the 1-hour timeout cost about $11 each.

G CONTAMINATION

CLEVER was public well before the release of the models in our three stacks: its repository since February 2025 and its paper since May 2025. Training exposure to the benchmark therefore cannot be fully excluded, and it is impossible to measure its effect on any stack directly. We argue instead that the audit's findings do not depend on it, on three grounds, and report an exploratory control with a model trained before CLEVER's release.

Little formal material to memorize. The algorithmic content of CLEVER is public by construction: the HUMANEVAL problems (Chen et al., 2021) and their Python solutions have circulated since 2021, and the benchmark ships reference solutions in Python and Rust. Its formal layer offers far less. Besides the 161 reference specifications, it includes Lean reference implementations for 43 entries and complete correctness proofs for only 4 (entries 0, 3, 17, and 72); no isomorphism proof exists for any entry. Most SPEC_ISO and PROOF_GEN targets thus have no public solution to recall.

Memorization would work against the audit. A model that recalled the public reference specifications would reproduce them: its specifications would be trivially isomorphic to the references and would inherit their bugs. We observe the opposite. Only 65 of 161 entries end with an *accepted* specification in the merged annotation, and 81 references are flagged as flawed, each with a supporting argument, typically a concrete counterexample, that we can check independently of how the model found it. Exposure could make *accepted* specifications easier to obtain; it cannot explain why the models reject the references they would have memorized.

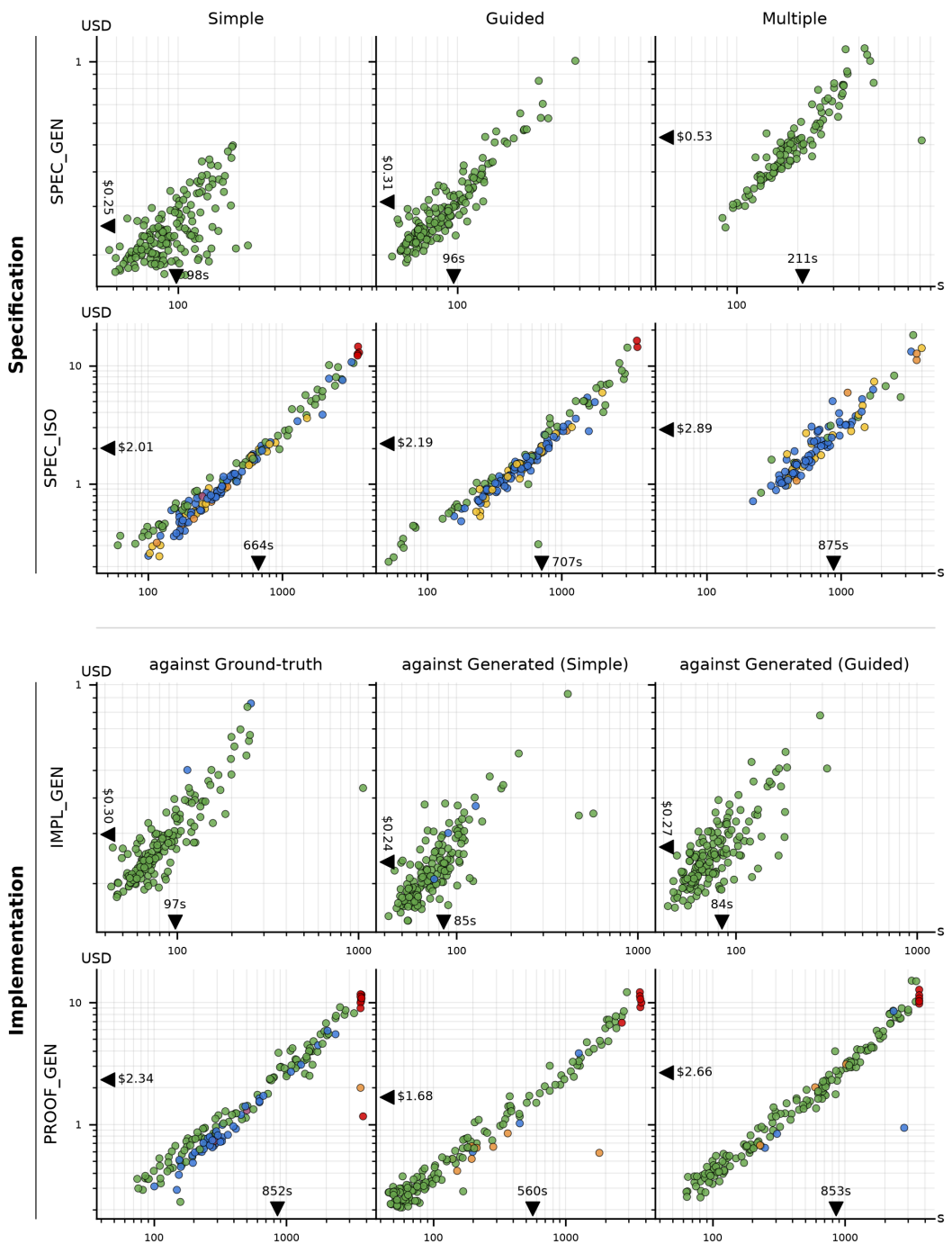

Figure 9: Cost (USD) against runtime (s) of Claude's attempts, across all tasks and runs. Markers follow the color scheme of Figure 2; labeled black triangles mark the average cost and runtime of each run.

Independent replication. GPT-5.4 and Kimi, which share neither model nor agent framework with Claude, re-attribute to the reference 77 of the 81 flaws (Appendix D). A flaw found by several independently trained models, and certified by a machine-checkable argument, is a property of the benchmark rather than of one model's training data.

Pre-cutoff control. A direct control requires a model trained before CLEVER became public that is also strong enough in our agentic setup for its score to be informative: a weak model scoring low cannot distinguish the absence of contamination from the lack of capability. No currently served Claude model meets the first requirement. We first ran Qwen3-235B-A22B, with training cutoff likely predating the release of CLEVER's repository, in the OpenHands harness, but the model proved too weak to be informative and we abandoned it.

We then turned to GPT-5.1, whose vendor-stated knowledge cutoff (September 30, 2024) predates CLEVER's release, while it belongs to the same model family as GPT-5.4. We ran it on all ten tasks of a 16-entry pilot (10% of dataset), drawn uniformly at random with a fixed seed, in the same Codex CLI harness, with the same prompts and settings as GPT-5.4. The pilot contains 9 flawed entries, 5 annotated OK, and 2 MISMATCH (Figure 10).

- **GPT-5.1 reproduces the audit's central finding.** It flags the reference as flawed on 7 of the 9 flawed entries and on none of the other 7 (Cohen's $\kappa = 0.75$ against the annotation; one-sided Fisher $p = 0.003$). Its two misses (entries 20 and 105) are MISMATCH readings, and it never proves a flawed reference valid.
- **GPT-5.1 closes fewer proofs.** It succeeds on 31 of the 80 certification attempts (16 entries $\times$ five certification runs) against 52 for GPT-5.4, which is ahead on 10 entries and behind on none (sign test $p = 0.002$); end-to-end success is 11 against 15 of 16.

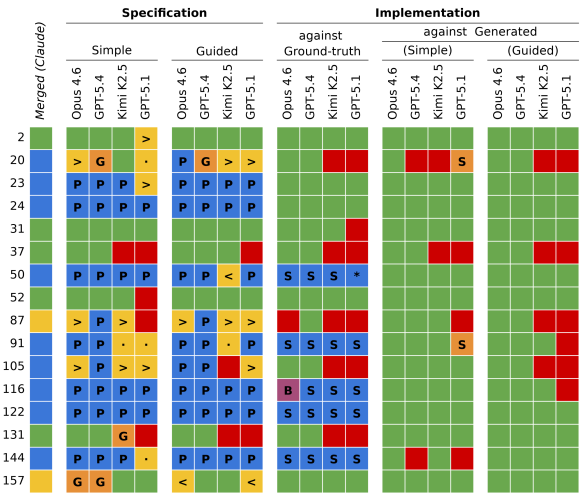

Figure 10: Per-entry outcomes of the 16-entry pilot for the three stacks and the pre-cutoff GPT-5.1, over the simple and guided SPEC_ISO runs, PROOF_GEN against the reference, and PROOF_GEN against the simple and guided generated specifications (last two groups). The leftmost column is Claude's merged annotation. Colors and symbols follow Figure 2.

None of these arguments proves the absence of contamination, and exposure could still help the main stacks, for instance by making the references' formulation style familiar. Such help would raise the counts of *accepted* specifications and certified implementations; it would not produce the flaws, mismatches, and counterexamples on which the audit rests.

H A Notable Example: Problem 32

Problem 32 deserves a separate look, both because it is arguably the hardest problem in the dataset and because it gathers in a single entry several of the phenomena observed throughout our evaluation. The task asks to find a real root of an odd-degree polynomial given by its list of coefficients.

Already in the preliminary experiments (Appendix A), the provided ground-truth implementation was found to be incorrect: it was based on the Newton–Raphson method, which is not globally convergent and can enter cycles even on simple polynomials. Following insight from the agent, it was replaced with a bisection method initialized from the Cauchy root bound, with bisection fuel adaptive in the bit-size of the coefficients, ensuring convergence to the target precision (see Figures 11 and 12).

```
17  def implementation (xs: List Rat) : Rat :=
18    let rec poly (xs: List Rat) (x: Rat) := xs.reverse.foldl (λ acc a ⇒ acc * x + a) 0;
  -   let rec poly' (xs: List Rat) (x: Rat) := (xs.drop 1).reverse.foldl (λ acc a ⇒ acc * x + a) 0;
19    let rec eps := (1: Rat) / 1000000;
  -   let rec find_zero (xs: List Rat) (guess: Rat) (fuel: Nat) :=
  -   let eval := poly xs guess;
  -   let eval' := poly' xs guess;
  -   if eval ≤ eps ∨ fuel = 0 then (guess, fuel)
  -   else
  -   let guess' := (eval' * guess - eval) / eval';
  -   find_zero xs guess' (fuel - 1);
  -   (find_zero xs 1.0 1000000).1
20+   let lc := |xs.getLast!|;
21+   let initBound : Rat := if lc = 0 then 1
22+     else 1 + xs.dropLast.foldl (fun acc a ⇒ acc + |a|) 0 / lc;
23+   let rec findBound (xs: List Rat) (b: Rat) (fuel: Nat) : Rat :=
24+     match fuel with
25+     | 0 ⇒ b
26+     | fuel' + 1 ⇒
27+       if poly xs b * poly xs (-b) ≤ 0 then b
28+       else findBound xs (b * 2) fuel';
29+   let rec bisect (xs: List Rat) (lo hi: Rat) (fuel: Nat) : Rat :=
30+     match fuel with
31+     | 0 ⇒ (lo + hi) / 2
32+     | fuel' + 1 ⇒
33+       let mid := (lo + hi) / 2;
34+       if |poly xs mid| ≤ eps then mid
35+       else if poly xs lo * poly xs mid ≤ 0 then bisect xs lo mid fuel'
36+       else bisect xs mid hi fuel';
37+   let bound := findBound xs initBound 100;
38+   let coeffBits := xs.foldl (fun acc a ⇒ acc + a.num.natAbs.size + a.den.size) 0;
39+   let fuel := 1000000 + coeffBits * (xs.length + 1) * 40;
40+   bisect xs (-bound) bound fuel
41
```

Figure 11: Claude Code's fix of the reference implementation of problem 32 in the preliminary experiments: bisection from the Cauchy root bound replaces Newton–Raphson.

At specification certification, no run produced a `generated_spec` that proved isomorphic to the ground-truth one. The reason was systematic: every generated specification required the implementation to return an *exact* root of the polynomial, whereas the ground-truth specification only required an *approximate* root within a fixed tolerance. The SPEC_GEN agent could not have inferred this from context, since neither the NL description nor the example usages mentioned approximation or any threshold for it. The corresponding entries were flagged either as MISMATCH or as ISSUE in `generated_spec`.

At implementation certification, no run succeeded either. Two factors compound: the intrinsic difficulty of the implementation (global convergence of the root-finding method, Newton–Raphson vs. bisection, finite fuel), already witnessed by the fix to the ground-truth implementation; and the function signatures imposed by the benchmark, which require a rational return type to sidestep the fact that `Real` is not a computable type in Lean (as a developer comment in the entry itself acknowledges). Odd-degree polynomials such as $x^3 + 2$ have a real root but no rational one, so under the imposed `Rat` type the task has no exact solution: any implementation can at best approximate the root. The PROOF_GEN attempts were accordingly flagged as ISSUE in either the

Here's a summary of the fix and why it's correct:

Root cause: Newton-Raphson is not globally convergent. Even with a correct derivative, it can enter cycles (as shown by [2, -2, 0, 1] cycling between 0 and 1). No starting point is safe for all polynomials.

Fix: Replaced Newton-Raphson with the bisection method, which is globally convergent:
1. findBound: Finds a bound b (starting from 1, doubling up to 2^{100}) such that poly(b) and poly(-b) have opposite signs. This is guaranteed to exist for odd-degree polynomials (even-length coefficient list) with non-zero leading coefficient — exactly the docstring's preconditions.
2. bisect: Standard bisection on [-bound, bound] with 1,000,000 iterations. Each step halves the interval containing the root. After n steps, the interval has width $2 \cdot \text{bound} / 2^n$, giving astronomically precise convergence — far more than needed for eps = 10^{-6}.

Verification against test cases:
- [1, 2] (1 + 2x): bisect finds mid = -0.5 where poly = 0 exactly in 2 steps
- [-6, 11, -6, 1] ($x^3-6x^2+11x-6$): bisect converges to a value near root 1 with |poly| $\leq$ eps

The fix has two changes to the implementation (spec unchanged):

Fix 1 - Cauchy bound initialization (lines 20-22): Instead of findBound xs 1 100, we compute the Cauchy root bound $1 + \sum|a_i|/|a_i|$ as the starting value. This guarantees the initial bound is already past all roots, regardless of coefficient magnitude.

Fix 2 - Adaptive bisection fuel (lines 38-39): The bisection fuel scales with the total bit-size of the coefficients (coeffBits * (xs.length + 1) * 40 + 1000000). This ensures enough precision even for polynomials with large/steep coefficients.

Figure 12: Extract of Claude Code's replies explaining its fix of problem 32.

implementation or the benchmark-imposed signatures. The same observation explains why logically correct specifications were flagged as ISSUE at SPEC_ISO time: the agent recognized that any specification asking for an exact root must fail on polynomials without rational roots, regardless of the implementation — a trap from which the ground-truth specification escapes precisely by asking for an approximate root.

Problem 32 thus condenses several of the difficulties this kind of benchmark exposes: a literal reading of the NL description leads to a specification incompatible with the structure imposed by the benchmark; reconciling the two requires a creative workaround (here, an approximation threshold); and the implementation itself calls for non-trivial mathematical insight.

I Complete Finding Summaries

This appendix lists the entry-level findings behind the aggregate counts in Section 4. Each row gives the benchmark entry identifier, the taxonomy term used in this paper, and a brief summary. The full description of each finding, with its supporting argument or counterexample as submitted for verification, is part of our release (`https://anonymous.4open.science/r/APPV-iclr27`). Entries 32 and 39 have distinct findings in two categories; their summaries are split accordingly. Entry 160 concerns a fixed helper rather than the ground-truth specification, so it appears only in the last table.

Table 16: Ground-truth specification ISSUE findings (80 entries).

ID	Taxonomy	Summary
5	Missing edge-case clause	Singleton branch wrongly appends delimiter to lone element
7	Universal over invalid domain	Guarded universal vacuously true, empty output satisfies spec
12	Vacuous disjunct	$\vee$ instead of $\wedge$ voids the longest-element constraint
20	Missing constraint	Minimality over distinct values misses equal-number zero-gap pairs
23	Precedence bug	$\wedge$ binds tighter than $\leftrightarrow$, length recurrence never enforced
24	Conjunct used as guard	Divisor requirement sits as guard, vacuous when result=0
26	Out-of-bounds default access	Unbounded order-witness indices let getElem! default admit fabricated elements
27	Wrong semantics	Constrains only output case category, not character identity
34	Trivial existential	count=1 inside biconditional lets empty list satisfy
36	Nat arithmetic truncation	Nat subtraction saturates, so 7-count tally can decrease
39	Trivial existential	n=1 vacuous finset imposes no minimality
44	Conjunct used as guard	Base-validity guard a conjunct not precondition, fails at base 0
45	Precedence bug	Missing parens make area constraint vacuous when a $\neq 0$
47	Wrong semantics	Even case computes midrange not median via swapped min/max sublists
49	Conjunct used as guard	Modulus positivity a conjunct not guard, unsatisfiable at p=0
50	Conjunct used as guard	isAlphabetic s a conjunct not guard, fails on non-alphabetic input
51	Precedence bug	Precedence guards consonant-preservation behind result.contains, empty string passes
62	Missing edge-case clause	Length xs.length-1 clause contradicts length-preserving derivative equality
64	Out-of-bounds default access	Empty string [0]! defaults to vowel A, making count unsatisfiable
65	Wrong semantics	Reverse branch tests input palindrome instead of forcing reversed result
66	Out-of-bounds default access	Empty string hits else, getElem! default forces 0=65
67	Universal over invalid domain	No precondition guards malformed input, existential unsatisfiable
68	Vacuous disjunct	Never asserts result length in {0,2}, admits wrong list
69	Precedence bug	-1 sentinel fused into biconditional, constant -1 satisfies spec
71	Extra/over-restrictive constraint	1/10000 tolerance on result^2 too tight for 2-decimal rounding
72	Conjunct used as guard	Only implications no converse, so false always satisfies
75	Precedence bug	$\rightarrow$ binds tighter than $\leftrightarrow$, misparses guard, fails at 101
78	Missing edge-case clause	0<length guard leaves empty string unconstrained, misses base case
79	Missing constraint	Never constrains inner digits to be binary bits
81	Conjunct used as guard Out-of-bounds default access	Range check as conjunct, result indexed by dict position
84	Conjunct used as guard	Binary-character check placed as guard not conjunct, spec vacuous
85	Precedence bug	Missing parens chain contradictory length guards, spec vacuously true
86	Wrong semantics	Char count indexes each word separately, permutation never enforced
88	Precedence bug	Missing parens demand sum odd and even at once, vacuous

Continued on next page

Table 16: Ground-truth specification ISSUE findings (80 entries). (continued)

ID	Taxonomy	Summary
89	Wrong semantics	Shift direction inverted, spec decrypts instead of encrypting
90	Wrong semantics	Second-smallest result never required to be list member
91	Wrong semantics	Recursion double-counts final sentence and drops first
93	Precedence bug	Missing parens make case-swap arms vacuous, identity passes
94	Wrong semantics	Inequality picks smallest prime instead of largest
95	Wrong semantics	Digit keys pass both toLower and toUpper checks
97	Missing constraint	Missing natAbs makes negative operands' unit digit wrong
98	Wrong semantics	Zero-count clause quantifies all indices not just even
99	Wrong semantics Missing constraint	Truncates instead of rounding; digitless string counts valid
100	Type-system mismatch	result.length coerced to Int unsatisfiable for negative n
103	Wrong semantics	Nat floor division instead of round-to-nearest average
104	Missing edge-case clause	Zero has no digits so wrongly retained
105	Missing constraint	Missing count constraint lets deduplicating implementation pass
106	Type-system mismatch	result.length coerced to Int unsatisfiable for negative n
108	Wrong semantics	Negates least-significant digit not most-significant
109	Precedence bug Nat arithmetic truncation	$\wedge/\leftrightarrow$ precedence misgroups empty guard; Nat subtraction breaks rotation
110	Conjunct used as guard	One-way guard lets constant YES satisfy spec
111	Universal over invalid domain	Membership clause nested under keys, empty result vacuously satisfies
114	Wrong semantics	List.sublists enumerates subsequences not contiguous subarrays
115	Conjunct used as guard	Biconditional forbids zero result on non-empty all-zero grids
116	Wrong semantics	Sorts by lexicographic binary digits instead of popcount
117	Wrong semantics	splitOn empty words unfiltered makes spec unsatisfiable
121	Wrong semantics	Reads lst[i+1]! summing odd positions instead of even
122	Conjunct used as guard	$\wedge$ instead of $\vee$ forces qualifying element, unsatisfiable
125	Universal over invalid domain	Minimality quantified over wrong-length paths makes spec unsatisfiable
126	Precedence bug	$\wedge$ binds tighter than $\rightarrow$ collapsing chain to True
129	Wrong semantics	words-none biconditional inverted and ord uses first-char codepoint
130	Universal over invalid domain Wrong semantics	Length n instead of n+1 contradicts at n=0
132	Precedence bug	$\rightarrow$ binds tighter than $\leftrightarrow$ mis-grouping the biconditional
134	Wrong semantics	String.join drops spaces collapsing multi-word tail to one word
138	Missing constraint	Missing positivity lets even zeros satisfy the sum
139	Nat arithmetic truncation	Nat division makes factorial recurrence lossy and non-unique
140	Conjunct used as guard	Non-maximal pref lets implementation dodge the dash collapse
143	Universal over invalid domain	Empty result forces empty string into words, unsatisfiable
144	Wrong semantics	Divisibility direction inverted between numerator and denominator products
150	Conjunct used as guard	Biconditionals instead of implications, unsatisfiable when x equals y and n prime
151	Type-system mismatch	Rat mod makes isEven always true, skip branch drops tail sum
152	Conjunct used as guard	Equal-length input assumption asserted as conjunct not precondition guard
153	Nat arithmetic truncation	Nat subtraction saturates negative extension strength to zero

Continued on next page

Table 16: Ground-truth specification ISSUE findings (80 entries). (continued)

ID	Taxonomy	Summary
154	Precedence bug	Missing parens make $\rightarrow$ bind inside $\leftrightarrow$, unsatisfiable for empty b
155	Universal over invalid domain	Missing num = 0 base case makes even-digit recurrence unsatisfiable
156	Extra/over-restrictive constraint	Lowercase check in antecedent makes spec vacuously satisfiable
158	Conjunct used as guard	Empty-string result biconditional unsatisfiable on empty word input
159	Missing constraint	Nat subtraction lets number=0 under-constrain result[0]
161	Vacuous disjunct	Disjunction of case implications is vacuous, needs $\wedge$
163	Conjunct used as guard	Length bound as top-level conjunct forbids empty branch

Table 17: Specification MISMATCH findings (13 entries).

ID	Taxonomy	Summary
0	Edge-case behavior	GT accepts any result on short lists, generated forces false
1	Precondition gap Structural-op choice	Disagreement over non-paren non-space characters in input
6	Structural-op choice Precondition gap	GT keeps empty split tokens and leaves deep prefixes unconstrained
17	Structural-op choice Precondition gap	GT counts empty slots and forces off-legend tokens to zero
19	Structural-op choice Precondition gap Ordering vs multiset	Whitespace tokenizing plus multiset versus membership equality disagreement
21	Edge-case behavior	Divergence on all-equal lists where min equals max
29	Ordering vs multiset	GT multiset reading, generated requires order preservation
30	Ordering vs multiset	GT ignores order, generated requires input order preserved
32	Algorithmic variant	GT allows approximate root, generated requires exact zero
57	Edge-case behavior Algorithmic variant	Trivial-list guard and strict vs non-strict monotonicity on plateaus
87	Ordering vs multiset	GT allows duplicate coordinates, generated forbids via strict order and Nodup
120	Precondition gap	Bounds as preconditions vs constraints impl must uphold
157	Precondition gap	GT rejects degenerate triples, generated specs guard or accept them

Table 18: Fixed benchmark-base ISSUE findings (4 entries).

ID	Taxonomy	Summary
32	Missing input precondition	omits rational-root existence precondition
39	Open mathematical problem	correctness $\forall$n needs open prime-Fibonacci conjecture
123	Open mathematical problem	spec ties Collatz-orbit completeness to the open Collatz conjecture; theorem unprovable
160	Fixed helper defect	applyOp returns none on division by zero, unsatisfiable