VERINA: Benchmarking Verifiable Code Generation

Anonymous Authors[1]

Abstract

Large language models (LLMs) are increasingly integrated in software development, but ensuring correctness in LLM-generated code remains challenging and often requires costly manual review. *Verifiable code generation*—jointly generating code, specifications, and proofs of code-specification alignment—offers a promising path to address this limitation and further unleash LLMs' benefits in coding. Yet, there exists a significant gap in evaluation: current benchmarks often lack support for end-to-end verifiable code generation. In this paper, we introduce VERINA (Verifiable Code Generation Arena), a high-quality benchmark enabling a comprehensive and modular evaluation of code, specification, and proof generation as well as their compositions. VERINA consists of 189 manually curated coding tasks in Lean, with detailed problem descriptions, reference implementations, formal specifications, and extensive test suites. Our extensive evaluation of state-of-the-art LLMs reveals significant challenges in verifiable code generation, especially in proof generation, underscoring the need for improving LLM-based theorem provers in verification domains. The best model, OpenAI o4-mini, generates only 61.4% correct code, 51.0% sound and complete specifications, and 3.6% successful proofs, with one trial per task. We hope VERINA will catalyze progress in verifiable code generation by providing a rigorous and comprehensive benchmark.

1. Introduction

Large language models (LLMs) have shown strong performance in programming (Jain et al., 2025; Jimenez et al., 2024; Chen et al., 2021) and are widely adopted in tools like Cursor and GitHub Copilot to boost developer productivity (Kalliamvakou). LLM-generated code is becoming prevalent in commercial software (Peters, 2024) and may eventually form a substantial portion of the world's code. However, due to their probabilistic nature, LLMs alone cannot provide formal guarantees for the generated code. As a result, the generated code often contains bugs, such as functional errors (Wang et al., 2025) and security vulnerabilities (Pearce et al., 2022). When LLM-based code generation is increasingly adopted, these issues can become a productivity bottleneck, as they typically require human review to be resolved (Finley). Formal verification presents a promising path to establish correctness guarantees in LLM-generated code but has traditionally been limited to safety-critical applications due to high cost (Gu et al., 2016; Leroy et al., 2016; Bhargavan et al., 2013). Similarly to how they scale up code generation, LLMs have the potential to significantly lower the barrier of formal verification. By jointly generating code, formal specifications, and formal proofs of alignment between code and specifications, LLMs can offer higher levels of correctness assurance and automation in software development. This approach represents an emerging programming paradigm known as *verifiable code generation* (Sun et al., 2024; Yang et al., 2024).

Given the transformative potential of verifiable code generation, it is crucial to develop suitable benchmarks to track progress and guide future development. This is challenging because verifiable code generation involves three interconnected tasks: code, specification, and proof generation. We need to curate high-quality samples and establish robust evaluation metrics for each individual task, while also composing individual tasks to reflect real-world end-to-end usage scenarios where LLMs automate the creation of verified software directly from high-level requirements. Existing benchmarks, as listed in Table 1 and detailed in Section 2, fall short as they lack comprehensive support for all three tasks (Loughridge et al., 2025; Aggarwal et al., 2024; Chen et al., 2024), quality control (Dougherty and Mehta, 2025), robust metrics (Misu et al., 2024), or a modular design (Sun et al., 2024).

To bridge this gap, we introduce VERINA (Verifiable Code Generation Arena), a high-quality benchmark to comprehensively evaluate verifiable code generation. It consists of 189 programming challenges with detailed problem descrip-

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

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

tions, code, specifications, proofs, and comprehensive test suites. We format these problems in Lean (Moura and Ullrich, 2021), a general-purpose programming language with a rapidly growing ecosystem and applications in both formal mathematics (Mathlib community, 2020; Mathlib Community, 2022) and verification (de Medeiros et al., 2025; Hietala and Torlak, 2024).

VERINA is constructed with careful quality control. It draws problems from various sources, including MBPP (Misu et al., 2024; Austin et al., 2021), LiveCodeBench (Jain et al., 2025), and LeetCode, offering a diverse range of difficulty levels. All samples in the benchmark are manually inspected and revised to ensure clear natural language descriptions and accurate formal specifications and code implementations. Moreover, each sample also includes a comprehensive test suite with both positive and negative cases, which achieves 100% line coverage on the code implementation and passes the ground truth specification.

VERINA facilitates the evaluation of code, specification, and proof generation, along with flexible combinations of these individual tasks. We utilize the standard pass@k metric (Fan et al., 2024) with our comprehensive test suites to evaluate code generation. For proof generation, we use the Lean compiler to automatically verify their correctness. Furthermore, we develop a practical, testing-based approach based on to automatically evaluate model-generated specifications, by verifying their soundness and completeness with respect to ground truth specifications.

The high-quality samples and robust metrics of VERINA establish it as a rigorous platform for evaluating verifiable code generation. We conduct a thorough experimental evaluation of nine state-of-the-art LLMs on VERINA. Our results reveal that even the top-performing LLM, OpenAI o4-mini, struggles with verifiable code generation, producing only 61.4% correct code solutions, 51.0% sound and complete specifications, and 3.6% successful proof, given a single sample for each task. Interestingly, iterative refinement using Lean compiler feedback can increase the proof success rate to 15.3% with 64 refinement steps. However, this approach significantly raises costs and the 15.3% success rate is still insufficient. These findings underscore the challenges of verifiable code generation and highlight the critical role of VERINA in advancing the field.

2. Background and Related Work

We present works closely related to ours in Table 1 and discuss them in detail below.

Task support for verifiable code generation. Writing code, specifications, and proofs for a verified software component is time-consuming when done manually. Although various studies have explored using LLMs to automate these tasks, they primarily focus on individual aspects, which are insufficient for fully automating end-to-end verifiable code generation. Benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) have sparked impressive progress on LLM-based code generation but do not handle formal specifications or proofs. Many verification-focused efforts target only one or two tasks, while assuming the other elements are provided by the human user. For example, DafnyBench (Loughridge et al., 2025) and miniCodeProps (Lohn and Welleck, 2024) are two benchmarks designed exclusively for proof generation. Moreover, AutoSpec (Wen et al., 2024) and SpecGen (Ma et al., 2025) infer specifications and proofs from human-written code.

To the best of our knowledge, Dafny-Synthesis (Misu et al., 2024) and Clover (Sun et al., 2024) are the only two works that cover all three tasks, like VERINA. However, they target automated theorem proving using Dafny (Leino, 2010), while VERINA leverages interactive theorem proving in Lean. Moreover, they have relatively small numbers of human-written samples (50 and 62 respectively). In contrast, VERINA provides 189 high-quality samples with varying difficulty levels.

Automated and interactive theorem proving. A major challenge in formal verification and verifiable code generation lies in tooling. Verification-oriented languages like Dafny (Leino, 2010) and Verus (Lattuada et al., 2023) leverage SMT solvers for automated theorem proving (De Moura and Bjørner, 2008; Barrett and Tinelli, 2018) and consume only proof hints, such as loop invariants (Pei et al., 2023) and assertions (Mugnier et al., 2025). However, SMT solvers handle only limited proof domains and behave as black boxes, which can make proofs brittle and hard to debug (Zhou et al., 2023). Interactive theorem proving (ITP) systems like Lean provide a promising target for verifiable code generation with LLMs. ITPs support constructing proofs with explicit intermediate steps. This visibility enables LLMs to diagnose errors, learn from unsuccessful steps, and iteratively refine their proofs. While ITPs traditionally require humans to construct proofs, recent work shows that LLMs can generate proofs at human level in math competitions (Google DeepMind, 2024). To our knowledge, the only existing verification benchmarks in Lean are miniCodeProps (Lohn and Welleck, 2024) and FVAPPS (Dougherty and Mehta, 2025). miniCodeProps translates 201 Haskell programs and their specifications into Lean but is designed for proof generation only. FVAPPS contains 4,715 Lean programs with LLM-generated specifications from a fully automated pipeline that lacks human validation and quality control. In contrast, VERINA provides high-quality, human-verified samples and captures all three foundational tasks in verifiable code generation.

Task compositionality. A key strength of VERINA is its

Table 1: A comparison of VERINA with related works on LLMs for code generation and verification. We characterize whether each work supports the three foundational tasks for end-to-end verifiable code generation: CodeGen, SpecGen, ProofGen (Section 4.1). ● means fully supported, ◐ means partially supported, ○ means unsupported. If ProofGen is supported, we specify the proving style: automated theorem proving (ATP) or interactive theorem proving (ITP). For works supporting multiple tasks, we annotate if these tasks are supported in a modular and composable manner. Overall, VERINA offers more comprehensive and high-quality benchmarking compared to prior works.

		CodeGen	SpecGen	ProofGen	Proving Style	Compositionality	Language
Benchmarks	HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021)	●	○	○	–	–	Python
	Dafny-Synthesis (Misu et al., 2024)	●	◐	●	ATP	✗	Dafny
	DafnyBench (Loughridge et al., 2025)	○	○	●	ATP	–	Dafny
	miniCodeProps (Lohn and Welleck, 2024)	○	○	●	ITP	–	Lean
	FVAPPS (Dougherty and Mehta, 2025)	●	○	●	ITP	✗	Lean
Techniques	nl2postcond (Endres et al., 2024)	○	●	○	–	–	Python, Java
	Clover (Sun et al., 2024)	●	●	●	ATP	✗	Dafny
	AlphaVerus (Aggarwal et al., 2024)	●	○	●	ATP	✗	Rust
	AutoSpec (Wen et al., 2024)	○	●	●	ATP	✗	C/C++
	SpecGen (Ma et al., 2025)	○	●	●	ATP	✗	Java
	SAFE (Chen et al., 2024)	○	○	●	ATP	✗	Rust
	AutoVerus (Yang et al., 2025)	○	○	●	ATP	–	Rust
	Laurel (Mugnier et al., 2025)	○	○	●	ATP	–	Dafny
	Pei et al. (2023)	○	○	●	ATP	–	Java
	Baldur (First et al., 2023), Selene (Zhang et al., 2024)	○	○	●	ITP	–	Isabelle
	Rango (Thompson et al., 2025), PALM (Lu et al., 2024)	○	○	●	ITP	–	Coq
	VERINA	●	●	●	ITP	✓	Lean

modular design, which enables flexible evaluation of not only individual tasks but also their combinations. This compositionality captures diverse real-world scenarios—from specification-guided code generation to end-to-end verifiable code generation—enabling a comprehensive assessment of different aspects of verifiable code generation. This modularity also facilitates targeted research on specific weaknesses, such as improving proof generation. On the contrary, all other prior works lack compositionality. For example, Dafny-Synthesis (Misu et al., 2024) and Clover (Sun et al., 2024) mix specification and proof generation into a single task, lacking support for separate evaluation of each.

3. VERINA: Data Format, Construction, and Quality Assurance

We describe the VERINA benchmark, its data construction pipeline, and quality assurance measures.

3.1. Overview and Data Format

VERINA consists of 189 programs, annotated with natural language descriptions, code, specifications, proofs, and test cases. The code, specification, and proof are all written in Lean. An example is illustrated in Figure 1, consisting of:

- *Natural language description (Line 1–4)*: informal description of the programming problem, capturing the intent of the human developer.
- *Code (Line 6–8)*: ground truth code implementation that solves the programming problem.
- *Specification (Line 10–18)*: ground truth formal specification for the programming problem. It consists of a pre-condition, which states properties the inputs must satisfy, and a post-condition, which states desired relationship between inputs and outputs.
- *Proof (Optional, Line 20–22)*: formal proof establishing that the code satisfies the specification. Ground truth proofs are optional in VERINA, as they are not required for evaluation. Model-generated proofs can be checked by Lean directly. Nevertheless, we invest significant manual effort in writing proofs for 46 out of 189 examples as they help quality assurance (Section 3.2).
- *Test suite (Line 24–29)*: a comprehensive suite of both positive and negative test cases. Positive tests are valid input-output pairs that meet both the pre-condition and the post-condition. Negative tests are invalid inputs-output pairs, which means either the inputs violate the pre-condition or the output violates the post-condition. These test cases are useful for evaluating model-generated code and specifications, as detailed in Section 4.1. They are formatted in Lean during evaluation.

Benchmark statistics. Table 2 presents key statistics of VERINA. Natural language descriptions have a median length of 110 words, ensuring they are both informative and detailed. Code ranges up to 38 lines and specifications up to 62 lines, demonstrating that VERINA captures complex tasks. With a median of 5 positive tests and 12 negative tests per instance, the constructed test suites provide strong evidence for the high quality and correctness of VERINA.

```
-- Description of the coding problem in natural language
-- Remove an element from a given array of integers at a specified index. The resulting array should
-- contain all the original elements except for the one at the given index. Elements before the
-- removed element remain unchanged, and elements after it are shifted one position to the left.

-- Code implementation
def removeElement (s : Array Int) (k : Nat) (h_precond : removeElement_pre s k) : Array Int :=
  s.eraseIdx! k

-- Pre-condition
def removeElement_pre (s : Array Int) (k : Nat) : Prop :=
  k < s.size -- the index must be smaller than the array size

-- Post-condition
def removeElement_post (s : Array Int) (k : Nat) (result: Array Int) (h_precond : removeElement_pre s k) : Prop :=
  result.size = s.size - 1 ∧ -- Only one element is removed
  (∀ i, i < k → result[i]! = s[i]!) ∧ -- The elements before index k remain unchanged
  (∀ i, i < result.size → i ≥ k → result[i]! = s[i + 1]!) -- The elements after index k are shifted by one position

-- Proof
theorem removeElement_spec (s: Array Int) (k: Nat) (h_precond : removeElement_pre s k) :
  removeElement_post s k (removeElement s k h_precond) h_precond := by sorry -- The proof is omitted for brevity

-- Test cases
(s : #[1, 2, 3, 4, 5]) (k : 2) (result : #[1, 2, 4, 5]) -- Positive test with valid inputs and output
(s : #[1, 2, 3, 4, 5]) (k : 5) -- Negative test: inputs violate the pre-condition at Line 12
(s : #[1, 2, 3, 4, 5]) (k : 2) (result : #[1, 2, 4]) -- Negative test: output violates the post-condition at Line 16
(s : #[1, 2, 3, 4, 5]) (k : 2) (result : #[2, 2, 4, 5]) -- Negative test: output violates the post-condition at Line 17
(s : #[1, 2, 3, 4, 5]) (k : 2) (result : #[1, 2, 4, 4]) -- Negative test: output violates the post-condition at Line 18
```

Figure 1: An example instance of VERINA, consisting of a problem description, code implementation, specifications (pre-condition and post-condition), a proof (optional), and comprehensive test cases. Note that we select this instance for presentation purposes and VERINA contains more difficult ones.

Table 2: Statistics of VERINA.

Metric	Median	Max
# Words in Description	110	296
LoC for Code	9	38
LoC for Spec.	4	62
# Positive Tests	5	13
# Negative Tests	12	27

3.2. Benchmark Construction and Quality Assurance

VERINA consists of two subsets sourced from different origins: VERINA-BASIC and VERINA-ADV. For VERINA-BASIC, the maximum (median) LoC for code and specification are 26 (6) and 17 (2), respectively. For VERINA-ADV, they are 38 (16) and 62 (7). This underscores the diversity and varying difficulty levels between VERINA-BASIC and VERINA-ADV. We employ a meticulous data curation process that combines careful translation, thorough manual review, and automated mechanisms, leading to a rigorous and high-quality benchmark for verifiable code generation.

VERINA-BASIC: translation from human-written Dafny code. We first consider MBPP-DFY-50 (Misu et al., 2024), which contains MBPP (Austin et al., 2021) coding problems paired with human-verified solutions in Dafny. Each instance contains a natural language task description, code, specifications, proof, and test cases. We manually translated 49 problems into Lean, refining and verifying each translation. To extend the benchmark, we added 59 more human-authored Dafny instances from CloverBench (Sun et al., 2024). These were translated into Lean using OpenAI o3-mini with few-shot prompting based on our manual translations, followed by manual inspection and correction. Overall, the coding difficulty in VERINA-BASIC, abstracting away language differences, is comparable to MBPP.

VERINA-ADV: writing Lean code from scratch. VERINA-ADV enhances the diversity of VERINA by incorporating more advanced coding problems and solutions. They were adapted from student submissions to a lab assignment in a course on theorem proving and program verification. Students, both undergraduate and graduate, were encouraged to source problems from platforms like LeetCode or more challenging datasets such as LiveCodeBench (Jain et al., 2025). They formalized and solved these problems in Lean, providing all necessary elements in VERINA's format (Section 3.1). We carefully selected the most suitable and high-quality submissions, resulting in 81 benchmark instances. In addition, we manually reviewed and edited the submissions to ensure their correctness.

Quality assurance. During the data collection process, we consistently enforce various manual and automatic mechanisms to ensure the high quality of VERINA:

- *Detailed problem descriptions*: The original problem descriptions, such as those from MBPP-DFY-50, can be short and ambiguous, making them inadequate for specification generation. To resolve this, we manually enhanced the descriptions by clearly outlining the high-level intent, specifying input parameters with explicit type information, and detailing output specifications.
- *Full code coverage with positive tests*: Beyond the original test cases, we expanded the set of positive tests to ensure

that they achieve full line coverage on the ground truth code. We created these additional tests both manually and with LLMs. We leveraged the standard `coverage.py` tool to verify complete line coverage, since Lean lacks a robust coverage tool. For Python reference implementations, we either used the original MBPP code or generated an implementation from the enhanced problem description via OpenAI's o4-mini with manual validation.

- *Full test pass rate on ground truth specifications*: We evaluated the ground truth specifications against our comprehensive test suites. All ground truth specifications successfully pass their respective positive tests, confirming the quality of the specifications in VERINA.
- *Necessary negative tests*: We mutated each positive test case to construct at least three different negative tests that violate either the pre- or the post-condition, except when the function's output has boolean type, in which case only a single negative test can be created. We made sure that our ground truth code and specifications do not pass these negative tests.
- *Preventing trivial code generation*: VERINA allows providing ground truth specifications as an optional input for the code generation task (discussed in Section 4.1). We crafted all ground truth specifications such that they cannot be directly used to solve the coding problem. This prevents LLMs from generating an implementation trivially equivalent to the specification. As a result, the model must genuinely demonstrate semantic comprehension of the reference specification and non-trivial reasoning to generate the corresponding implementation.
- *Manual review and edits*: Each benchmark instance was manually reviewed by at least two authors, carefully editing them to ensure correctness and high quality.

4. Evaluating Verifiable Code Generation Using VERINA

VERINA enables comprehensive evaluation of verifiable code generation, covering foundational tasks—code, specification, and proof generation—and their combinations to form an end-to-end pipeline from natural language descriptions to verifiable code. We also introduce a novel framework for a reliable automatic evaluation of model-generated specifications.

4.1. Foundational Tasks and Metrics

As shown in Figure 2, all three foundational tasks include natural language descriptions and function signatures (Lines 7, 11, and 15 in Figure 1) as model inputs, which captures human intent and enforces consistent output formats, facilitating streamlined evaluation.

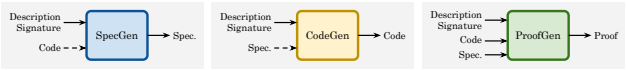

Figure 2: VERINA's three foundational tasks. Dashed arrows represent optional inputs.

Specification generation (SpecGen). Given a description, signature, and *optionally* code implementation, the model generates a formal specification. Specifications must accurately capture human intent. Let ϕ denote the set of correct programs that satisfy human intent and $\hat{\phi}$ the set that aligns with the generated specification. An ideal specification should achieve $\hat{\phi} = \phi$, which entails two properties—(i) *soundness* ($\hat{\phi} \subseteq \phi$): it is "small enough" to cover only correct programs, and (ii) *completeness* ($\phi \subseteq \hat{\phi}$): it is "large enough" to cover all correct programs.

In practice, two challenges arise for evaluating $\hat{\phi}$. First, we must capture ϕ formally. VERINA addresses this by leveraging high-quality ground truth specifications (see Section 3.2) and comprehensive test suites. Second, we need to assess the relationship between $\hat{\phi}$ and ϕ to establish soundness and completeness. Since specifications consist of pre-conditions and post-conditions, let P and $\hat{P}$ denote the ground truth and model-generated pre-conditions, respectively, and Q and $\hat{Q}$ the corresponding post-conditions. In VERINA, we define the soundness and completeness of $\hat{P}$ and $\hat{Q}$ as follows:

- $\hat{P}$ is sound iff $\forall \overline{x}.P(\overline{x}) \Rightarrow \hat{P}(\overline{x})$, where $\overline{x}$ are the program's input values. Given the same post-condition (e.g., Q), it is more difficult for a program to satisfy $\hat{P}$ than P. This is because $\hat{P}$ allows more inputs, which the program must handle to meet the post-condition. As a result, the set of programs accepted by $\hat{P}$ a subset of those accepted by P.
- $\hat{P}$ is complete iff $\forall \overline{x}.\hat{P}(\overline{x}) \Rightarrow P(\overline{x})$. Given the same post-condition, the set of programs accepted by $\hat{P}$ is now a superset of those accepted by P, since $\hat{P}$ is more restrictive than P.
- $\hat{Q}$ is sound iff $\forall \overline{x}, y.P(\overline{x}) \wedge \hat{Q}(\overline{x}, y) \Rightarrow Q(\overline{x}, y)$, where y is the output value. For any valid inputs w.r.t. P, the set of output accepted by $\hat{Q}$ is a subset of those accepted by Q, establishing soundness.
- Symmetrically, $\hat{Q}$ is complete iff $\forall \overline{x}, y.P(\overline{x}) \wedge Q(\overline{x}, y) \Rightarrow \hat{Q}(\overline{x}, y)$.

To evaluate SpecGen, we need automatic and robust mechanisms to check if the above relationships hold. Formally proving them is difficult, as they may contain nested quantifiers and complex program properties. LLM-based provers are ineffective in the verification domain, as shown in Section 5, making them unreliable for this use case. Another

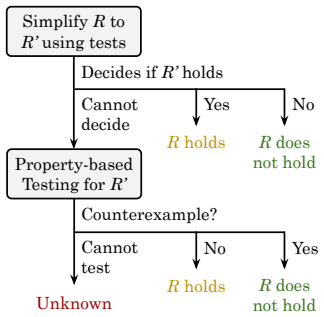

Figure 3: Our evaluator for specification generation.

approach is to convert these relationships into ATP; however, existing tools do not adequately model the necessary Lean features (Mohamed et al., 2025). To overcome these limitations, we leverage a practical testing-based evaluation framework using our comprehensive test suites, as shown in Figure 3. We formalize a given soundness or completeness relationship, denoted by R, in Lean. Instead of proving R for universally quantified input and output variables, we check R against concrete values in test cases. For example, to evaluate $\hat{Q}$'s soundness, we check if $P(\overline{x}) \wedge \hat{Q}(\overline{x}, y) \Rightarrow Q(\overline{x}, y)$ holds for all test cases $(\overline{x}, y)$ in our test suite. We denote this simplified version of R as R'. For many cases, e.g., the specification in Figure 1, Lean can automatically determine if R' holds (Selsam et al., 2020) and we return the corresponding result. Otherwise, we employ property-based testing with the `plausible` tactic in Lean (Lean Prover Community, 2024). It generates diverse inputs specifically targeting the remaining universally and existentially quantified variables in R', extensively exploring the space of possible values to test R'. In Appendix A.4, we provide a detailed description on how we implement these metrics in Lean.

Since our evaluator is based on testing, it can prove that R does not hold through counterexamples, as highlighted in green in Figure 3. While it cannot formally establish R holds, it remains highly robust in this regard, due to our comprehensive test suite with both positive and negative tests, which achieve full coverage on ground truth code implementations. Lean's property-based testing cannot handle a small number of complicated relationships, for which our evaluator returns unknown. To further enhance the accuracy of our metric, we repeat our evaluation framework in Figure 3 to check $\neg R$. We compare the evaluator outcomes on R and $\neg R$, and select the more accurate result as the final output. Our final metrics for SpecGen include individual pass@k (Chen et al., 2021) scores for soundness and completeness of all generated pre-conditions and post-conditions, as well as aggregated scores that soundness and completeness hold simultaneously for pre-condition, post-condition, and the complete specification. Since the evaluation of the specification may return unknown, we plot error bars indicating the lower bound (treating unknown as R does not hold) and upper bound (treating as R holds).

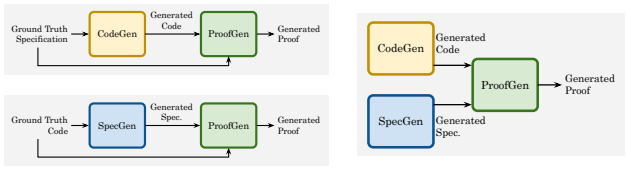

Figure 4: Combinations of VERINA's foundational tasks: specification-guided code generation (*top left*), specification inference from code (*bottom left*), and end-to-end verifiable code generation (*right*). Natural language descriptions and function signatures are omitted in the figure for brevity.

To illustrate our metric, consider the ground truth pre-condition `k < s.size` at Line 12 of Figure 1, and model-generated pre-condition `k < s.size - 1` and `k < s.size + 1`. `k < s.size - 1` can be determined as unsound using the positive test `(s : #[1, 2, 3, 4, 5]) (k : 4)`, while `k < s.size + 1` is incomplete based on the negative test `(s : #[1, 2, 3, 4, 5]) (k : 5)`. We more examples of our metrics for specification generation in Appendix C.

Code generation (CodeGen). Given a natural language description, function signature, and *optionally* specification, the model generates code implementing the desired functionality. Following standard practice, we evaluate the generated code using positive test cases in VERINA and reporting the pass@k metric defined by Chen et al. (2021). In Section 4.2, we will explore evaluating the code by proving its correctness with respect to the formal specification.

Proof generation (ProofGen). Given a description, signature, code, and specification, the model generates a formal proof in Lean to establish that the code satisfies the specification. This task evaluates the model's ability to reason about code behavior and construct logically valid arguments for correctness. We use Lean to automatically check the validity of generated proofs, and proofs containing placeholders (e.g., the `sorry` tactic) are marked as incorrect.

4.2. Task Combinations

VERINA enables combining the three foundational tasks to evaluate various capabilities in verifiable code generation. These combined tasks reflect real-world scenarios where developers utilize the model to automatically create verified software in an end-to-end manner. Such modularity and compositionality highlight the generality of VERINA, which encompasses various tasks studied in previous work (Table 1). Three examples of combined tasks are (Figure 4):

- *Specification-Guided Code Generation*: Given a natural language description, function signature, and the *ground truth* specification, the model first generates the code and

then proves that the code satisfies the specification. This aligns with tasks explored in FVAPPS (Dougherty and Mehta, 2025) and AlphaVerus (Aggarwal et al., 2024).

- *Specification Inference from Code*: In some cases, developers may have the code implementation and want the model to annotate it with a formal specification and prove their alignment. This corresponds to the setting in AutoSpec (Wen et al., 2024), SpecGen (Ma et al., 2025), and SAFE (Chen et al., 2024).
- *End-to-End Verifiable Code Generation*: For an even higher degree of automation, developers might start with only a high-level problem description in natural language and instruct the model to generate code and specification independently, and then generate the proof. This captures the scenario in Dafny-Synthesis (Misu et al., 2024) and Clover (Sun et al., 2024).

In these task combinations, a crucial design consideration is the dependency between code and specification. For example, in specification-guided code generation, it is important to assess how beneficial the ground truth specification is beyond the natural language description, which already captures the developer's intent. Additionally, for end-to-end verifiable code generation, it is essential to decide the order of the CodeGen and SpecGen modules—whether to make SpecGen dependent on the output of CodeGen, place SpecGen before CodeGen, or run them independently (as in Figure 4). We experimentally explore these design choices using VERINA in Section 5.

In end-to-end verifiable code generation, it is crucial that the model generates code and specification *independently*, rather than sequentially. Otherwise, it may exploit shortcuts by producing definitionally equivalent code-specification pairs, making the proof task trivial. When designing VERINA's tasks, we enforce independence—for example, the model cannot access the code when generating the specification. While we do not yet check for definitional equivalence (e.g., using BEq (Liu et al., 2025)), we leave this as an important direction for future work.

5. Experimental Evaluation

Experimental setup. We evaluate a diverse set of nine state-of-the-art LLMs on VERINA. We leverage 2-shot prompting to enhance output format adherence, with the 2-shot examples excluded from the final benchmark. For each task, we primarily report the pass@1 metric (Chen et al., 2021). We provide detailed input prompts, output formats, and LLM setups in Appendix A.

All foundational tasks are challenging, especially ProofGen. Figure 5 shows a clear difficulty hierarchy across the three foundational tasks. Code generation achieves the highest success rates across models, followed by specification generation, while proof generation remains the most challenging with pass@1 rates below 3.6% for all models. All three tasks pose significant challenges for current LLMs, with constructing Lean proofs that the implementation satisfies the specification being particularly hard and requiring specialized theorem proving capabilities. This also means that for any combined task involving ProofGen, LLMs' performance will be heavily bottlenecked by the ProofGen subtask. Among the evaluated models, o4-mini, GPT 4.1, Claude Sonnet 3.7, and Gemini 2.5 Flash demonstrate relatively stronger performance across tasks. We report detailed results on pre-condition and post-condition soundness and completeness in Appendix B, where we observe that generating sound and complete post-conditions is generally more difficult than pre-conditions.

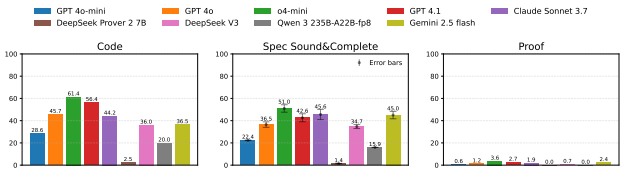

Figure 5: pass@1 performance of LLMs on VERINA's three foundational tasks.

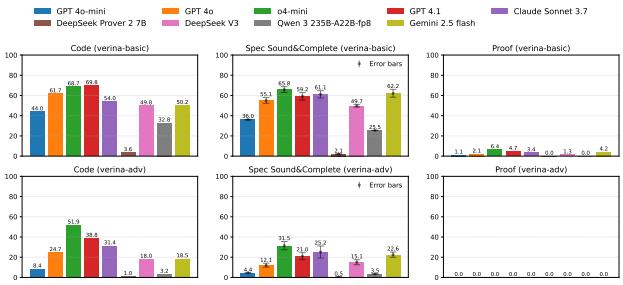

Figure 6: pass@1 performance on three foundational tasks for VERINA-BASIC and VERINA-ADV.

VERINA-ADV is much more challenging than VERINA-BASIC. The comparison between VERINA-BASIC and VERINA-ADV in Figure 6 reveals substantial difficulty gaps on all three tasks. This demonstrates that problem complexity significantly impacts all aspects of verifiable code generation, and VERINA-ADV provides a valuable challenge for advancing future research in this domain.

Iterative proof refinement shows meaningful improvements. For ProofGen task, besides pass@1, we also extend the evaluation of the 4 best performing LLMs (o4-mini, GPT 4.1, Claude Sonnet 3.7, Gemini 2.5 Flash) to further investigate LLMs' theorem proving capabilities. We evaluate them with iterative proof refinement, where the evaluated model receives Lean verifier error messages and is prompted to revise its proof, and with direct generation, where the

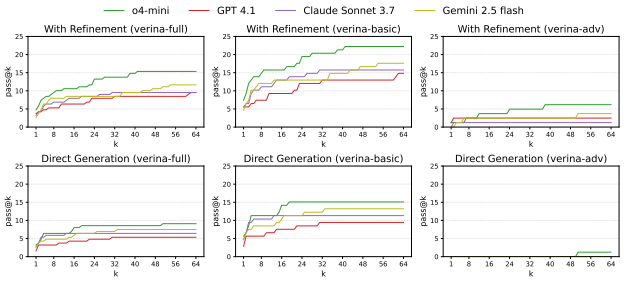

Figure 7: pass@k performance of selective LLMs on the ProofGen tasks in VERINA using proof refinement (first row) and direct generation (second row).

evaluated model generates responses independently in each iteration. For both method, we report pass@k, the success rate after k rounds of iterations, for k up 64. This metric investigates how much additional interaction helps repair the proof that a single-pass generation would miss, and whether providing Lean verifier feedback improves success rates compared to independent generation attempts.

As shown in Figure 7, iterative proof refinement yields meaningful improvements on simpler problems, with o4-mini improving from 7.4% to 22.2% on VERINA-BASIC after 64 iterations. However, these gains are substantially less on VERINA-ADV (1.2% to 6.2%), indicating that naive refinement strategies are insufficient for complex proving tasks. Furthermore, comparing refinements and direct generation without error messages demonstrates the clear value of Lean verifier feedback.

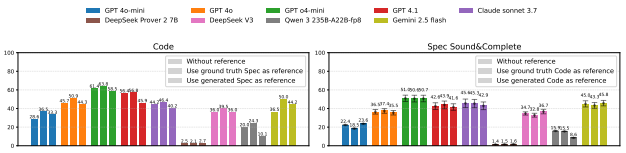

Figure 8: Impact of contextual information (reference code or specification input) on CodeGen and SpecGen performance.

Providing ground truth specification benefits CodeGen. Providing ground truth specifications as context consistently improves CodeGen performance across models. Since the ground truth specifications cannot be used directly as code (as explained in 3.2), all CodeGen improvements rely on semantic understanding of the reference specification. On the contrary, providing ground truth code as context shows minimal or negative improvement for SpecGen. While it is possible for LLMs to directly use the ground truth code in the specification, manual inspection of our evaluation results reveals no evidence of such behaviors. This is likely because using code as specification is uncommon in standard development practices, and our prompts A.3 ask LLMs to focus on constraining code behavior rather than replicating implementation details. The asymmetry in using ground truth information for CodeGen versus SpecGen suggests that formal specifications effectively constrain and guide code synthesis, while verbose code implementations may introduce noise to or over-constrain specification generation rather than providing helpful guidance. Furthermore, when using LLM-generated code or specifications as context instead of ground truth, performance generally degrades. The generated artifacts can be of insufficient quality to serve as reliable reference. This suggests that combined tasks, where LLMs must generate both code and specifications jointly, might be significantly more challenging than individual tasks in isolation.

Qualitative case studies. We present detailed qualitative case studies with analysis of failure modes and success patterns across different tasks in Appendix C.

6. Conclusion and Discussion

We have introduced VERINA, a comprehensive benchmark comprising 189 carefully curated examples with detailed task descriptions, high-quality codes and specifications in Lean, and extensive test suites with full line coverage. This benchmark enables systematic assessment of various verifiable code generation capabilities, and our extensive evaluation result presents substantial challenges that expose limitations of state-of-the-art language models on verifiable code generation tasks. We hope that VERINA will serve as a valuable resource by providing both a rigorous evaluation framework and clear directions towards more reliable and formally verified automated programming systems.

Limitations and future work. Despite advancing the state-of-the-art in benchmarking verifiable code generation, VERINA has several limitations. First, its size (189 examples) is modest, scaling to a larger dataset suitable for finetuning likely requires automated annotation with LLM assistance. Second, it emphasizes simple, standalone tasks—well-suited for benchmarking but not fully representative of complex real-world verification projects (Klein et al., 2009; Leroy et al., 2016). Third, while existing provers have limited capabilities for handling complex soundness/completeness relationships, our SpecGen metric (Section 4.1) could be improved by potential more capable provers in the future, including those based on LLMs or SMT solvers, to prove soundness/completeness relationships. Finally, while Lean programs in VERINA are newly written, the underlying task topics are drawn from widely used sources, posing a risk of data contamination.

References

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *Interna-*

tional Conference on Learning Representations (ICLR), 2025. 1, 2, 4, 11

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024. 1

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 1, 2, 3, 6, 7, 11

Eirini Kalliamvakou. Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness. Accessed: 2025-05-10. 1

Jay Peters. More than a quarter of new code at Google is generated by AI. https://www.theverge.com/2024/10/29/24282757/google-new-code-generated-ai-q3-2024, 2024. 1

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models . In *International Conference on Software Engineering (ICSE)*, 2025. 1

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of Github Copilot's code contributions. In *Symposium on Security and Privacy*, 2022. 1

Klint Finley. How developers spend the time they save thanks to AI coding tools. https://github.blog/ai-and-ml/generative-ai/how-developers-spend-the-time-they-save-thanks-to-ai-coding-tools/. Accessed: 2025-05-10. 1

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. 1

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*, 2016. 1, 8

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *Symposium on Security and Privacy*, 2013. 1

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, 2024. 1, 2, 3, 4, 7, 11

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin Lauter, Swarat Chaudhuri, and Dawn Song. Formal mathematical reasoning: A new frontier in AI. *arXiv preprint arXiv:2412.16075*, 2024. 1

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. 1, 2, 3

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. *arXiv preprint arXiv:2412.06176*, 2024. 1, 3, 7

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, et al. Automated proof generation for Rust code via self-evolution. In *International Conference on Learning Representations (ICLR)*, 2024. 1, 3, 7

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation. *arXiv preprint arXiv:2502.05714*, 2025. 1, 2, 3, 7

Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. Towards AI-assisted synthesis of verified Dafny methods. *Proceedings of the ACM on Software Engineering*, 2024. 1, 2, 3, 4, 7, 11

Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction (CADE)*, 2021. 2

Mathlib community. The Lean mathematical library. In *Certified Programs and Proofs (CPP)*, 2020. 2

Mathlib Community. Completion of the liquid tensor experiment. https://leanprover-community.github.io/blog/posts/lte-final/, 2022. 2

Markus de Medeiros, Muhammad Naveed, Tancrede Lepoint, Temesghen Kahsai, Tristan Ravitch, Stefan Zetzsche, Anjali Joshi, Joseph Tassarotti, Aws Albarghouthi, and Jean-Baptiste Tristan. Verified foundations for differential privacy. In *Programming Language Design and Implementation (PLDI)*, 2025. 2

Kesha Hietala and Emina Torlak. Lean into verified software development. https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/, 2024. 2

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. 2, 3, 4

Wen Fan, Marilyn Rego, Xin Hu, Sanya Dod, Zhaorui Ni, Danning Xie, Jenna DiVincenzo, and Lin Tan. Evaluating the ability of large language models to generate verifiable specifications in verifast. *arXiv preprint arXiv:2411.02318*, 2024. 2

Evan Lohn and Sean Welleck. miniCodeProps: a minimal benchmark for proving code properties. *arXiv preprint arXiv:2406.11915*, 2024. 2, 3

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 2024. 3

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification (CAV)*, 2024. 2, 3, 7

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated generation of formal program specifications via large language models. In *International Conference on Software Engineering (ICSE)*, 2025. 2, 3, 7

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. AutoVerus: Automated proof generation for Rust code. In *International Conference on Learning Representations (ICLR)*, 2025. 3

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM on Programming Languages*, 2025. 2, 3

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning (ICML)*, 2023. 2, 3

Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. 3

Lichen Zhang, Shuai Lu, and Nan Duan. Selene: Pioneering automated proof in software verification. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024. 3

Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *International Conference on Software Engineering (ICSE)*, 2025. 3

Minghai Lu, Benjamin Delaware, and Tianyi Zhang. Proof automation with large language models. In *International Conference on Automated Software Engineering (ASE)*, 2024. 3

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2010. 2

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 2023. 2

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. 2

Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. *Handbook of model checking*, 2018. 2

Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2023. 2

Google DeepMind. AI achieves silver-medal standard solving international mathematical olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024. 2

Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. Lean-smt: An smt tactic for discharging proof goals in lean. *arXiv preprint arXiv:2505.15796*, 2025. 6

Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution. *arXiv preprint arXiv:2001.04301*, 2020. 6

Lean Prover Community. Plausible: A property testing framework for Lean 4 that integrates into the tactic framework. https://github.com/leanprover-community/plausible, 2024. 6

Qi Liu, Xinhao Zheng, Xudong Lu, Qinxiang Cao, and Junchi Yan. Rethinking and improving autoformalization: towards a faithful metric and a dependency retrieval-based approach. In *International Conference on Learning Representations (ICLR)*, 2025. 7

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating systems principles (SOSP)*, 2009. 8

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2021. 11

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations (ICLR)*, 2024. 11

A. Datasets and Detailed Experimental Setup

A.1. License

We ensure compliance with all relevant licenses: MBPP-DFY-50 (Misu et al., 2024) is licensed under GPL-3.0, while both CloverBench (Sun et al., 2024) and LiveCodeBench (Jain et al., 2025) use MIT licenses. Our datasets VERINA is licensed under GPL-3.0. Consistent with established research practices (Hendrycks et al., 2021; Jain et al., 2025), we only use publicly available materials from competitive programming platforms such as LeetCode. Our collection and use of these problems is strictly for academic research purposes, and VERINA involves no model training or fine-tuning processes.

A.2. Model Configurations and Compute

Table 3 presents the configuration details and total experiment costs for all nine evaluated LLMs. For all LLMs, we use a temperature of 1.0 and a maximum output token budget of 10,000. For reasoning models, we use default settings of reasoning efforts or budgets. We host DeepSeek Prover 2 7B and Qwen 3 235B-A22B locally using 8 NVIDIA H100 80GB GPUs. We run other LLMs through APIs, for which we provide the total cost and cost per million tokens. The costs marked with asterisks include the additional expenses incurred during iterative proof refinement experiments, which required up to 64 refinement attempts per datapoint.

A.3. Prompts

We employ a consistent 2-shot prompting approach across all models and tasks to enhance output format adherence and task understanding. The 2-shot examples are excluded from the final benchmark evaluation. For each problem instance, we sample 5 responses from each model and calculate pass@1 metrics (Chen et al., 2021) using these 5 samples to ensure robust evaluation statistics. We utilize DSPy (Khattab et al., 2024) for structural prompting. We provide the detailed prompts in the following: Prompt 1 for CodeGen, Prompt 2 for SpecGen, Prompt 3 for ProofGen, and Prompt 4 for ProofGen with iterative refinement.

Table 3: Detailed configurations and costs for evaluated LLMs.

Vendor	Model Name	Checkpoint	Type	Price ($/1M tokens) (Input / Output)	Cost
OpenAI	GPT 4o-mini	gpt-4o-mini-2024-07-18	API	$0.15 / $0.60	$10.94
	GPT 4o	gpt-4o-2024-08-06	API	$2.50 / $10.0	$153.01
	GPT 4.1	gpt-4.1-2025-04-14	API	$2.00 / $8.00	$453.72*
	o4 mini	o4-mini-2025-04-16	API	$1.10 / $4.40	$894.38*
Anthropic	Claude Sonnet 3.7	claude-3-7-sonnet-20250219	API	$3.00 / $15.0	$777.60*
Google	Gemini 2.5 Flash	gemini-2.5-flash-preview-04-17	API	$0.15 / $0.60	$295.20*
DeepSeek	DeepSeek V3	DeepSeek-V3-0324	API	$1.25 / $1.25	$51.15
	DeepSeek Prover 2 7B	DeepSeek-Prover-V2-7B	GPU	-	-
Qwen	Qwen 3 235B-A22B	Qwen3-235B-A22B-FP8	GPU	-	-

* Including costs for iterative proof refinement experiments.

Prompt 1 (CodeGen)

Instructions

```
You are an expert in Lean 4 programming and theorem proving.
Please generate a Lean 4 program that finishes the task described in
‘task_description‘ using the template provided in ‘task_template‘.
The ‘task_template‘ is a Lean 4 code snippet that contains placeholders
(warpped with {{}}) for the code to be generated.
The program should:
- Be well-documented with comments if necessary
- Follow Lean 4 best practices and use appropriate Lean 4 syntax and features
- DO NOT use Lean 3 syntax or features
- DO NOT import Std or Init
Hint:
- Use a[i]! instead of a[i] when a is an array or a list when necessary
```

Input Fields

- **task_description**
  `Description of the Lean 4 programming task to be solved.`
- **task_template**
  `Lean 4 template with placeholders for code generation and optional reference specification.`

Output Fields

- **imports**
  `Imports needed for ‘code‘. Keep it empty if not needed.`
- **code_aux**
  `Auxiliary definitions for ‘code‘. Keep it empty if not needed.`
- **code**
  `Generated Lean 4 code following the template signature and complete the task.`

Prompt 2 (SpecGen)

Instructions

```
You are an expert in Lean 4 programming and theorem proving.
Please generate a Lean 4 specification that constrains the program
implementation using the template provided in `task_template`.
The `task_template` is a Lean 4 code snippet that contains placeholders
(warpped with {{}}) for the spec to be generated.
The precondition should be as permissive as possible, and the postcondition
should model a sound an complete relationship between input and output of the
program based on the `task_description`.
The generated specification should:
- Be well-documented with comments if necessary
- Follow Lean 4 best practices and use appropriate Lean 4 syntax and features
- DO NOT use Lean 3 syntax or features
- DO NOT import Std or Init
- Only use `precond_aux` or `postcond_aux` when you cannot express
the precondition or postcondition in the main body of the specification
- add @[reducible, simp] attribute to the definitions in `precond_aux` or
`postcond_aux`
Hint:
- Use a[i]! instead of a[i] when a is an array or a list when necessary
```

Input Fields

- **task_description**
  `Description of the Lean 4 programming task to be solved.`
- **task_template**
  `Lean 4 template with placeholders for specfication generation and optional reference code.`

Output Fields

- **imports**
  `Imports needed for 'precond' and 'postcond'. Keep it empty if not needed.`
- **precond_aux**
  `Auxiliary definitions for 'precond'. Keep it empty if not needed.`
- **precond**
  `Generated Lean 4 code specifying the precondition.`
- **postcond_aux**
  `Auxiliary definitions for 'postcond'. Keep it empty if not needed.`
- **postcond**
  `Generated Lean 4 code specifying the postcondition.`

Prompt 3 (ProofGen)

Instructions

```
You are an expert in Lean 4 programming and theorem proving.
Please generate a Lean 4 proof that the program satisfies the specification
using the template provided in `task_template`.
The `task_template` is a Lean 4 code snippet that contains placeholders
(warpped with {{}}) for the proof to be generated.
The proof should:
- Be well-documented with comments if necessary
- Follow Lean 4 best practices and use appropriate Lean 4 syntax and features
- DO NOT use Lean 3 syntax or features
- DO NOT import Std or Init
- DO NOT use cheat codes like `sorry`
Hint:
- Unfold the implementation and specification definitions when necessary
- Unfold the precondition definitions at h_precond when necessary
```

Input Fields

- **task_description**
  `Description of the Lean 4 programming task to be solved.`
- **task_template**
  `Lean 4 template with code and specification to be proved, and placeholders for proof generation.`

Output Fields

- **imports**
  `Imports needed for 'proof'. Keep it empty if not needed.`
- **proof_aux**
  `Auxiliary definitions and lemma for 'proof'. Keep it empty if not needed.`
- **proof**
  `Generated Lean 4 proof that the program satisfies the specification.`

Prompt 4 (ProofGen with Iterative Refinement)

Instructions

```
You are an expert in Lean 4 programming and theorem proving.
Please generate a Lean 4 proof that the program satisfies the specification
using the template provided in `task_template`.
The `task_template` is a Lean 4 code snippet that contains placeholders
(warpped with {{}}) for the proof to be generated.
The proof should:
- Be well-documented with comments if necessary
- Follow Lean 4 best practices and use appropriate Lean 4 syntax and features
- DO NOT use Lean 3 syntax or features
- DO NOT import Std or Init
- DO NOT use cheat codes like `sorry`
Hint:
- Unfold the implementation and specification definitions when necessary
- Unfold the precondition definitions at h_precond when necessary

Furthermore, `prev_error` is the error message from the previous proving
attempt.
Please use the `prev_imports`, `prev_proof_aux`, and `prev_proof` as
references to improve the generated proof.
- You can ignore unused variable warnings in the error message.
```

Input Fields

- **task_description**
  Description of the Lean 4 programming task to be solved.
- **task_template**
  Lean 4 template with code and specification to be proved, and placeholders for proof generation.
- **prev_imports**
  Previously generated imports for reference.
- **prev_proof_aux**
  Previously generated proof auxiliary for reference.
- **prev_proof**
  Previously generated proof for reference.
- **prev_error**
  Error message from the previous proving attempt.

Output Fields

- **imports**
  Imports needed for `proof`. Keep it empty if not needed.
- **proof_aux**
  Auxiliary definitions and lemma for `proof`. Keep it empty if not needed.
- **proof**
  Generated Lean 4 proof that the program satisfies the specification.

A.4. Implementation of Evaluation Metrics in Lean

In Section 4.1, we provide a high-level description of our evaluation metrics for the three foundational tasks of verifiable code generation. Now we describe how we implement these metrics in Lean 4.

Proof evaluation. We directly evaluate generated proofs using the Lean compiler and filter out any proofs containing placeholders, as described in Section 4.1.

Code evaluation. We evaluate generated code on unit tests using `#guard` statements in Lean 4, ensuring the implementation produces correct outputs for given inputs. The evaluation harness for generated codes is illustrated in **??**.

Specification evaluation. Recall in Section 4.1, we define the soundness and completeness of model-generated pre-condition $\hat{P}$ and post-condition $\hat{Q}$ in relation to their ground truth counterparts P and Q: (i) $\hat{P}$ is sound iff $\forall \overline{x}.P(\overline{x}) \Rightarrow \hat{P}(\overline{x})$; (ii) $\hat{P}$ is complete iff $\forall \overline{x}.\hat{P}(\overline{x}) \Rightarrow P(\overline{x})$; (iii) $\hat{Q}$ is sound iff $\forall \overline{x}, y.P(\overline{x}) \wedge \hat{Q}(\overline{x}, y) \Rightarrow Q(\overline{x}, y)$; (iv) $\hat{Q}$ is complete iff $\forall \overline{x}, y.P(\overline{x}) \wedge Q(\overline{x}, y) \Rightarrow \hat{Q}(\overline{x}, y)$. Moreover, since our evaluator is based on testing, we only require that $\overline{x}$ and y are from our test suite. Our quality assurance process in Section 3.2 ensures that all ground truth pre-conditions and post-conditions pass our positive tests and do not pass our negative tests. Therefore, we can simplify the soundness and completeness metrics as follows:

- Deciding the soundness of $\hat{P}$ is equivalent to verifying whether $\hat{P}(\overline{x})$ holds for all positive tests $\overline{x}$ in our test suite. This is because for all negative tests $\overline{x}$, $P(\overline{x})$ does not hold, making $P(\overline{x}) \Rightarrow \hat{P}(\overline{x})$ true by default. For all positive tests $\overline{x}$, $P(\overline{x})$ holds, and $P(\overline{x}) \Rightarrow \hat{P}(\overline{x})$ is true iff $\hat{P}(\overline{x})$ is true.
- Similarly, deciding the completeness of $\hat{P}$ is equivalent to verifying whether $\hat{P}(\overline{x})$ does not hold for all negative tests $\overline{x}$ in our test suite.
- The soundness of $\hat{Q}$ can be evaluated using our negative test cases.
- The completeness of $\hat{Q}$ can be evaluated using our positive test cases.

For each test case evaluation, we employ the two-step approach described in Section 4.1. First, we check if the relationship (with the specific test case incorporated) is directly decidable in Lean 4 on the test case via `decide`. If not, we proceed to property-based testing using `plausible` tactic. The evaluation implementation in Lean 4 is illustrated in **????**.

B. Additional Experimental Evaluation Results

Achieving simultaneous soundness and completeness poses great challenge, particularly for post-conditions. As shown in Figure 12, the substantial performance gap between preconditions and postconditions confirms that generating complex input-output relationships remains significantly more challenging than input validation constraints. Furthermore, the drop in performance when requiring both soundness and completeness simultaneously—compared to achieving either individually—demonstrates that partial correctness is insufficient and justifies our comprehensive evaluation framework for specification quality.

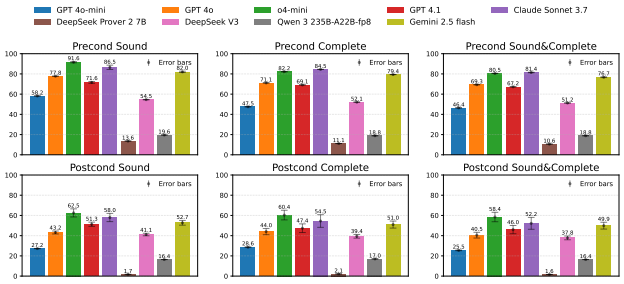

Figure 12: Detailed performance of LLMs on VERINA's SpecGen task.

Detailed performance breakdown. Tables 4 to 6 provide detailed breakdowns of model performance across the three foundational tasks. They reveal that syntax incorrectness and use of non-existent library functions (as demonstrated in Appendix C) represent the major problems, especially for less capable models. Specifically, after manual inspection of the evaluation result, DeepSeek Prover 2 7B and Qwen 3 235B-A22B-FP8 suffer from instruction following ability, failing to output the desired format specified in our prompts (cf. Appendix A.3). The relatively low unknown percentages across most evaluations demonstrate that our specification evaluation metric is reliable. Pre-conditions are generally simpler than post-conditions, resulting in lower unknown rates during evaluation. More capable models often generate specifications with more complicated logical structures, leading to higher unknown percentages in post-condition evaluation. We present a case study in Appendix C on the challenge of automatically evaluating LLM-generated specifications. In our main results, we report the uncertainty from unknown cases using error bars, where the lower bound represents the Pass% in the table and the upper bound represents Pass%+Unknown% in the table.

Table 4: Detailed performance of CodeGen.

Model	Cannot Compile%	Fail Unit Test%	Pass%
GPT 4o-mini	70.1	1.4	28.6
GPT 4o	51.6	2.8	45.7
GPT 4.1	40.5	3.1	56.4
o4-mini	34.1	4.5	61.4
Claude Sonnet 3.7	54.1	1.7	44.2
Gemini 2.5 Flash	62.9	0.6	36.5
DeepSeek Prover 2 7B	96.8	0.8	2.5
DeepSeek V3	62.3	1.7	36.0
Qwen 3 235B-A22B-fp8	80.0	0.0	20.0

Table 5: Detailed performance of SpecGen for pre-condition.

Model	Cannot Compile%	Soundness			Completeness		
		Pass%	Fail%	Unknown%	Pass%	Fail%	Unknown%
GPT 4o-mini	40.8	58.2	1.1	0.0	47.5	11.8	0.0
GPT 4o	19.8	77.7	1.8	0.8	71.1	8.7	0.4
GPT 4.1	24.3	70.7	1.1	4.0	69.1	3.5	3.1
o4-mini	5.4	91.0	0.6	3.0	82.1	10.7	1.8
Claude Sonnet 3.7	4.9	84.4	2.3	8.5	84.5	3.7	6.8
Gemini 2.5 Flash	14.7	81.4	1.5	2.5	79.4	5.0	1.0
DeepSeek Prover 2 7B	85.9	13.6	0.5	0.0	11.1	3.0	0.0
DeepSeek V3	43.7	54.3	0.8	1.2	52.1	3.1	1.1
Qwen 3 235B-A22B-fp8	80.4	19.6	0.0	0.0	18.8	0.8	0.0

Table 6: Detailed performance of SpecGen for post-condition.

Model	Cannot Compile%	Soundness			Completeness		
		Pass%	Fail%	Unknown%	Pass%	Fail%	Unknown%
GPT 4o-mini	68.3	27.1	4.2	0.4	28.2	2.6	0.9
GPT 4o	49.1	41.7	4.6	4.6	41.0	1.8	8.1
GPT 4.1	41.8	49.2	1.8	7.2	43.1	0.8	14.3
o4-mini	22.7	58.5	3.1	15.7	55.6	2.7	19.0
Claude Sonnet 3.7	30.6	53.9	3.2	12.3	48.2	1.6	19.6
Gemini 2.5 Flash	40.6	50.4	1.5	7.5	47.5	1.0	10.9
DeepSeek Prover 2 7B	97.2	1.7	1.0	0.1	2.1	0.5	0.1
DeepSeek V3	53.9	39.9	2.6	3.6	37.5	3.6	4.9
Qwen 3 235B-A22B-fp8	83.0	16.4	0.6	0.0	17.0	0.0	0.0

C. Case Studies of Model Failures and Evaluation Metrics

In this appendix section, we provide a detailed qualitative analysis of common model failure patterns across the three foundational tasks and illustrate how LLMs struggle with different aspects of verifiable code generation through concrete examples. We also discuss how our evaluation metrics flag these failures, highlighting both their effectiveness and limitations.

Code generation failure: hallucinated method usage. Figure 13 demonstrates a common LLM failure mode where o4-mini generates code that appears syntactically correct but contains non-existent methods. While the model correctly identifies the XOR-based algorithmic approach and provides accurate comments, it hallucinates the `Int.xor` method that does not exist in Lean 4's standard library. This shows that current LLMs fall short in understanding Lean 4's language features.

Code generation failures: unit test rejections. Figure 14 illustrates how subtle logical errors in LLM-generated code can lead to unit test failures. The task requires implementing a function that finds the next greater element for each number in `nums1` within the array `nums2`, or outputs `-1` if there is none. o4-mini generates a `nextGreaterOne` helper function with a bug in the state management logic. After finding the target element, the function incorrectly calls `aux tl false` instead of `aux tl true` in Line 22, causing it to lose track of having found the target and fail to identify subsequent greater elements. This results in incorrect outputs for the test case where `nums1 = [1, 2, 3]` and `nums2 = [3, 2, 1, 4]` should return `[4, 4, 4]`.

Specification generation failures: unsound pre-conditions. Figure 15 demonstrates how LLMs can generate specifications that are too restrictive, leading to unsound pre-conditions. The task description states "Assuming k $\leq$ number of distinct elements in nums". The ground truth pre-condition correctly uses `k ≤ nums.eraseDups.length` to allow k to equal the number of distinct elements. However, the LLM-generated version uses strict inequality `k < (distinct nums).length`, which incorrectly excludes valid cases where k equals the total number of distinct elements. This makes the pre-condition unsound as it rejects legitimate inputs that should be accepted by the specification. In our test suites, we have a positive test case with `nums = [5]` and `k = 1`. Since the LLM-generated pre-condition rejects this test case, our evaluation metric determines that it is unsound.

Specification generation failures: incomplete pre-conditions. Figure 16 demonstrates how LLMs can generate overly permissive preconditions that fail to capture essential constraints. The task description specifies that "All integers in both arrays are unique" and that "nums1: A list of integers, which is a subset of nums2". The ground truth precondition correctly enforces three critical requirements: `List.Nodup nums1` ensures uniqueness in the first array, `List.Nodup nums2` ensures uniqueness in the second array, and `nums1.all (fun x => x ∈ nums2)` verifies that `nums1` is indeed a subset of `nums2`. However, the LLM-generated precondition simply uses `True`, completely ignoring all stated constraints. This makes the precondition incomplete as it accepts invalid inputs that violate the problem's fundamental assumptions, potentially leading to incorrect behavior in the implementation and proof generation phases. In our test suites, we have a negative test case with `nums1 = [1, 1]` and `nums2 = [1, 2]`. Since the LLM-generated pre-condition accepts this negative test case, our evaluation metric determines that the LLM-generated pre-condition is incomplete.

Specification generation failures: unsound post-conditions. Figure 17 illustrates how LLMs can generate post-conditions that miss critical constraints, leading to unsound specifications. The task involves adding two numbers represented as digit lists in reverse order. The ground truth post-condition correctly enforces three essential properties: arithmetic correctness, digit validity (each digit < 10), and prohibition of leading zeros except for the special case where the result is zero. However, the LLM-generated post-condition omits the leading zero constraint entirely, only checking that the result is non-empty and digits are valid. This unsound specification would accept invalid outputs like `[2, 1, 0]` representing `012` (which is one of the negative test case), which should be rejected in favor of `[2, 1]`. The missing constraint demonstrates how LLMs may capture the primary functional requirements while overlooking more subtle constraints that are crucial for correctness.

Specification generation failures: unsound and incomplete post-conditions. Figure 18 demonstrates how LLMs can generate post-conditions that are both unsound and incomplete by failing to handle edge cases properly. The task involves finding the smallest single-digit prime factor of a natural number. The ground truth post-condition correctly handles all cases including the edge case where `n = 0`, specifying that the result should be 0 when the input is 0 or when no single-digit prime divides `n`. However, the LLM-generated post-condition fails to consider `n = 0` entirely. When `n = 0`, the condition `n % p ≠ 0` is false for any prime p (since `0 % p = 0`), making the first disjunct impossible to satisfy. This renders the specification both unsound (accepts incorrect outputs) and incomplete (rejects valid cases where `n = 0`). The missing edge case handling demonstrates how LLMs may overlook corner cases that are crucial for specification completeness (and soundness). We have a positive test case where `n = 0` and `result = 0` and a corresponding negative test case where

```
def FindSingleNumber (nums : List Int) (h_precond : FindSingleNumber_precond (nums)) : Int :=
  -- !benchmark @start code
  -- XOR all elements: duplicates cancel out, leaving the unique number
  nums.foldl (fun acc x => Int.xor acc x) 0
  -- !benchmark @end code
```

Figure 13: Example (verina_advanced_1): o4-mini generates code with hallucinated `Int.xor` method that does not exist in Lean 4.

```

-- !benchmark @start code_aux
/-
Helper function to find the next greater element of 'target' in 'nums2'.
We traverse the list, first locating 'target', then searching for the first element > 'target'.
If not found, returns -1.
-/
def nextGreaterOne (nums2 : List Int) (target : Int) : Int :=
  let rec aux (l : List Int) (foundTarget : Bool) : Int :=
    match l with
    | []      => -1
    | hd::tl  =>
        if !foundTarget then
          if hd == target then
            aux tl true
          else
            aux tl false
        else
          if hd > target then
            hd
          else
            aux tl false
  aux nums2 false
-- !benchmark @end code_aux

def nextGreaterElement (nums1 : List Int) (nums2 : List Int)
     (h_precond : nextGreaterElement_precond (nums1) (nums2)) : List Int :=
  -- !benchmark @start code
  -- For each element in 'nums1', compute its next greater in 'nums2'
  nums1.map fun x => nextGreaterOne nums2 x
  -- !benchmark @end code
```

Figure 14: Example (verina_advanced_57): o4-mini generates code with logical error in state management, causing unit test failure.

```
@[reducible, simp]
def topKFrequent_precond (nums : List Int) (k : Nat) : Prop :=
  -- !benchmark @start precond
  k ≤ nums.eraseDups.length
  -- !benchmark @end precond
```

(a) Ground truth pre-condition.

```
-- !benchmark @start precond_aux
@[reducible, simp]
def contains (xs : List Int) (x : Int) : Bool :=
    xs.foldl (fun b y => b || y == x) false
@[reducible, simp]
def distinct (xs : List Int) : List Int :=
    xs.foldl (fun acc y => if contains acc y then acc else acc ++ [y]) []
-- !benchmark @end precond_aux
@[reducible, simp]
def topKFrequent_precond (nums : List Int) (k : Nat) : Prop :=
  -- !benchmark @start precond
  k < (distinct nums).length
  -- !benchmark @end precond
```

(b) Unsound pre-condition generated by o4-mini.

Figure 15: Example (verina_advanced_76): o4-mini generates unsound pre-condition using strict inequality instead of allowing k to equal the number of distinct elements.

```
-- Ground truth pre-condition
@[reducible, simp]
def nextGreaterElement_precond (nums1 : List Int) (nums2 : List Int) : Prop :=
  -- !benchmark @start precond
  List.Nodup nums1 ∧
  List.Nodup nums2 ∧
  nums1.all (fun x => x ∈ nums2)
  -- !benchmark @end precond
```

(a) Ground truth pre-condition.

```
@[reducible, simp]
def nextGreaterElement_precond (nums1 : List Int) (nums2 : List Int) : Prop :=
  -- !benchmark @start precond
  True
  -- !benchmark @end precond
```

(b) Incomplete pre-condition generated by o4-mini.

Figure 16: Example (`verina_advanced_57`): o4-mini generates incomplete pre-condition using `True` instead of enforcing uniqueness and subset constraints.

```
-- !benchmark @start postcond_aux
def listToNat : List Nat → Nat
| []       => 0
| d :: ds  => d + 10 * listToNat ds
-- !benchmark @end postcond_aux
@[reducible, simp]
def addTwoNumbers_postcond (l1 : List Nat) (l2 : List Nat) (result: List Nat) (h_precond : addTwoNumbers_precond (l1) (l2)) : Prop
    ↪ :=
  -- !benchmark @start postcond
  listToNat result = listToNat l1 + listToNat l2 ∧
  (∀ d ∈ result, d < 10) ∧
  -- No leading zeros unless the result is zero
  (result.getLast! ≠ 0 ∨ (l1 = [0] ∧ l2 = [0] ∧ result = [0]))
  -- !benchmark @end postcond
```

(a) Ground truth post-condition.

```
-- LLM generated post-condition
-- !benchmark @start postcond_aux
@[reducible, simp]
def listToNatRev : List Nat → Nat
| []       => 0
| d :: ds  => d + 10 * listToNatRev ds
-- !benchmark @end postcond_aux
@[reducible, simp]
def addTwoNumbers_postcond (l1 : List Nat) (l2 : List Nat) (result: List Nat) (h_precond : addTwoNumbers_precond (l1) (l2)) : Prop
    ↪ :=
  -- !benchmark @start postcond
  result ≠ [] ∧
  listToNatRev result = listToNatRev l1 + listToNatRev l2 ∧
  ∀ d, d ∈ result → d < 10
  -- !benchmark @end postcond
```

(b) Unsound post-condition generated by o4-mini.

Figure 17: Example (`verina_advanced_5`): o4-mini generates unsound postcondition that fails to rule out leading zeros in the result.

`n = 0` and `result = 2` that capture this edge case. The LLM-generated post-condition rejects the positive test case and accepts the negative test case, therefore our evaluation metric determines that this generated post-condition is both unsound and incomplete.

Untestable post-conditions. Figure 19 demonstrates the limitations of our testing-based evaluation framework when encountering specifications with quantifiers over complicated structures or infinite domains. The LLM-generated post-condition for finding the length of the longest increasing subsequence contains a universal quantifier `∀ s : List Int` that ranges over all possible integer lists, making it impossible to evaluate even with plausible testing. Our evaluation framework returns unknown for such cases, as neither decidable testing nor plausible exploration can adequately handle the unbounded quantification. This example highlights a fundamental challenge in automatically evaluating LLM-generated formal specifications: while our framework successfully handles most practical cases, very complicated specifications require more comprehensive approaches such as automated theorem provers or LLM-based proof generation, which we leave to future work.

Proof generation success with iterative refinement. Figure 20, 21, 22, and 23 demonstrate o4-mini's iterative proof refinement process over 24 attempts. The task involves proving that a list element removal function satisfies its specification. Initially, o4-mini generates an overly simple proof using direct induction and simp tactics, which fails to handle the complex logical structure. After 23 failed refinement attempts, the model makes useful improvements based on Lean compiler error messages, significantly restructuring its approach. The error message from the 23rd attempt reveals that the proof got stuck on conditional expressions involving decide predicates. In the successful 24th attempt, o4-mini improves its strategy by introducing an auxiliary lemma `helper_eq_filter` and using explicit case analysis with `by_cases`. This demonstrates how iterative refinement with compiler feedback can guide LLMs toward substantially different and correct proof strategies, though it may require many attempts to achieve success.

```
-- Ground truth post-condition
@[reducible, simp]
def singleDigitPrimeFactor_postcond (n : Nat) (result: Nat) (h_precond : singleDigitPrimeFactor_precond (n)) : Prop :=
  -- !benchmark @start postcond
  result ∈ [0, 2, 3, 5, 7] ∧
  (result = 0 → (n = 0 ∨ [2, 3, 5, 7].all (n % · ≠ 0))) ∧
  (result ≠ 0 → n ≠ 0 ∧ n % result == 0 ∧ (List.range result).all (fun x => x ∈ [2, 3, 5, 7] → n % x ≠ 0))
  -- !benchmark @end postcond
```

(a) Ground truth post-condition.

```
-- LLM generated post-condition
-- !benchmark @start postcond_aux
@[reducible, simp]
def isSingleDigitPrime (p : Nat) : Prop :=
    p = 2 ∨ p = 3 ∨ p = 5 ∨ p = 7
-- !benchmark @end postcond_aux
@[reducible, simp]
def singleDigitPrimeFactor_postcond (n : Nat) (result: Nat) (h_precond : singleDigitPrimeFactor_precond (n)) : Prop :=
  -- !benchmark @start postcond
  -- Either no small prime divides n, so we return 0
  (result = 0 ∧ ∀ p, isSingleDigitPrime p → n % p ≠ 0)
  ∨
  -- Or result is the smallest -singledigit prime divisor of n
  (isSingleDigitPrime result ∧ n % result = 0 ∧ ∀ q, isSingleDigitPrime q → n % q = 0 → result ≤ q)
  -- !benchmark @end postcond
```

(b) Unsound and incomplete post-condition generated by o4-mini.

Figure 18: Example (`verina_advanced_72`): o4-mini generates unsound and incomplete post-condition that fails to handle the edge case `n = 0`.

```
-- !benchmark @start postcond_aux
@[reducible, simp]
def IsSubsequence : List Int → List Int → Prop
| [], _       => True
| _ :: _, []  => False
| x :: xs, y :: ys =>
    if x = y then IsSubsequence xs ys
    else IsSubsequence (x :: xs) ys

@[reducible, simp]
def strictlyIncreasing : List Int → Prop
| []          => True
| [_]         => True
| x :: y :: rest => x < y ∧ strictlyIncreasing (y :: rest)
-- !benchmark @end postcond_aux
@[reducible, simp]
def lengthOfLIS_postcond (nums : List Int) (result: Nat) (h_precond : lengthOfLIS_precond (nums)) : Prop :=
  -- !benchmark @start postcond
  (∀ s : List Int, IsSubsequence s nums ∧ strictlyIncreasing s → List.length s ≤ result)
  ∧ ∃ s : List Int, IsSubsequence s nums ∧ strictlyIncreasing s ∧ List.length s = result
  -- !benchmark @end postcond
```

Figure 19: Example (`verina_advanced_25`): o4-mini generates post-condition with quantifiers over lists that cannot be evaluated by plausible testing.

```
@[reducible, simp]
def removeElement_precond (lst : List Nat) (target : Nat) : Prop :=
  -- !benchmark @start precond
  True
  -- !benchmark @end precond

def removeElement (lst : List Nat) (target : Nat) (h_precond : removeElement_precond (lst) (target)) : List Nat :=
  -- !benchmark @start code
  let rec helper (lst : List Nat) (target : Nat) : List Nat :=
    match lst with
    | [] => []
    | x :: xs =>
      let rest := helper xs target
      if x = target then rest else x :: rest
  helper lst target
  -- !benchmark @end code

@[reducible]
def removeElement_postcond (lst : List Nat) (target : Nat) (result: List Nat) (h_precond : removeElement_precond (lst) (target)):
    ↪ Prop :=
  -- !benchmark @start postcond
  result = lst.filter (fun x => x ≠ target)
  -- !benchmark @end postcond


-- !benchmark @start proof_aux
-- !benchmark @end proof_aux

theorem removeElement_spec_satisfied (lst: List Nat) (target: Nat) (h_precond : removeElement_precond (lst) (target)):
    removeElement_postcond (lst) (target) (removeElement (lst) (target) h_precond) h_precond := by
  -- !benchmark @start proof
  -- Unfold the definitions of removeElement and the postcondition
    dsimp [removeElement, removeElement_postcond]
    -- Proceed by induction on the input list
    induction lst with
    | nil =>
    -- Base case: both sides reduce to []
    simp
    | cons x xs ih =>
    -- Inductive step: use the induction hypothesis
    simp [ih]
  -- !benchmark @end proof
```

Figure 20: Example (`verina_advanced_64`): o4-mini generates incorrect proof on the first attempt.

```
-- !benchmark @start proof_aux
-- Auxiliary lemma: the `helper` function produces exactly `lst.filter (fun x => x ≠ target)`
theorem helper_eq_filter (lst : List Nat) (target : Nat) :
    removeElement.helper lst target = lst.filter (fun x => x ≠ target) := by
  induction lst with
  | nil =>
    -- both sides reduce to `[]`
    simp [removeElement.helper]
  | cons x xs ih =>
    -- unfold one step of `helper`
    dsimp [removeElement.helper]
    -- solve the `if` statement using induction hypothesis
    simp [ih]
-- !benchmark @end proof_aux

theorem removeElement_spec_satisfied (lst: List Nat) (target: Nat) (h_precond : removeElement_precond (lst) (target)):
    removeElement_postcond (lst) (target) (removeElement (lst) (target) h_precond) h_precond := by
  -- !benchmark @start proof
  -- unfold the postcondition and the definition of `removeElement`
  dsimp [removeElement_postcond, removeElement]
  -- the precondition is trivial
  clear h_precond
  -- apply the auxiliary lemma to conclude
  exact helper_eq_filter lst target
  -- !benchmark @end proof
```

Figure 21: Example (`verina_advanced_64`): o4-mini generates incorrect proof after 23 refinement attempts.

```
unsolved goals
case cons
target x : Nat
xs : List Nat
ih : removeElement.helper xs target = List.filter (fun x => decide (x ≠ target)) xs
⊢ (if x = target then List.filter (fun x => !decide (x = target)) xs
  else x :: List.filter (fun x => !decide (x = target)) xs) =
  List.filter (fun x => !decide (x = target)) (x :: xs)
```

Figure 22: Example (`verina_advanced_64`): Lean 4 compiler error message from the 23rd refinement attempt.

```
-- !benchmark @start proof_aux
-- Auxiliary lemma: the `helper` function produces exactly `lst.filter (fun x => x ≠ target)`
theorem helper_eq_filter (lst : List Nat) (target : Nat) :
    removeElement.helper lst target = lst.filter (fun x => x ≠ target) := by
  induction lst with
  | nil =>
    -- both sides reduce to `[]`
    simp [removeElement.helper]
  | cons x xs ih =>
    -- unfold one step of `helper`
    dsimp [removeElement.helper]
    -- case-split on whether `x = target`
    by_cases h : x = target
    case pos =>
      -- if `x = target`, `helper` drops it, and `filter` also drops it
      simp [h, ih]
    case neg =>
      -- if `x ≠ target`, `helper` keeps it, and `filter` also keeps it
      simp [h, ih]
-- !benchmark @end proof_aux

theorem removeElement_spec_satisfied (lst: List Nat) (target: Nat) (h_precond : removeElement_precond (lst) (target)):
    removeElement_postcond (lst) (target) (removeElement (lst) (target) h_precond) h_precond := by
  -- !benchmark @start proof
  -- unfold the postcondition and the definition of `removeElement`
  dsimp [removeElement_postcond, removeElement]
  -- the precondition is trivial
  clear h_precond
  -- apply the auxiliary lemma to conclude
  exact helper_eq_filter lst target
  -- !benchmark @end proof
```

Figure 23: Example (`verina_advanced_64`): o4-mini generates correct proof on the 24th attempt.