Agentic Proving for Program Verification

Alessandro Sosso [1] **Akhil Arora** [1] **Bas Spitters** [1]

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*, we evaluate Claude Code in an agentic proving framework on CLEVER, a Lean 4 benchmark for verifiable code generation. Our results show that Claude generates arguably valid specifications for 98.8% of problems (with 81.3% also accepted by CLEVER's isomorphism-based scoring on the correct portion of the benchmark), certifies implementations against correct ground-truth specifications for 87.5% of problems, and reaches a 98.1% success rate on the end-to-end program generation and verification pipeline over entries with self-consistent premises. Across all stages, Claude further provides high-quality feedback on its own attempts (as confirmed under manual review), identifying underlying causes of failure and lingering bugs in the dataset. These findings highlight a growing mismatch between the difficulty of existing program verification benchmarks and the capabilities of modern agentic provers, and point to the need for more rigorous, bug-resilient evaluation methodologies, and in particular for alternatives to isomorphism-based scoring of generated specifications. More broadly, our results provide empirical evidence that tight compiler-in-the-loop agentic paradigms are currently the most effective approach for foundational program verification.

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)). In this work, we focus on the first category, and in particular on 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, and increasingly, agentic designs that integrate planning, tool use, and iterative refinement.

In this paper, we study how these advances translate to *program verification*, a setting that is both practically relevant and structurally challenging. Unlike informal 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 evaluate state-of-the-art agentic paradigms on CLEVER (Thakur et al., 2025), a recent Lean 4 benchmark for verifiable code generation. Figure 1 provides a schematic overview of our experimental pipeline.

Preliminary experiments. Prior testing by the authors evaluated leading agentic and non-agentic provers on the proof-generation task of two Lean 4 benchmarks for verifiable code generation, CLEVER and VERINA (Ye et al., 2025). Those experiments showed that agentic systems based on Claude Code achieved near-saturation performance on the verified-correct portions of these 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 specifications and implementations of the

[1]Department of Computer Science, Aarhus University, Aarhus, Denmark. Correspondence to: Alessandro Sosso <sosso@cs.au.dk>.

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

benchmarks themselves, in several cases proposing fixes and successfully proving the corrected statements. These findings suggested that compiler-in-the-loop agentic systems are the most effective scaffold for foundational program verification, and that existing program verification benchmarks no longer stress modern agentic provers as much as their design intended. The present work follows up on both observations, evaluating an agentic system on the full CLEVER pipeline using its native evaluation infrastructure rather than the custom distillation used in the preliminary study; we refer to Appendix A for details.

Contributions. Our agentic Claude Code setup reaches a new state of the art on the full CLEVER pipeline. Specifically:

- **Specification certification.** Claude generates arguably valid specifications for 98.8% of problems, of which 81.3% are also accepted by CLEVER's isomorphism-based scoring on the portion of the benchmark where no issues in the ground-truth specifications were identified.
- **Implementation certification.** Claude generates and successfully certifies implementations against the correct ground-truth specifications for 87.5% of problems.
- **End-to-end pipeline.** On the entries with self-consistent premises, Claude reaches a remarkable 98.1% success rate over the full specification + implementation + proof pipeline.
- **Self-diagnostic feedback.** Beyond raw performance, Claude consistently produces well-argued analyses of its own outputs, identifying and classifying both the underlying causes of failure and lingering bugs in the CLEVER benchmark.
- **Methodological insight.** Our findings expose structural limitations of isomorphism-against-a-ground-truth as an evaluation method for autoformalization, and motivate concrete proposals for alternative evaluation strategies.

2. Methodology

2.1. Dataset

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 testing harness provided with the benchmark allows the selective retrieval of the sections related to each task, that can then be merged back into a unique Lean script. The benchmark then evaluates the solutions 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 benchmark repository was used, in order to track the implementation of fixes in the benchmark and of minor edits in the evaluation scripts required by our custom 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 a strict end-to-end ceiling of $1/161$ ($\approx 0.62\%$) under a pass@600s protocol, achieved by o4-mini, Claude-3.7 and DeepSeek-R1 in a few-shot configuration, and by GPT-4o and Claude-3.7 paired with the COPRA proof agent (Thakur et al., 2024). The best per-stage figures were $14/161$ (8.7%) on implementation certification (Claude-3.7+COPRA) and $3/161$ (1.86%) on specification certification (GPT-4o+COPRA). The strongest partial-pipeline follow-up (Li et al., 2026) reports 54.0% on CLEVER, but the figure measures only the correctness-proof stage given a separately-supplied implementation generated by GPT-5.2; frontier reasoning models in the same setting reach at most

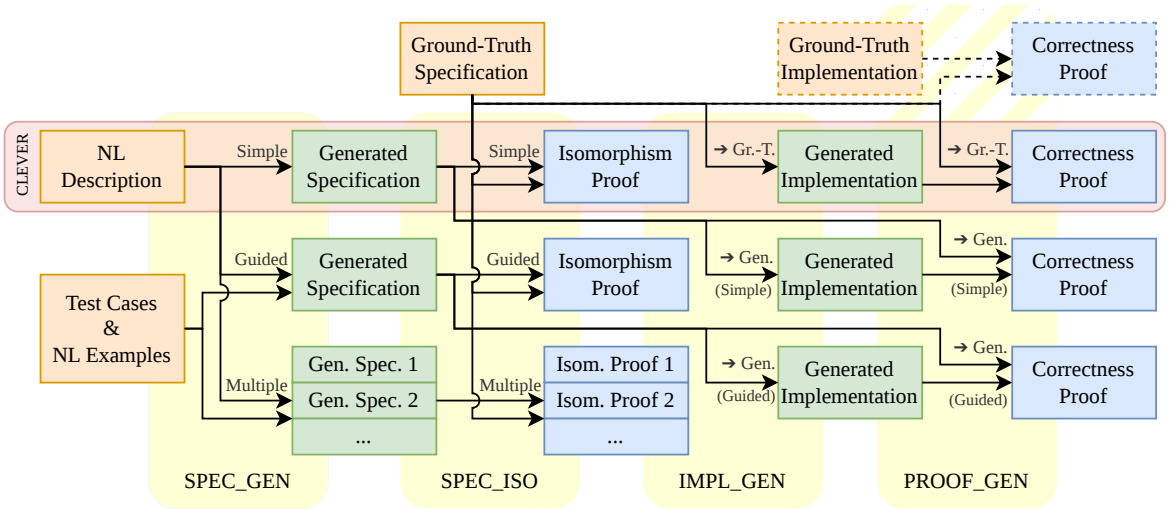

Figure 1. Schematic overview of our experimental pipeline. The four generation and proof tasks are identified in yellow: CLEVER's original intended setup spans horizontally in red across the top row, while other pathways represent our custom variations of the setup. The dashed portion in the top-right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

$\approx 23.6\%$.

Independent follow-up works have also surfaced spec-quality issues in CLEVER. Specification-level property-based testing (Feng et al., 2026) identifies 18 defective entries (11.2%), comprising 16 underspecified postconditions, 1 implementation bug and 1 incorrect specification; integrated quickcheck on the top-level correctness theorem (Li et al., 2026) additionally disproves 10 entries.

2.2. Experimental Setup

Our experimental setup is built on top of the Claude Agent SDK (Anthropic, 2026) running Claude Opus 4.6, with each agent instance primed with *lean-lsp-mcp* (Dressler, 2026), a specialized MCP server that interfaces with the Lean LSP and provides search tools to find relevant lemmas in the project context and Mathlib, and *lean4-skills* (Freer, 2026), a package injecting Lean-specific instructions and commands, tactic best-practices and workflow patterns into the model's context. For each attempt, the system initializes a temporary directory from a template as a Lean project, then creates a Lean file formatted using relevant components of a benchmark entry, and prompts a fresh agent instance to edit the file according to the current task. Upon completion or timeout (set at 3600 seconds), the resulting file is retrieved along with the trace of the agent conversation, of tool invocations, and metadata of the run.

Each new Claude instance is provided with a description of the task as system prompt, outlining details and the objective of the task, the format of the input file and of the desired output, behavior rules, and eventual guidelines that the agent is expected to follow. The agent is then simply prompted with either a short NL prompt asking to fill sections of the target Lean file following the previously provided instructions (SPEC_GEN, IMPL_GEN), or, in case the task requires proving a given theorem (SPEC_ISO, PROOF_GEN), the `/lean4:autoprove` command from the skills package, which starts a multi-cycle theorem-proving routine.

Claude is then asked to report on the outcome of the task with a flag, either in the same session as a follow-up prompt after termination or through the successive analysis of the Lean outputs and agent replies. The agent can classify it as successful (OK), unsuccessful (FAIL), or rigorously argued as impossible due to bugs in the input or incorrect assumptions (ISSUE) (see Appendix B.1).

For theorem-proving tasks (SPEC_ISO, PROOF_GEN), the 'impossibility' case was further subdivided into MISMATCH, when the theorem was not provable but the input elements were different yet valid interpretations of the source NL description and test cases, and ISSUE, when the proof failed due to at least one element (specification / implementation / fixed benchmark component such as signatures) being erroneous, i.e. contradicting the NL description (again, in both cases rigorous supporting arguments or counterexamples must have been provided). To extract deeper diagnostic insights, results in the ISSUE category were further analyzed with granularity 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 benchmark structural components for PROOF_GEN.

Entries in the MISMATCH category were found only in the solutions of SPEC_ISO. Further analysis was conducted in this case to identify if either direction of the isomorphism theorem was provable (i.e. whether the generated specifica-

tion was strictly stronger or weaker than the ground truth) or whether neither held.

Cross-checks of the Claude-reported flags against the evaluation through the benchmark harness showed no discrepancies, as all and only OK solutions passed the evaluation, except for two instances: one where compilation failed due to a bug in the test cases, but the solution was otherwise correct; the other, an instance that successfully passed the evaluation, but was reported as FAIL due to Claude timing-out before termination.

3. Results

We ran experiments on all four pipeline tasks using a variety of prompts of increasing detail and guidance. In all cases, a single attempt (pass@1) was made, as this proved to be enough for Claude to produce specifications, implementations and proofs of consistently high quality, as confirmed under manual inspection. The pass rates reported by the benchmark harness, however, did not always reflect this: in several configurations the success rate fell substantially below what manual review of the outputs would suggest. We therefore designed the sequence of prompts described below to progressively address this discrepancy. More details on results can be found in Appendix C.

3.1. Specification certification

Simple prompt. The first run used a minimal prompt outlining only the task instructions and the expected input/output format, without any specific guidance on how to proceed. The SPEC_GEN task ran on the NL description of the problem alone, without example usages or formal test cases, matching the protocol of the original benchmark authors. It returned the OK flag for all 161 entries. The SPEC_ISO task on the generated specifications did not exhibit the same pass rate. Only 4 of the failures stemmed from a failure in synthesizing the proof (timeouts); the remainder corresponded to non-isomorphic specifications. Among the non-isomorphic entries, Claude flagged 22 as MISMATCH, and 67 as ISSUE due to bugs in the ground-truth `problem_spec`, against only 9 attributable to bugs in the `generated_spec` and 1 in both. In other words, all but 14 entries (issues in `generated_spec` plus failures) corresponded to valid generated specifications. 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 of the implication. This skew is consistent with the tendency of the ground-truth specifications towards less restrictive formulations: they typically do not constrain the behaviour of the implementation on inputs outside the valid domain, for instance.

Guided prompt. A second prompt aimed to align the generated specifications more closely with the ground-truth, without revealing it. We obtained candidate guidelines from an automated analysis of the previous run's outputs and conversation traces, aimed at identifying the recurring design choices and patterns underlying the ground-truth specifications, and refined them by hand. We also removed any direct references or examples drawn from the benchmark entries at this stage. The guidelines covered subjects such as the structural pattern of the specification, the treatment of edge cases and degenerate inputs, the preference for *relational* formulations of properties, and useful Lean 4 syntactic recommendations. In addition, the prompt imposed a specific format for the specification body (again derived from the analysis of the ground-truth specifications), and the input file now included the example usages from the NL description together with the formal Lean test cases, with the goal of clarifying which interpretation of the problem the ground-truth had adopted.

SPEC_GEN again produced high-quality specifications, conforming to the required format and guidelines. SPEC_ISO did not show a significant overall improvement, except for a decrease of ISSUE entries due to bugs in `generated_spec` (down to 2), most of which now appeared as MISMATCH instead. This plausibly reflects the disambiguation provided by the more detailed instructions and the additional test cases. The total number of MISMATCH entries thus increased to 31, while ISSUE entries arising from bugs in the ground-truth specification grew to 70, also driven by some ground-truth specifications that now conflicted with the supplied test cases. The skew within MISMATCH entries towards stronger generated specifications was less pronounced than before (17 strictly stronger, 8 strictly weaker, and 6 admitting neither direction), although the direction remained consistent with the previous run for entries already classified as mismatches.

Multiple specifications prompt. The infeasibility of proving the isomorphism for MISMATCH entries (where, recall, both specifications validly interpreted the NL description) stemmed from the generated specification not aligning with the same interpretation adopted by the ground-truth. To increase the likelihood of producing a specification matching that interpretation, we designed a third prompt to obtain, for each problem, multiple candidate specifications covering distinct admissible readings of the NL description.

We instructed the agent to output "all and only" valid interpretations of the problem, that is the set of semantically distinct specifications that a reasonable reader could defend as correct without contradicting the NL description or the test cases. Variation ranged across dimensions such as precondition boundary, edge-case permissiveness, quantifier interpretation, operation semantics, and property strength

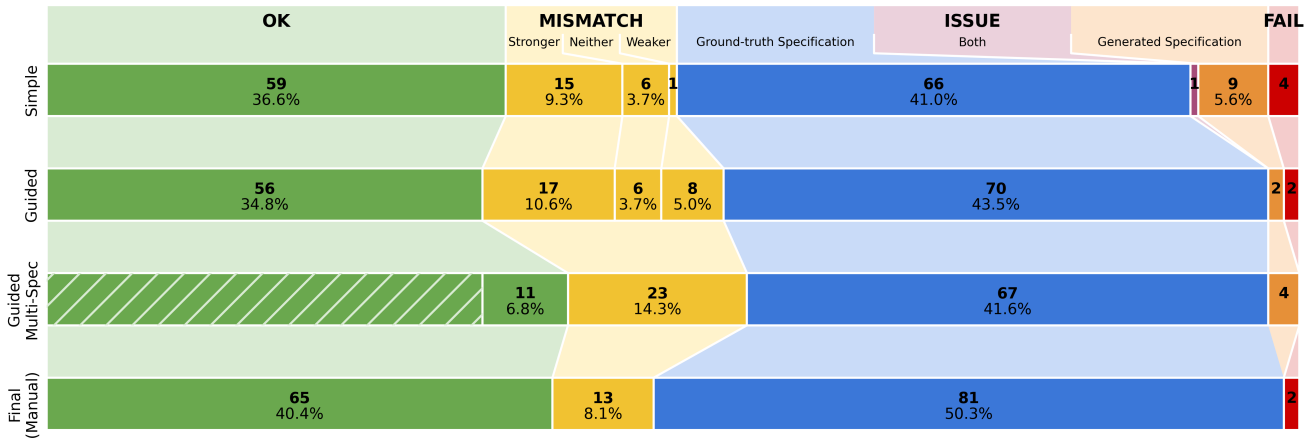

Figure 2. Distribution of output flags for SPEC_ISO runs. The following color scheme will be used throughout the paper: green for OK entries, red for FAIL, yellow for MISMATCH, blue for ISSUE in benchmark components, orange for those in generated components, purple for issues in both.

(minimal vs. maximal). The prompt retained the same guidelines and example test cases as the previous one.

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, with an average of 2.76 and a total of 290 candidate specifications. We then ran the SPEC_ISO task on each problem against all of its candidate specifications, with multiple Lean files as inputs. Only 15 candidate `generated_spec` successfully proved isomorphic to the respective `problem_spec`, spanning 11 entries; among the remaining entries, 66 received the ISSUE flag in at least one attempt, while 24 received MISMATCH overall.

Final classification. The outputs of the three runs underwent a final manual examination, that additionally fixed the classification where runs disagreed and identified lingering bugs not captured by the automated flags (e.g. cases where SPEC_ISO succeeded due to bugs in the underlying helper definitions). This examination also confirmed that Claude's per-entry classifications and supporting arguments were reliable, accurate, and well-reasoned in all cases. The resulting consolidated classification tallies 65 OK entries, where Claude generated at least one valid specification that proved isomorphic to the ground-truth; 81 ISSUE entries,[1] with arguably bugged ground-truth `problem_spec`; 13 MISMATCH entries, where the ground-truth `problem_spec` was valid but never matched by any valid `generated_spec`; and 2 FAIL entries, exhibiting only timeouts or bugs in the generated specifications. Therefore, Claude Code generated specifications that validly interpreted the source NL description of the function in at least one run for 98.8% of the entries.

[1]all flagged as ISSUE in at least one SPEC_ISO run, except for one (problem 160) where most runs succeeded, but as a consequence of all specs relying on the same bugged helper function.

3.2. Implementation certification

From ground-truth specifications. Following the original CLEVER protocol, the first run of this stage generated implementations starting from the ground-truth specifications provided in the benchmark. The IMPL_GEN task succeeded on all entries, despite Claude initially flagging a few runs as ISSUE after identifying bugs in the provided test cases, which we subsequently fixed. The PROOF_GEN task on the resulting implementations produced 104 OK entries successfully certified, 9 FAIL entries due to timeouts, and 48 entries flagged as ISSUE. The vast majority of these ISSUE flags (42 out of 48) originated in the provided specifications, and all of them belong to the 81 problems identified as ISSUE in the final classification of SPEC_ISO. Bugs in the ground-truth specifications partly explain this high rate of issues, as Claude sometimes generates correct implementations that do not satisfy the erroneous specification. On the other hand, Claude sees the specification at generation time, and can therefore produce implementations that satisfy it; combined with the fact that many specification bugs come from underspecification, this also explains why a number of entries with bugged ground-truth specifications still succeed at PROOF_GEN. Among the 80 problems with no bug found in the final SPEC_ISO analysis, 70 implementations were successfully generated and certified, for a final performance rate of 87.5%.

From generated specifications. A second run generated implementations starting from the specifications produced by the simple SPEC_GEN prompt. As before, IMPL_GEN returned OK on all entries, except for three instances where

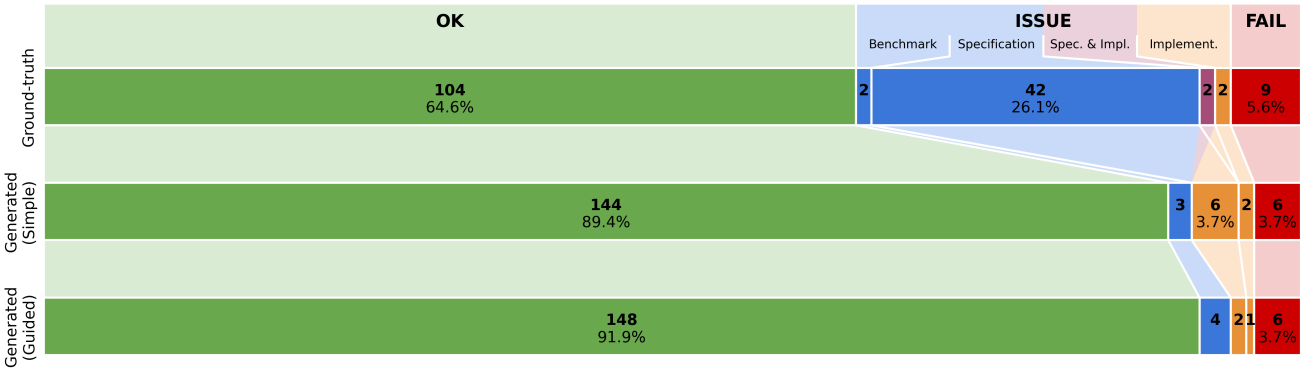

Figure 3. Distribution of output flags for PROOF_GEN runs. Note the change of coloring for specification ISSUE between rows, as their source changes from the ground-truth to automatic generation.

Claude identified a mismatch between the generated specification and the provided test cases. This is unsurprising, as the simple SPEC_GEN prompt did not include the formal test cases or the example usages from the NL description, so we could expect inconsistencies between the two. The corresponding PROOF_GEN run produced 144 successfully certified OK entries, 6 FAIL, and 11 ISSUE.

We then carried out the same pipeline on the specifications produced by the guided SPEC_GEN prompt. IMPL_GEN again succeeded on all entries; no specification–test-cases mismatch arose this time, as the test cases had already been supplied at SPEC_GEN time. The PROOF_GEN task produced 148 successfully certified OK entries, again 6 FAIL, and 7 ISSUE.

In both runs, the ISSUE entries originate from a few distinct sources: bugs in the generated implementations (e.g. implementations relying on finite fuel and thus not generalizing across all valid inputs); bugs in the generated specifications (e.g. incorrect handling of borderline cases); and, in some cases, the fixed elements in the benchmark itself. We discuss these in detail in Section 4.

Across both runs, 154 entries had at least one OK attempt, or a 95.7% success rate on the end-to-end program verification pipeline, having automatically generated arguably correct specifications, implementations and correctness proofs over two attempts (simple and guided prompts). If we remove the 4 instances of ISSUE in the benchmark itself, this success rate further rises to 98.1%.

4. Discussion

4.1. Specification certification

Autoformalization is hard to evaluate. Beyond the technical hurdle of producing a Lean specification that compiles, two more substantive issues dominate: choosing the right *strength* for the specification (loose enough not to over-constrain valid implementations, but 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 it is left implicit whether single-digit months and days should be accepted or rejected; 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, as they are often hard to spot. Despite this, our results indicate that the agentic paradigm copes well with this kind of evaluation, as evidenced by the fact that only a handful of generated specifications (and no ground-truth ones) slipped through SPEC_ISO as ostensibly valid, only to be flagged as ISSUE downstream at PROOF_GEN time, when the impossibility of certifying a valid implementation against them surfaced bugs that the equivalence check had missed.

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 that arose in each ISSUE entry across the three SPEC_ISO runs and the PROOF_GEN run from ground-truth specifications, identifying a comprehensive taxonomy of 12 error types (see Appendix B.2). 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 where the body is unsatisfiable (7); plus

Table 1. 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

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 problems 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 problems). See Table 1.

Generated specifications. In contrast, Claude-generated specifications fail 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 the most frequent and *wrong semantics* less so: an effective inversion of the ground-truth profile, suggesting that Claude accurately formalizes valid interpretations. 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 the larger number of 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 2.

Mismatches. The distribution of MISMATCH entries, where neither specification is buggy but the two are not isomorphic, reflects *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 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. See Table 3.

4.2. Implementation certification

Implementations. Claude detected issues in the generated implementations for only 7 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 D) 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. Problem 160 (`do_algebra`) ships a fixed helper `applyOp` that returns a `none` on division by zero, which is not parsed by other 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 D.

Table 2. Frequencies by type of Claude-generated specification *semantic* issues only across SPEC_ISO runs as in Table 1 and the PROOF_GEN runs from generated specs (Simple and Guided prompts). Asterisks and single-letter superscripts formatted similarly to Table 1.

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

Table 3. 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 1.

	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							

4.3. 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 formalization tasks (SPEC_GEN and IMPL_GEN), with an exception for the multi-specification SPEC_GEN run averaging around 200 seconds. Costs per solve (as reported by the Claude Agent SDK) scaled linearly with time, on average around \$0.3 for formalization tasks (\$0.5 in the multi-spec SPEC_GEN case) and \$2.0–2.5 for proving tasks. The longest-running instances that reached the 1-hour timeout stood at around \$10 per run. Appendix C.2 provides scatter plots with a more detailed breakdown of runtimes.

5. Conclusions

The experiments reported in this paper meet the contributions announced in Section 1. In our setup, Claude reaches a new state of the art on the full CLEVER pipeline: 98.8% of problems receive an arguably valid generated specification (of which 81.3% also pass the isomorphism scoring on the benchmark's correct portion), 87.5% of implementation certifications succeed from correct ground-truth specifications, and 98.1% of problems with self-consistent premises are solved end-to-end. Across all stages, Claude consistently produces well-argued classifications of its own outputs, surfacing lingering bugs in the benchmark. These findings corroborate the claim that compiler-in-the-loop agentic systems are currently the most effective for foundational program verification, and that existing program-verification benchmarks no longer adequately stress modern agentic provers.

The same findings also expose structural limitations in how the CLEVER benchmark is designed and evaluated, motivating the need for alternatives in future iterations. Indeed, for isomorphism-based scoring to remain a meaningful evaluation signal, the generated and reference specifications must come from a shared intent: this calls for more precise NL descriptions, explicit formalization hints (e.g. pre-committing to a type signature), or an iterative disambiguation pass run before formalization, in which an LLM judge surfaces and resolves multiple admissible readings until none remains. Otherwise, where formal isomorphism is intractable, the reliability of Claude's analyses under manual review suggests agentic LLM judges, complemented by property-based testing of the specifications themselves, as pragmatic alternatives. Finally, asking the model to generate an implementation and a correctness proof against its own specification is also effective at surfacing vacuous, contradictory or underconstrained specs that the equivalence check had missed, suggesting that future harnesses could combine these signals rather than rely on any of them in isolation.

References

Achim, T., Best, A., Bietti, A., Der, K., Fédérico, M., Gukov, S., Halpern-Leistner, D., Henningsgard, K., Kudryashov, Y., Meiburg, A., Michelsen, M., Patterson, R., Rodriguez, E., Scharff, L., Shanker, V., Sicca, V., Sowrirajan, H., Swope, A., Tamas, M., Tenev, V., Thomm, J., Williams, H., and Wu, L. Aristotle: IMO-level Automated Theorem Proving, October 2025.

Anthropic. Claude-agent-sdk-python, 2026.

Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Paąk, K., and Urban, J. Mizar: State-of-the-art and beyond. In Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., and Sorge, V. (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.

Bayazıt, B., Li, Y., and Si, X. A case study on the effectiveness of llms in verification with proof assistants, 2025.

Carneiro, M., Brown, C. E., and Urban, J. Automated theorem proving for metamath. In Naumowicz, A. and Thiemann, R. (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.

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

Chen, W., Wei, S., Wang, H., et al. Seed-prover: Deep and broad reasoning for automated theorem proving, 2025.

Clune, J. LeanHammer, 2026.

Dressler, O. Lean-lsp-mcp, 2026.

Feng, Y., Kafle, D., Gladshtein, V., Kurin, V., Pîrlea, G., Zhao, Q., Müller, P., and Sergey, I. Certified program synthesis with a multi-modal verifier, 2026. URL https://arxiv.org/abs/2604.16584.

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

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

Huang, Y. and Yang, L. F. Winning Gold at IMO 2025 with a Model-Agnostic Verification-and-Refinement Pipeline, September 2025.

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

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

Li, Z., Yang, Z., He, D., Zhao, H., Zhao, A., Tang, S., Yang, K., Gupta, A., Su, Z., and Jin, C. Goedel-Code-Prover: Hierarchical proof search for open state-of-the-art code verification, 2026. URL https://arxiv.org/abs/2603.19329.

Limperg, J. and From, A. H. 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.

Lin, X., Cao, Q., Huang, Y., Wang, H., Lu, J., Liu, Z., Song, L., and Liang, X. Fvel: Interactive formal verification environment with large language models via theorem proving, 2024.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J.-H., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction, August 2025.

Liu, J., Zhou, Z., Zhu, Z., Santos, M. D., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., Zhi, L., Li, J., and Li, W. Numina-Lean-Agent: An Open and General Agentic Reasoning System for Formal Mathematics, January 2026.

Moura, L. d. and Ullrich, S. 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.

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

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

Norman, C. and Avigad, J. Canonical for Automated Theorem Proving in Lean, April 2025.

Project Numina. Lean-lsp-mcp, 2026a.

Project Numina. Numina-lean-agent, 2026b.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H., Liu, Y., Gao, W., Guo, D., and Ruan, C. 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.

Slind, K. and Norrish, M. 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.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving, 2024. URL https://arxiv.org/abs/2310.04353.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., and Chaudhuri, S. CLEVER: A Curated Benchmark for Formally Verified Code Generation, October 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.

Viennot, J., Baudart, G., Arias, E. J. G., and Lelarge, M. Minif2f in rocq: Automatic translation between proof assistants – a case study, 2025.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W.-D., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., and Li, J. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning, April 2025.

Xu, Q., Wang, R., Wang, P., Li, H., and Watt, C. A minimalist proof language for neural theorem proving over isabelle/hol, 2025.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. VERINA: Benchmarking Verifiable Code Generation, October 2025.

Acknowledgements

We thank the Ethereum foundation for partially funding this research. Arora's lab is partly supported by grants from the Novo Nordisk Foundation (NNF24OC0099109), the Pioneer Centre for AI, and EU Horizon 2020 (101168951). We also gratefully acknowledge generous gifts from Microsoft and It-vest - networking universities.

A. Preliminary Experiments

In prior experiments, the authors evaluated a suite of automated provers on proof generation on the CLEVER (Thakur et al., 2025) and VERINA (Ye et al., 2025) datasets, finding that agentic paradigms substantially outperformed all 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. This 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. 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 four families of provers. 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, 2026b). The package *lean4-skills* injects domain-specific instructions, best practices, and workflow patterns into the model prompt, and bundles a wide array of proof-development utilities, including experimental subagents for specific tasks; the testing relied specifically on its `/fill-sorry` command (provided by the *lean4-theorem-proving* plugin), which systematically targets and resolves outstanding `sorry` goals. The companion MCP server *lean-lsp-mcp* interfaces with the Lean LSP and exposes search tools for relevant lemmas in the project context and Mathlib, enabling an iterative agentic loop in which the model inspects the current proof state, proposes tactics, and refines them based on compiler diagnostics. The *Numina-Lean-Agent* is a wrapper around Claude Code that combines it with *Numina-Lean-MCP* (Project Numina, 2026a), a fork of *lean-lsp-mcp* extended with semantic search and other functionality: *LeanDex*, a semantic search engine for theorems and definitions in standard Lean libraries; an *Informal Prover* (Huang & Yang, 2025) used to generate informal proof sketches; and a *Discussion Partner* tool for querying external LLMs for reasoning and planning. It was run with the provided `prompts/prompt_complete_file.txt` prompt and the setting `max_rounds=3`, allowing up to three continuations of the same Claude Code conversation before exiting.

The remaining provers served as performance baselines:

- **Monte Carlo Graph Search**: Aristotle (Achim et al., 2025), Harmonic's high-level, API-accessible proof system, achieves state-of-the-art performance on established advanced-mathematics benchmarks (most notably gold-medal performance at IMO 2025). It performs a proof-state-aware search over a hypertree of proving steps and tactics, together with an informal reasoning system generating lemmas and proof sketches, and a specialized geometry solver.

- **Whole-proof generation models**: specialized models trained or fine-tuned for formal verification, that interleave chain-of-thought reasoning with formal code generation. This family 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.

- **Symbolic tactics**: symbolic automated reasoning tools integrated directly into Lean as tactics, serving as a baseline for symbolic automation — Grind (The Lean Developers, 2024), LeanHammer (Clune, 2026), Aesop (Limperg & From, 2023), and Canonical (Norman & Avigad, 2025).

Benchmarks. As mentioned, the prior study used both the CLEVER and VERINA benchmarks. CLEVER's format and staged evaluation pipeline are described in Section 2.1; of its 167 problems (161 adapted from HUMANEVAL and 6 sample examples), only 49 carry the complete ground-truth implementation that the proof-generation distillation requires. Beyond the base CLEVER-P and VERINA-P distillations, further derivative variants were created to probe the sensitivity to problem formulation displayed by some provers:

- *-PU*, where an initial unfolding and simplification of the specification and implementation definitions (inspired by existing solutions in each dataset) is prepended to the proofs before they are passed to the provers;
- *-PM*, where Mathlib is always imported as a dependency (required by Aristotle);
- *-PF*, carrying the auto-generated fixes to the wrong statements identified in the datasets.

VERINA (Ye et al., 2025) comprises 189 manually curated tasks split into two difficulty tiers: VERINA-A (basic), 108 tasks derived from existing human-written Dafny benchmarks (MBPP-DFY, CloverBench) and translated to Lean; and VERINA-B (advanced), 81 more complex algorithmic problems adapted from platforms such as LeetCode and from challenging datasets such as LiveCodeBench. Ground-truth specifications and implementations are provided for all entries, while proof-generation solutions were supplied for only 46 of the basic entries, the remainder being left as `sorry`; this allowed the whole dataset to be used in the proof-generation evaluation.

Results. The evaluation revealed a clear performance hierarchy, as displayed by Figure 4: agentic systems based on Claude Code defined the state-of-the-art, with Aristotle trailing by a small margin; Kimina-Prover displayed much lower pass rates (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. They also displayed a marked sensitivity to problem formulation, and in particular to term unfolding, as reflected in the performance gap between the -P and -PU variants of both datasets. 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 (Table 4). 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.

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

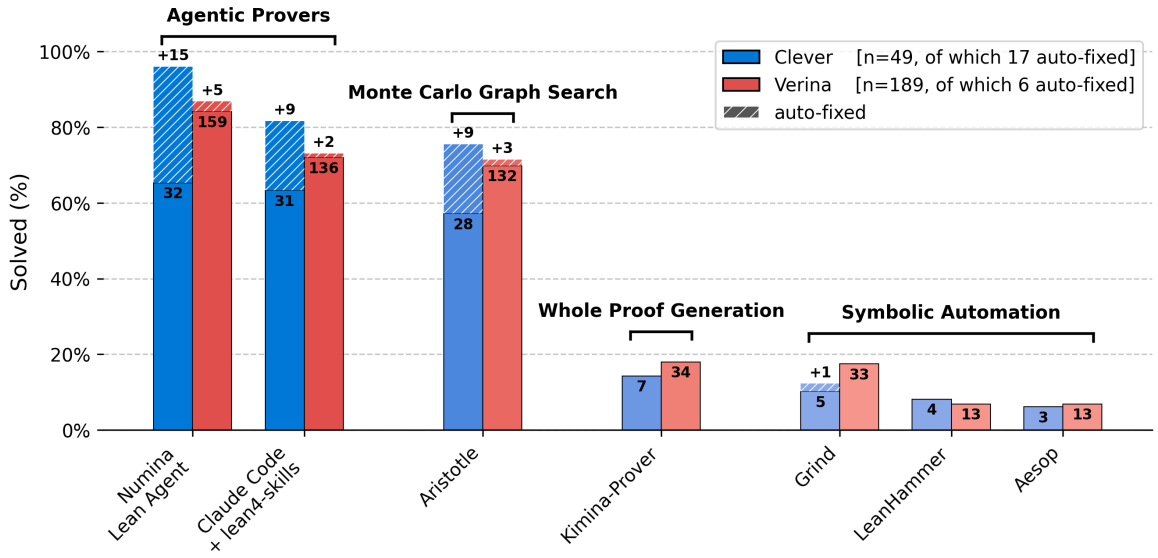

Figure 4. Overview of prover performance on the CLEVER and VERINA benchmarks. Solid bars indicate the number of tests solved on the verified-correct portions of each dataset, while the patterned bar segments above show additional solved instances obtained after correcting erroneous benchmark entries. Numbers annotated with a '+' denote the corresponding count of these additional solves.

Table 4. 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. The second column reports the benchmark variant on which each prover was evaluated (see the *Benchmarks* paragraph above).

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

17 erroneous entries in CLEVER[2]and 6 in VERINA[3]. 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, broken down by kind in Table 5, addressed wrong specification logic (9 cases), implementation bugs (6) and missing preconditions (5), with a few off-by-one and boundary errors (3).

Table 5. 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

[2]Reported publicly on 18 December 2025, see https://github.com/trishullab/clever/issues/65.

[3]Reported publicly on 26 January 2026, see https://github.com/sunblaze-ucb/verina/issues/16. A total of 16 wrong entries were found in the original version of VERINA, some of which were later fixed by the benchmark's authors on 5 January 2026 (commit e9926ecbc6d203b8e2ca008b500ccda4d09a8c6e); those errors had already been reported by Aristotle and were identified independently by us on 16 November 2025.

B. Results Classification

B.1. 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, and rigorous arguments in support have been provided.

For the SPEC_ISO task, the ISSUE flag was replaced by the following two cases:

- 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.

For the PROOF_GEN task, the ISSUE flag was replaced by the following two cases:

- MISMATCH equivalence is unprovable because the specification/implementation 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 the specification, the implementation or both are erroneous (contradicting the NL description or test cases); rigorous arguments in support have been provided.

B.2. Taxonomy of issue and mismatch kinds

Table 6. Taxonomy of specification ISSUE types.

Type	Description
Vacuous disjunct	∨ (often a biconditional disjunct like `result = none` ↔ `list = []`) trivially satisfied, admitting trivial implementations.
Precedence bug	Missing parentheses; ∧/→/↔ precedence makes the spec parse differently than intended (e.g. ∧ binds tighter than →, making the per-element constraint vacuous).
Conjunct used as guard	`(P ∧ ...)` used where `(P → ...)` was intended, making the spec unsatisfiable for invalid inputs (or, vice versa, vacuous).
Trivial existential	∃ or ∃! trivially satisfied (e.g. `∃! S, S.card = 0` witnessed by ∅; `∃ v : ℚ, |v - n| = 0` by `v = ↑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.).

Table 7. 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: ∀ over distinct values vs. distinct indices; existential parsing vs. universal verification; ∃! 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 ≥ 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`).

Table 8. 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 behaviour.
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.

C. Detailed Breakdown of Results

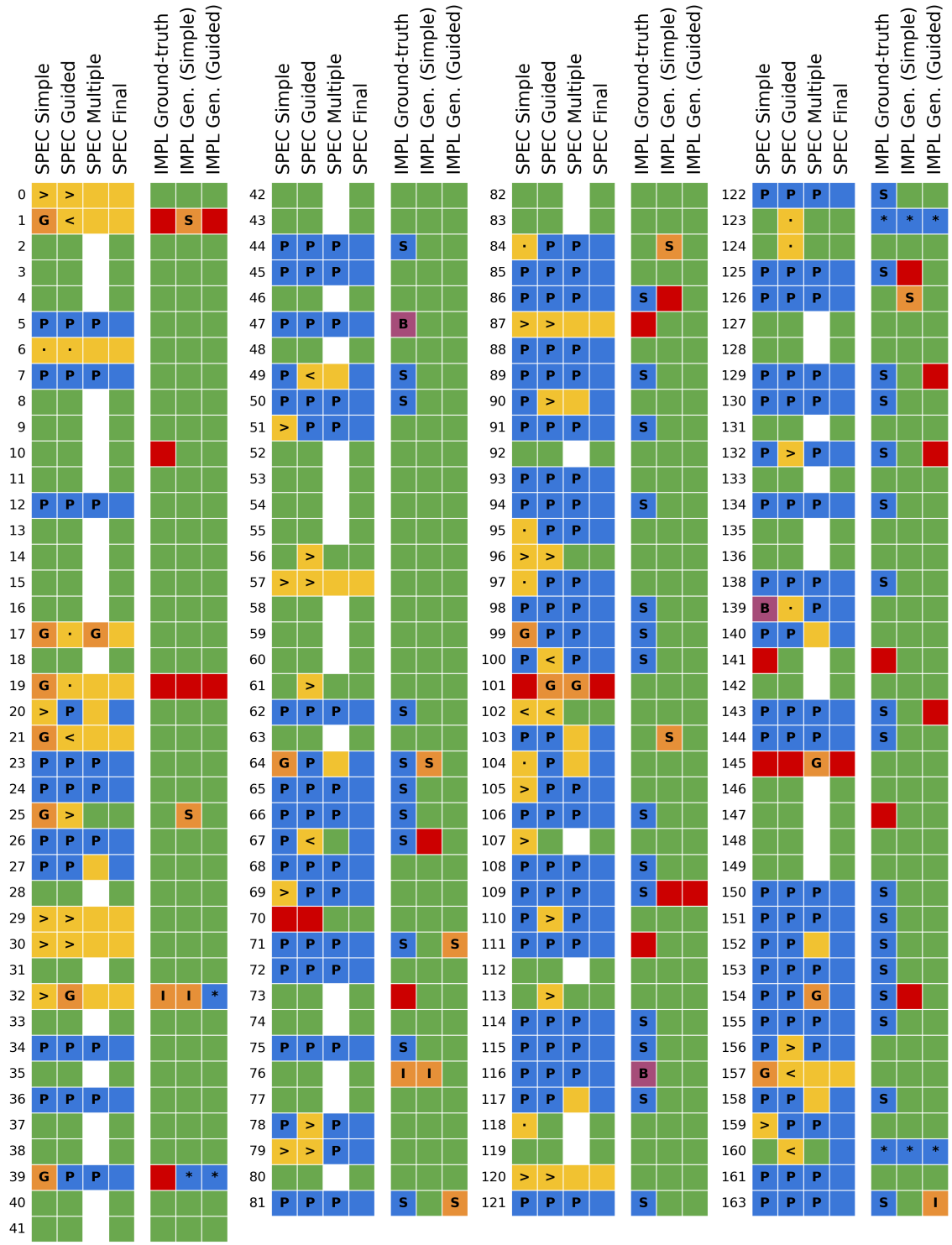

Figure 5. Output flag for each problem ID across the three SPEC_ISO runs + the final spec classification, and the PROOF_GEN runs. Color scheme corresponds to the ones in Figures 2 and 3, while one-character annotations indicate: for MISMATCH, specifications generated stronger (>), weaker (<) than reference, or neither (·); for ISSUE, bugs originating in ground-truth (P) and generated (G) specs, or both (B) in SPEC_ISO, and bugs originating in the specification (S), the implementation (I), both (B), or the fixed parts of the benchmark (⋆) in PROOF_GEN.

C.1. Multi-spec SPEC_ISO results

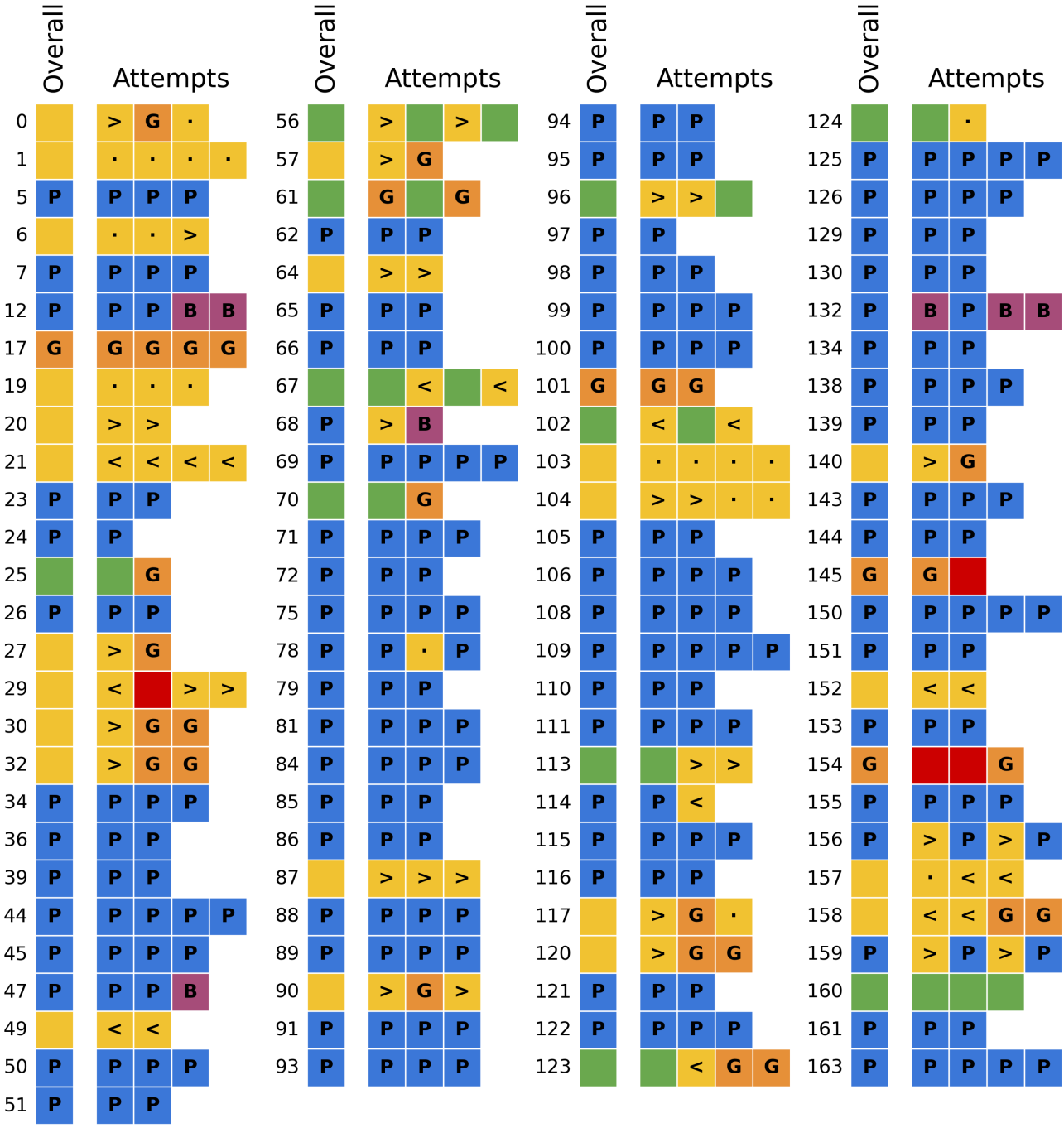

Figure 6. Breakdown of results for the Multi-spec SPEC_ISO run. Output flag for tested problems in the Multi-spec SPEC_ISO run: overall result and detailed per-spec flagging. Color scheme corresponds to the one in Figure 2, while one-character annotations indicate: for MISMATCH, specifications generated stronger (>), weaker (<) than reference, or neither (·); for ISSUE, bugs originating in ground-truth (P) and generated (G) specs, or both (B).

C.2. Runtimes and costs

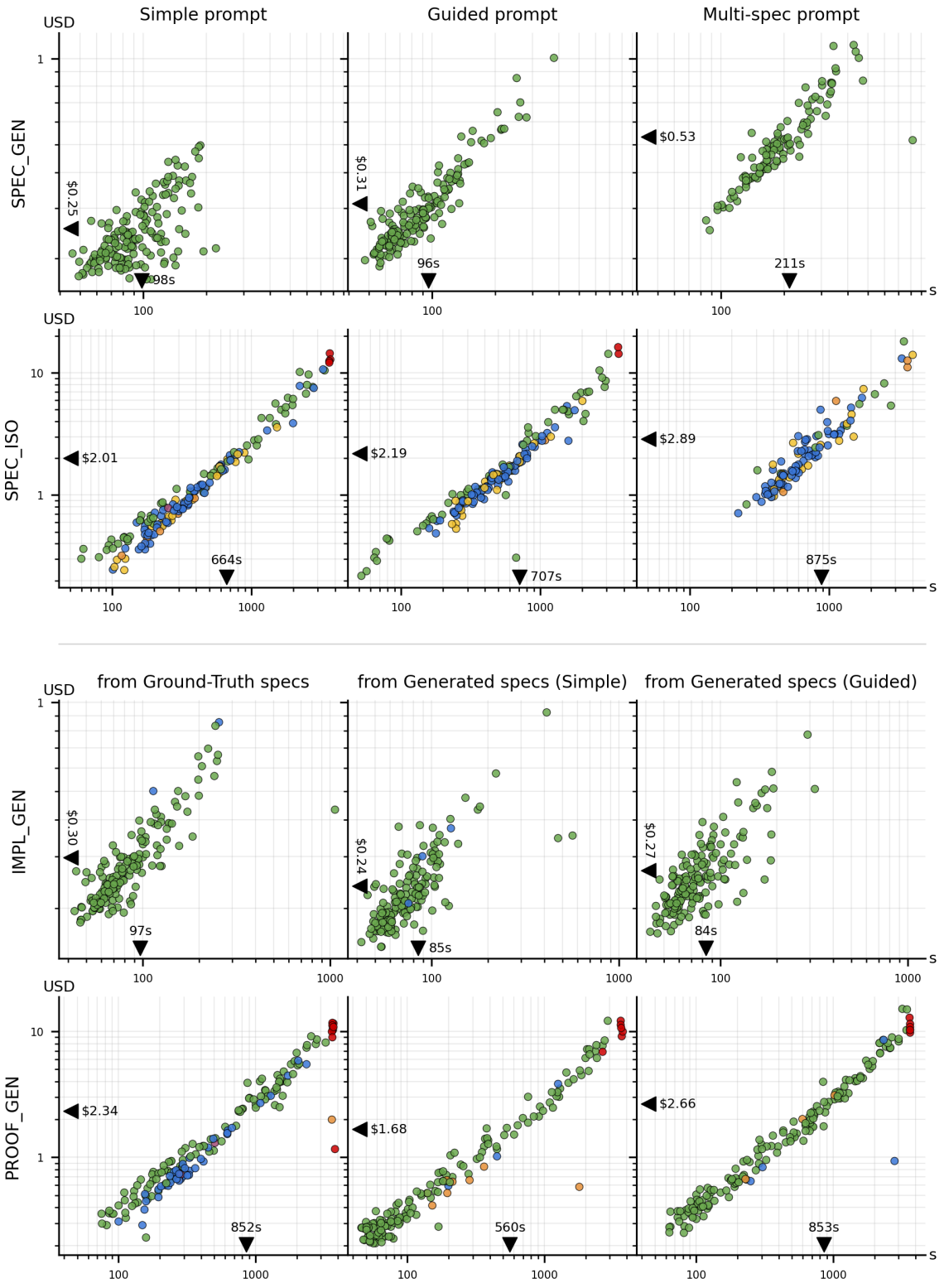

Figure 7. Scatter plot of attempt costs (USD) vs. runtimes (s), across all tasks and runs per task. Markers colored according to the schemes of Figures 2 and 3. Labeled black triangles report the average cost and runtime per run.

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 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 8 and 9).

```
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 8. Fix of CLEVER problem 32 generated by Claude Code.

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 side-step the fact that `Real` is not a computable type in Lean (as a developer comment in the entry itself

```
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] (x³-6x²+11x-6):  bisect converges to a value near root 1 with |poly| ≤ 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 + ∑|aᵢ|/|a₀| 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 9. Extract of Claude's replies explaining the edits it made in fixing problem 32.

acknowledges). Any implementation is therefore incorrect on odd-degree polynomials with no rational root, such as $x^3 + 2$. The PROOF_GEN attempts were accordingly flagged as ISSUE in either the 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.