Agentic Proving for Program Verification

Alessandro Sosso[1], Akhil Arora[1], and Bas Spitters[1]

Department of Computer Science,
Aarhus University, Aarhus, Denmark
{sosso,akhil.arora,spitters}@cs.au.dk

Large Language Models hallucinate, and even modern "reasoning" models can produce arguments that look coherent but are logically invalid. Formal logic offers a rigorous alternative, embodied in interactive proof assistants (Lean [14], Rocq [18], Isabelle/HOL [15], ...). Recent work has made striking progress on LLM-based proof construction in such settings [8, 1, 4, 11, 2], using a range of strategies — whole-proof generation, proof-state search, and, increasingly, agentic designs combining planning, tool use and iterative refinement.

We study how those advances translate to *program verification*, a setting both practically relevant and structurally challenging: unlike formal mathematics, it requires reasoning about executable artifacts, explicit specifications and subtle side conditions, under strict syntactic and semantic constraints. We evaluate state-of-the-art agentic paradigms on CLEVER [19], a recent Lean 4 benchmark for verifiable code generation. The work follows up on our prior experiments (Appendix A), in which we found that agentic systems based on Claude Code outperformed whole-proof generation models and symbolic tactics on the proof-generation tasks of CLEVER and VERINA [22], and consistently identified (and often fixed) bugs in the benchmarks themselves.

Contributions. Our agentic Claude Code setup reaches a new state of the art on the full CLEVER pipeline. **Specifications:** Claude generated specifications judged valid on 98.8% of problems, of which 81.3% also passed CLEVER's isomorphism check on the benchmark's non-buggy portion. **Implementations:** we certified 87.5% of Claude's implementations against the correct ground-truth specifications. **End-to-end:** we completed the full generation and verification pipeline on 98.1% of entries with self-consistent premises. Beyond raw scores, Claude produced well-argued suggestions about its own outputs, which we then verified by hand; these suggestions exposed lingering bugs in CLEVER and motivate concrete alternatives to isomorphism-against-ground-truth as an evaluation method for autoformalization.

Methodology

Benchmark. CLEVER [19] is a Lean 4 dataset of 161 problems adapted from HUMANEVAL [3]. Each entry pairs a natural-language docstring with a function signature and a ground-truth specification `problem_spec`, and asks the model in successive steps to generate a specification `generated_spec` (SPEC_GEN), a proof of isomorphism between `problem_spec` and `generated_spec` (SPEC_ISO), an implementation of the function (IMPL_GEN), and a correctness proof relating the implementation to the specification (PROOF_GEN). The four tasks split into two stages: *specification certification* (SPEC_GEN+SPEC_ISO) and *implementation certification* (IMPL_GEN+PROOF_GEN).

Setup. We run the Claude Agent SDK on Claude Opus 4.6, primed with *lean-lsp-mcp* (Lean LSP access + lemma-search over the project context and Mathlib) and *lean4-skills* (Lean-specific instructions, tactic recipes and workflow patterns). Generation tasks use a short natural-language prompt; proving tasks use *lean4-skills*' `/lean4:autoprove` multi-cycle routine (3600 s timeout). We patched bugs and typos in the benchmark on a fork as we encountered them (leaving ground-truth specifications, helper functions and theorem signatures untouched). At each task completion we asked Claude to label the result with a status flag — OK, FAIL, ISSUE (provably

impossible due to bugs, with rigorous supporting argument), and for proving tasks the refined MISMATCH (both sides valid but distinct interpretations); see Appendix B. We cross-checked Claude's flags and found they were consistent with the benchmark evaluation harness results. Claude further provided analyses (again human-verified) of each ISSUE and MISMATCH entry identifying the source of the discrepancy (e.g. bugs in ground-truth vs generated specifications).

Results

We ran experiments on all four pipeline tasks using a variety of prompts of increasing detail and guidance. In all cases we used a single attempt, since this proved enough for Claude to produce specifications, implementations and proofs of consistently high quality, as confirmed under manual inspection.

Specification certification. SPEC_GEN returned OK on all 161 entries across all our prompts (simple, guided, and a multi-spec prompt enumerating distinct admissible readings). The results of SPEC_ISO however did not match: despite high-quality generated specifications, proving isomorphism remained hard, with most failures stemming from bugs in the ground-truth itself. We developed the sequence of prompts described in Appendix C to address this discrepancy, but no amount of prompting could overcome the inherent limitations of this evaluation methodology. Our final manual triage consolidated 65 OK, 81 ISSUE due to ground-truth bugs, 13 MISMATCH, 2 FAIL. Claude thus generated at least one specification that judged as valid interpretation of the description for 98.8% of entries, with 81.3% (65/80) also passing the isomorphism check over non-buggy ground-truths.

Implementation certification. Starting from ground-truth `problem_spec`, IMPL_GEN succeeded on all entries; PROOF_GEN returned 104 OK, 9 FAIL, 48 ISSUE, of which we attributed 42 to ground-truth bugs (all within the 81 SPEC_ISO ISSUE entries). Restricted to the 80 problems with non-buggy ground-truth, we certified 70 implementations: 87.5%. We then re-ran the pipeline on Claude's own simple- and guided-prompt specifications, certifying 144 and 148 entries respectively; across both, 154/161 had at least one OK attempt: 98.1% end-to-end after we excluded 4 entries with bugs in unchangeable benchmark components.

Discussion. Appendix C contains more detailed insights into the obtained results: per-prompt breakdown for both the specification and implementation certification stages; an analysis of the sources of ISSUE and MISMATCH flags; and the resources employed. Appendix D dissects Problem 32, which condenses many phenomena.

Conclusions

Our experiments show that the Claude-based agentic setup almost saturates the CLEVER benchmark, and together with our preliminary experiments they corroborate the claim that compiler-in-the-loop agentic systems are the most effective scaffold for foundational program verification. We therefore expect other recent models to perform similarly under equivalent agentic access to compiler feedback. Existing benchmarks no longer adequately stress modern agentic provers, motivating a need for harder benchmarks and more rigorous evaluation methodologies.

Our results also expose structural limitations in how CLEVER is evaluated: isomorphism scoring presupposes that the generated and reference specifications share a single intent, a presupposition broken by the inherent ambiguity of natural-language descriptions [9] and the latent freedom in specification form, as evidenced by our MISMATCH category. Pinning down intent (precise descriptions, formalisation hints, LLM-judge disambiguation) is one route forward; where this is intractable, Claude's self-analyses, property-based testing of specifications, and certifying an implementation against the model's own specification are pragmatic alternatives.

References

[1] 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.

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

[3] 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.

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

[5] Joshua Clune. LeanHammer. *GitHub repository*, 2026. https://github.com/JOSHCLUNE/LeanHammer, accessed on Jan 21st 2026.

[6] Oliver Dressler. Lean-lsp-mcp. *GitHub repository*, 2026. https://github.com/oOo0oOo/lean-lsp-mcp, accessed on Feb 3rd 2026.

[7] Cameron Freer. Lean4-skills. *GitHub repository*, 2026. https://github.com/cameronfreer/lean4-skills, accessed on Jan 21st 2026.

[8] Thomas Hubert, Harsh Mehta, Laurent Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025.

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

[10] 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, pages 253–266, New York, NY, USA, January 2023. Association for Computing Machinery.

[11] 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.

[12] 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.

[13] 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.

[14] 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*, page 625–635, Berlin, Heidelberg, 2021. Springer-Verlag.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] The Lean Developers. *The Lean Language Reference: The grind tactic*. Lean FRO, 2024. Lean version 4.15.0 or later.

[21] 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.

[22] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation, October 2025.

A Preliminary Experiments

In prior experiments, we evaluated a suite of automated provers (agentic, specialised, and symbolic) on proof generation on the CLEVER and VERINA [22] datasets, finding that agentic paradigms substantially outperformed all alternatives. We summarise our findings here, both because they directly motivate the present work and because some of those observations reappear, refined, in our current results.

Setup. The earlier study focused exclusively on proof generation starting from ground-truth specifications and implementations, a task outside the scope of either benchmark's provided harness. 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; natural-language descriptions were not provided. This restriction also limited the CLEVER subset to the 49 entries (out of 161) for which an implementation was available. The agentic configurations were further built on Claude Opus 4.5 and on an earlier release of *lean4-skills* [7] 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 we reported.

Provers. The evaluation covered three families. The *agentic* family included Claude Code with *lean4-skills* and *lean-lsp-mcp* [6], and the *Numina-Lean-Agent* [13], a wrapper around Claude Code combining it with an extended fork of *lean-lsp-mcp*, run with `max_rounds=3`. The *whole-proof generation* family — specialised models trained or fine-tuned for formal verification — was represented by Kimina-Prover-72B [21], with DeepSeek-Prover-V2 [17] and Goedel-Prover-V2 [12] in the same class. The *symbolic* family covered Grind [20], LeanHammer [5], Aesop [10], and Canonical [16]. We tested Aristotle [1], Harmonic's high-level proof-state-aware Monte Carlo Graph Search system, alongside as a separate state-of-the-art baseline.

Results. The evaluation revealed a clear performance hierarchy (Figure 1 and Table 1): agentic systems based on Claude Code defined the state of the art, 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

	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)

Table 1: 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.

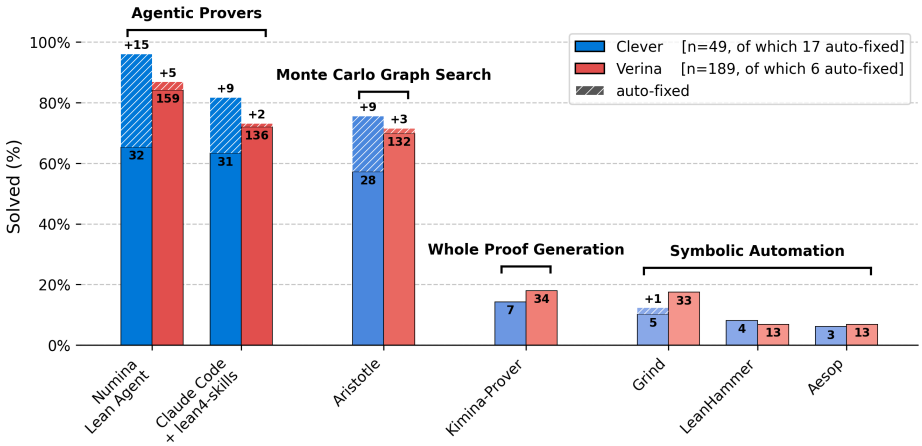

Figure 1: Overview of prover performance on the CLEVER and VERINA benchmarks. Solid bars indicate the number of tests solved on the verified-correct portions of each dataset, while the patterned bar segments above show additional solved instances obtained after correcting erroneous benchmark entries. Numbers annotated with a '+' denote the corresponding count of these additional solves.

28 respectively; out of the 183 correct VERINA-P entries, the same three provers solved 159, 136 and 132. We attribute the high performance of the agentic systems largely to their direct access to compiler feedback through their search and tool-use loop; 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.

We also found that Claude Code and Aristotle consistently identified errors in the benchmarks themselves — Aristotle through its goal-negation search mechanism, Claude Code through its natural-language 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 we then supplied the natural-language 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. We then classified the fixes (Table 2): most addressed wrong specification logic (9 cases), implementation bugs (6) and missing preconditions (5), with a few off-by-one and boundary errors (3).

	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

Table 2: Number of fixes per benchmark by kind.

B Methodology Details

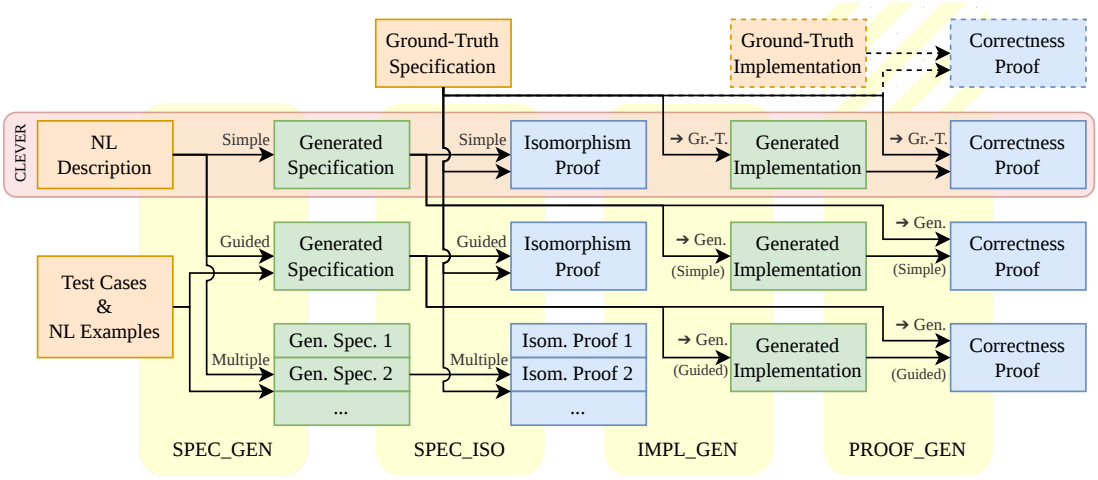

Figure 2: Schematic overview of our experimental pipeline. The four generation and proof tasks are identified in yellow: CLEVER's original intended setup spans horizontally in red across the top row, while other pathways represent our custom variations of the setup. The dashed portion in the top-right corner is the preliminary experimentation setup of Appendix A.

Status flags. At completion of each task, Claude was prompted to assign a status flag to the result, according to the following criteria:

- 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 / incorrect assumptions, with rigorous supporting arguments.

For SPEC_ISO, the ISSUE case is further split:

- MISMATCH: equivalence is unprovable because the specifications differ, yet both are valid interpretations of the description and test cases;
- ISSUE: equivalence is unprovable because at least one specification is erroneous (contradicts the description or test cases).

For PROOF_GEN, the ISSUE case is similarly refined:

- MISMATCH: equivalence is unprovable because the specification/implementation differ, yet both are valid interpretations of the description and test cases;
- ISSUE: equivalence is unprovable because the specification, the implementation, or both are erroneous.

ISSUE entries were further analysed by source: ground-truth `problem_spec`, generated `generated_spec`, or both for SPEC_ISO; and specification, implementation, or fixed benchmark components for PROOF_GEN. MISMATCH entries (only encountered in SPEC_ISO) were classified by whether the generated specification was strictly stronger or weaker than the ground-truth, or neither.

C Detailed Breakdown of Results

Specification certification: per-prompt breakdown

Simple prompt. Our first run used a minimal prompt outlining only task instructions and the expected input/output format. The SPEC_GEN task ran on the natural-language description alone, matching the protocol of the original benchmark authors, and returned OK on all 161 entries. The corresponding SPEC_ISO run did not exhibit the same pass rate: only 4 of the failures stemmed from synthesis timeouts, the remainder from non-isomorphic specifications. Among these, Claude flagged 22 MISMATCH, 67 ISSUE due to bugs in the ground-truth `problem_spec`, against only 9 attributable to bugs in the generated `generated_spec` and 1 in both — classifications we then verified by hand. Within the 22 MISMATCH entries, `generated_spec` was strictly stronger than `problem_spec` in 15 cases against only 1 instance of the contrary, with the remaining 6 admitting neither direction — consistent with the tendency of the ground-truth specifications towards less restrictive formulations.

Guided prompt. Our second prompt aimed to align the generated specifications more closely with the ground-truth, without revealing it. We derived candidate guidelines from an automated analysis of the previous run's outputs and conversation traces, identifying recurring design choices and patterns underlying the ground-truth specifications, and refined them by hand. The guidelines covered: the structural pattern of the specification (existential statement of termination and post-condition satisfaction, input constraints as implication guards), permissive treatment of edge cases and degenerate inputs, the preference for *relational* over *functional* formulations, recursive/inductive patterns, and Lean 4 syntactic recommendations. The prompt also imposed a specific specification-body format and supplied the example usages and the formal Lean test cases. SPEC_GEN again produced high-quality specifications conforming to the format; SPEC_ISO showed a decrease of ISSUE entries due to bugs in `generated_spec` (down to 2), most of which now appeared as MISMATCH instead. MISMATCH entries rose to 31, ground-truth ISSUE entries to 70, partly driven by ground-truth specifications now conflicting with the supplied test cases. The strict-stronger skew was less pronounced (17 stronger, 8 weaker, 6 neither) but consistent in direction.

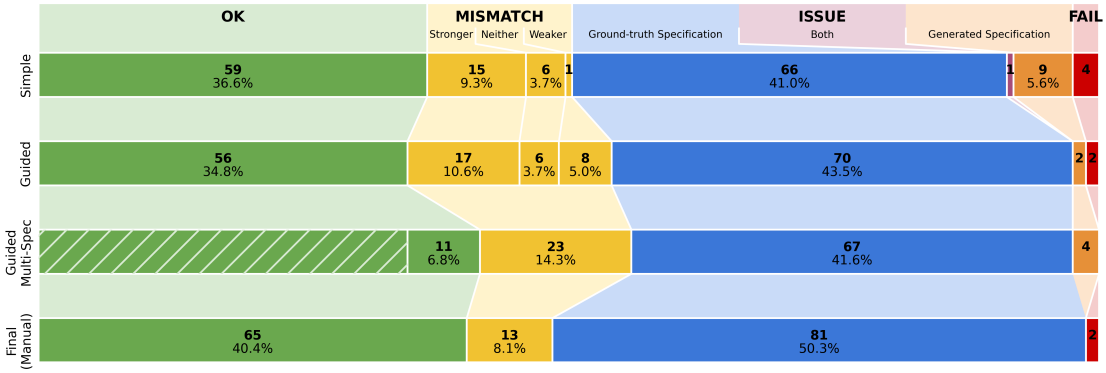

Figure 3: Distribution of output flags for SPEC_ISO runs. The following color scheme will be used throughout the paper: green for OK entries, red for FAIL, yellow for MISMATCH, blue for ISSUE in benchmark components, orange for those in generated components, purple for issues in both.

Multi-spec prompt. The infeasibility of proving the isomorphism for MISMATCH entries stemmed from the generated specification not aligning with the ground-truth's interpretation. We therefore designed a third prompt asking Claude to output "all and only" valid interpretations of the problem, varying across precondition boundary, edge-case permissiveness, quantifier interpretation, operation semantics, and property strength. We tested this prompt on the 105 entries flagged as ISSUE or MISMATCH in the previous run. Claude generated up to 4 specifications per problem (avg. 2.76, 290 total). Running SPEC_ISO on each, only 15 candidates proved isomorphic to `problem_spec`, spanning 11 entries; 66 received ISSUE in at least one attempt, 24 received MISMATCH overall.

Final classification. We consolidated the three runs through a final manual examination, fixing classifications where runs disagreed and identifying lingering bugs not captured by the automated flags (e.g. SPEC_ISO succeeding due to bugs in helper definitions). This pass also confirmed that Claude's per-entry classifications and arguments were reliable, accurate and well-reasoned. The consolidated classification tallies 65 OK, 81 ISSUE, 13 MISMATCH and 2 FAIL (Figure 3).

Implementation certification: per-prompt breakdown

From ground-truth specifications. IMPL_GEN succeeded on all entries (Claude initially flagged a few as ISSUE after identifying bugs in test cases, which we fixed). PROOF_GEN produced 104 OK, 9 FAIL (timeouts) and 48 ISSUE. We attributed 42 of the 48 ISSUE flags to the supplied specifications, and all of them fall within the 81 ISSUE entries from the final SPEC_ISO classification. Bugs in ground-truth specifications partly explain this rate: Claude sometimes produces a correct implementation that does not satisfy the erroneous specification. Conversely, since Claude sees the specification at generation time, it can produce implementations satisfying it; combined with the fact that many specification bugs are underspecifications, this explains why several entries with bugged ground-truths still succeed at PROOF_GEN. Restricted to the 80 entries with non-buggy ground-truth, we certified 70 implementations (87.5%).

From generated specifications. We then re-ran the pipeline on the specifications produced by simple and guided SPEC_GEN. IMPL_GEN returned OK on all entries, with three exceptions in the simple-prompt run where Claude detected a mismatch with the supplied test cases (the simple SPEC_GEN prompt did not include them). PROOF_GEN certified 144 entries in the

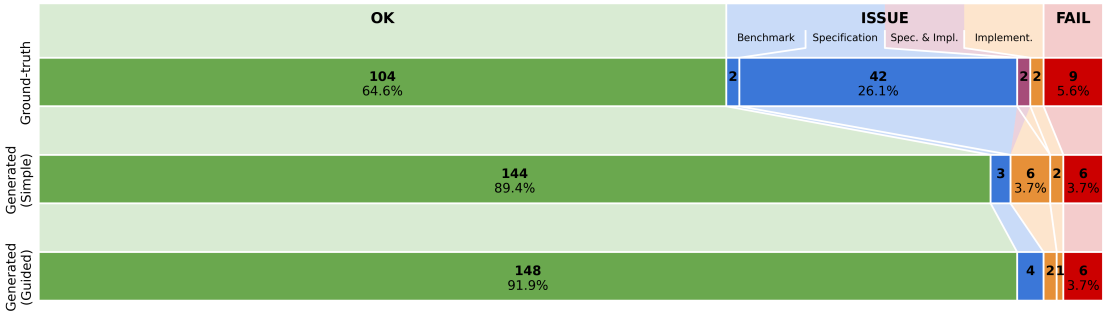

Figure 4: Distribution of output flags for PROOF_GEN runs. Note the change of coloring for specification ISSUE between rows, as their source changes from the ground-truth to automatic generation.

simple-prompt run (6 FAIL, 11 ISSUE) and 148 in the guided-prompt run (6 FAIL, 7 ISSUE). After manual review, we found that the ISSUE entries originate from a few distinct sources: bugs in the generated implementations (e.g. implementations with finite fuel that does not generalise), bugs in the generated specifications (e.g. incorrect handling of borderline cases), and the fixed elements in the benchmark itself.

Across both runs, 154/161 entries had at least one OK attempt (95.7% end-to-end). Excluding 4 entries with bugs in fixed benchmark components yields 98.1% (Figure 4).

Bug taxonomy and breakdown

Ground-truth specifications. 80/161 ground-truth specifications were flagged at least once across our runs. Our manual review confirmed that the bugs cluster on Lean-encoding pitfalls (48 problems): *conjunct-as-guard* patterns using $(P \wedge Q)$ in place of $(P \to Q)$ (15); *precedence bugs* from omitted parentheses around $\wedge, \to, \leftrightarrow$ (12); universal quantifiers over inputs where the 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. The remaining 34 problems carry semantic errors: *over-* or *under-restrictive constraints* (10), *missing edge-case clauses* (4), and overall wrong interpretations of the description (21). For 5 problems (81, 99, 109, 130, 139), we identified two types of issue co-occurring.

Generated specifications. Claude-generated specifications fail in a *semantic* register, misstating or under-specifying the relation, and rarely in a *syntactic* one (only two instances). Within the semantic class, *missing constraints* dominate and *wrong semantics* are less frequent: an effective inversion of the ground-truth profile, suggesting that Claude accurately formalises valid interpretations. Guidance is effective: our design guidelines produced a five-fold drop in issues (10 to 2) and the complete disappearance of *missing-constraint* and *over-/under-restrictive-constraint* errors. Those errors reappear in the multi-specification run, reflecting Claude's intentionally more "adventurous" generation when asked to cover the interpretation space.

Mismatches. MISMATCH entries reflect *authoring choices* rather than errors, often along several dimensions per entry. 73 firings concern the boundary of the input domain (gaps in preconditions, edge-case handling); 37 concern conditions on the function output (ordering of lists vs. multisets, canonical forms); 22 concern structural and definitional choices; 10 are semantic in nature (different but mathematically defensible formulations, quantification over indices instead of values).

Implementations. We flagged issues in generated implementations for only 7 problems. Three involve hard-coded *bounded fuel* (e.g. the 200-iteration Newton–Raphson loop of Problem 32; see Appendix D) that does not suffice for the arbitrary inputs 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: numerical sort instead of sort by binary digit count).

Benchmark. Four problems carry issues in fixed, non-editable parts of the benchmark, which neither the generated specification nor the generated implementation can repair. Problem 123 (`collatz`) and Problem 39 (`prime_fib`) require, respectively, a proof of the Collatz conjecture and the open question of infinitely many Fibonacci primes. Problem 160 (`do_algebra`) ships a fixed helper `applyOp` returning `none` on division by zero, which is not parsed by other parts of the script. We discuss Problem 32 separately in Appendix D.

C.1 Resources

All experiments ran on a *HP EliteBook 850 G6* laptop, as no particular computational resources were required: the workstation hosted our minimal evaluation infrastructure and the Lean-related routines accessed via MCP, while inference was performed through Anthropic's Claude API. Runtimes per solve were usually between 10 and 15 minutes for proving tasks (SPEC_ISO and PROOF_GEN), against around 90 seconds on average for formalisation tasks (SPEC_GEN and IMPL_GEN), with an exception for the multi-specification SPEC_GEN run averaging around 200 seconds. Costs per solve scaled linearly with time, on average around $0.3 for formalisation tasks ($0.5 for multi-spec SPEC_GEN) and $2.0–2.5 for proving tasks. The longest-running instances reaching the 1-hour timeout stood at around $10 per run.

D 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 we observed throughout our evaluation. The task asks for a real root of an odd-degree polynomial given by its list of coefficients.

Already in the preliminary experiments (Appendix A), we found the provided ground-truth implementation to be incorrect: it was based on Newton–Raphson, which is not globally convergent and can enter cycles even on simple polynomials. Following insight from the agent, we replaced it with bisection initialised from the Cauchy root bound, with bisection fuel adaptive in the bit-size of the coefficients, ensuring convergence to the target precision (see Figures 5 and 6).

At specification certification, no run produced a `generated_spec` that proved isomorphic to the ground-truth. 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 description nor the example usages mentioned approximation or any threshold for it. We flagged the corresponding entries either as MISMATCH or as ISSUE in `generated_spec`.

```
def implementation (xs: List Rat) : Rat :=
  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;
  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
+ let lc := |xs.getLast!|;
+ let initBound : Rat := if lc = 0 then 1
+   else 1 + xs.dropLast.foldl (fun acc a ⇒ acc + |a|) 0 / lc;
+ let rec findBound (xs: List Rat) (b: Rat) (fuel: Nat) : Rat :=
+   match fuel with
+   | 0 ⇒ b
+   | fuel' + 1 ⇒
+     if poly xs b * poly xs (-b) ≤ 0 then b
+     else findBound xs (b * 2) fuel';
+ let rec bisect (xs: List Rat) (lo hi: Rat) (fuel: Nat) : Rat :=
+   match fuel with
+   | 0 ⇒ (lo + hi) / 2
+   | fuel' + 1 ⇒
+     let mid := (lo + hi) / 2;
+     if |poly xs mid| ≤ eps then mid
+     else if poly xs lo * poly xs mid ≤ 0 then bisect xs lo mid fuel'
+     else bisect xs mid hi fuel';
+ let bound := findBound xs initBound 100;
+ let coeffBits := xs.foldl (fun acc a ⇒ acc + a.num.natAbs.size + a.den.size) 0;
+ let fuel := 1000000 + coeffBits * (xs.length + 1) * 40;
+ bisect xs (-bound) bound fuel
```

Figure 5: Fix of CLEVER problem 32 generated by Claude Code.

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·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 6: Extract of Claude's replies explaining the edits it made in fixing problem 32.

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 side-step the fact that `Real` is not a computable type in Lean (as a developer comment in the entry itself acknowledges). Any implementation is therefore incorrect on odd-degree polynomials with no rational root, such as $x^3 + 2$. We flagged the PROOF_GEN attempts accordingly as ISSUE in either the implementation or the benchmark-imposed signatures. The same observation explains why we flagged logically correct specifications as ISSUE at SPEC_ISO time: 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 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.