EFFICIENT FORMALLY VERIFIED CODE GENERATION VIA SPECIFICATION QUALITY ESTIMATES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have become a ubiquitous element of software development workflows, but the code they generate may diverge from user intent or contain bugs. Formal verification has shown promise for addressing the correctness gap: instead of simply generating executable code, an LLM-based agent is tasked with producing a specification, an implementation, and a formal language proof of correctness. Generating the artifacts required for formal verification, though, is difficult: synthesized specifications are often incorrect, and generating implementations and proofs is computationally expensive. In this paper, we ask whether variation in specifications can be harnessed to improve agents' performance on verified code generation tasks. We first find that the downstream cost of implementation and proof generation can be accurately predicted given a specification as input. We then show that these cost estimates are useful: they can guide a verified code generation agent in selecting a specification that reduces the downstream cost of implementation and proof writing without diverging from user intent. We realize our approach as a tool, SPEQUEST, and find that it makes implementation and proof generation more than $3\times$ faster on a standard benchmark set.

1 INTRODUCTION

LLM-generated code is a component of almost all modern software development processes: agentic systems can resolve real-world GitHub issues (Jimenez et al., 2024; Yang et al., 2024), generate large software artifacts (Zhao et al., 2025), and identify and fix security vulnerabilities (Lee et al., 2025). Despite significant progress on agentic code generation, however, the correctness of generated code is still overwhelmingly measured with test cases, which have well-known limits: they are incomplete (Dijkstra, 1972) and agents can "reward hack", overfitting a solution to test cases rather than producing a general implementation (Aleithan et al., 2024; Ahmed et al., 2026; Zhong et al., 2026).

Recently, formal verification has been proposed as a solution to the ills of test-based correctness checking. Verified code generation formally proves the correctness of agent-written code in a proof language like Lean (Ye et al., 2025; Thakur et al., 2025) or using an SMT solver (Loughridge et al., 2025; Gladshtein et al., 2026). Starting from a natural language description of user intent, a verified code generation agent produces three artifacts: a specification that states the user's requirements in formal language, an implementation written in an executable programming language that realizes the user intent, and a proof or certificate that the implementation matches the specification. Generating implementations and writing proofs have received significant attention in existing work, including producing implementations in formal languages (Brandfonbrener et al., 2024; Misu et al., 2024; Aggarwal et al., 2025), and generating proofs for both programs (Yang et al., 2025; Chen et al., 2025) and mathematical tasks (First et al., 2023; Yang et al., 2023; Xin et al., 2024).

Generating a formal specification given a natural language description, on the other hand, remains a challenge. Producing correct specifications is difficult because natural language descriptions of user intent are inherently imprecise and ambiguous. As a result, even expert-written natural language "specifications" often map to more than one possible *formal* specification (Stengel-Eskin et al., 2024; Mu et al., 2024). Even when ambiguity is not at issue, writing a formal specification is an error-prone process: prior work has found that agents often produce specifications that are plausible but subtly wrong on corner cases (Endres et al., 2024; Lahiri, 2024; Ma et al., 2025). Human-written

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

specifications are not immune: formally verified systems have shipped with bugs caused by gaps between their specifications and the environments they run in (Fonseca et al., 2017), and the reference specifications of existing verified code generation benchmarks often fail to capture the full natural language intent (Thakur et al., 2025). The task is only made harder by the fact that a particular formal specification can be expressed in infinitely many syntactically different but mathematically equivalent forms; whether or not two such specifications are equivalent is undecidable (Rice, 1953).

Since the specification lies at the head of the verified code generation pipeline, variation in specifications has significant downstream impacts. In our experiments, for example, we found that varying the syntactic form of a specification can lead to as much as a $5\times$ difference in code generation and proof synthesis performance on the Verina benchmark set (Ye et al., 2025), with generation times varying between 47.3 s and 264.5 s. Since practical code generation agents operate with fixed timeouts, this means that a poor choice of specification can doom implementation generation and proof writing: on the AlgoVeri benchmark (Zhao et al., 2026), for example, variation in specifications can account for a 17-percentage-point difference in the number of proved instances.

In this paper, we propose to harness this variation in specification quality to improve agents' performance on the end-to-end verified code generation task. Our approach is to compute specification quality estimates *before an implementation or proof is written*, allowing a verified code generator to iteratively improve the specification and thereby boost performance on the downstream code generation and proof writing tasks. We formally model specification quality as a random cost function for implementation and proof generation. We implement novel estimators for specification quality using both LLM-based agents and conventional prediction models, and describe three methods of harnessing these estimates to improve performance on verified code generation tasks.

We implement our approach as a tool called SPEQUEST (**SPE**cification **QU**ality **EST**imator) and perform an extensive evaluation. We first find that specification quality can be accurately predicted before any implementation or proof is written, including by conventional regression models with far lower computational cost than LLM-based agents. Second, we show that quality estimates are useful on multiple standard benchmark sets: on Verina, for example, our approach matches the accuracy of the reference specifications but makes implementation generation and proof writing more than $3\times$ faster.

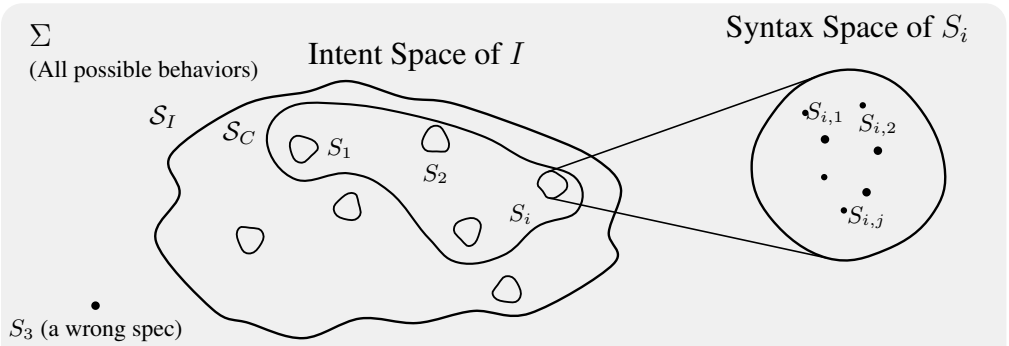
In sum, our work makes the following contributions:

- We introduce a novel method for improving performance on verified code generation by predicting and improving the quality of formal specifications.
- We develop a metric for specification quality and show that it can be accurately predicted.
- We find that specification quality estimates can significantly improve the performance of verified code generation agents.

2 Related Work

Verified Code Generation A growing family of benchmarks measures whether LLMs can produce specifications, implementations, and proofs. Verina (Ye et al., 2025), CLEVER (Thakur et al., 2025), and miniCodeProps (Lohn & Welleck, 2024) target Lean; DafnyBench (Loughridge et al., 2025) and Vericoding (Bursuc et al., 2025) target Dafny; and Velvet (Gladshtein et al., 2026) uses a solver-based verifier working on annotations expressed in a Lean DSL. Prior literature has built on these benchmarks by proposing new methods of verified code generation for particular applications. Misu et al. (2024) prompt LLMs to synthesize verified Dafny methods, while VerMCTS (Brandfonbrener et al., 2024) uses a Monte Carlo tree search guided by verifier verdicts on partial programs. While our work also addresses verified code generation, we focus on harnessing variations in the specification to improve performance, rather than introducing a new verification methodology.

Proof Search and Cost Our work is particularly related to efforts to improve proof search for code verification. Most work on agent-based proof search targets mathematics rather than code, although some search techniques may be applicable to both (Yang et al., 2023; Xin et al., 2024). In the code domain, Baldur (First et al., 2023) generates whole proofs of correctness in one shot and then provides a proof checker's error messages as feedback during repair. For Verus, Yang et al. (2025)



Intent:	I = "Create function `f` that always returns an even number"
Intent Space:	$S_1 : \forall$`i, f(i) % 2 == 0` vs. $S_2 : \forall$`i, f(i) == 4`
Syntax Space:	$S_{1,1} : \forall$`i, f(i) % 2 == 0` vs. $S_{1,2} : \forall$`i,` $\exists$`j, f(i) == 2*j`

Figure 1: The space of possible behaviors Σ. Many candidate specifications $(S_1, S_2, \ldots, S_i)$ are consistent with the natural language intent, while others (e.g. S_3) are wrong. Each specification S_i additionally admits many syntactic realizations $S_{i,1}, S_{i,2}, \ldots, S_{i,j}$ that differ in proof difficulty.

propose a multi-agent pipeline of generation, refinement, and repair agents writing proofs, while SAFE (Chen et al., 2025) and VeruSyn (Di et al., 2026) propose fine-tuning using synthetic example proofs. For Rocq (formerly Coq), Rango (Thompson et al., 2025) maintains a database of similar proofs for retrieval-augmented generation of proofs. Some existing work has focused on lowering the cost of individual proof steps. LeanHammer (Zhu et al., 2025) integrates a library of lemmas with automated provers to construct proofs for subgoals of a Lean theorem. Hilbert (Varambally et al., 2025) recursively decomposes a theorem into subgoals using an informal sketch. Goedel-Code-Prover (Li et al., 2026) applies this decomposition strategy to code verification, ranking candidate decompositions with a learned score, which has some parallels to our specification quality estimates. However, while all of these methods focus on improving proof search, they are internal to the construction of a proof. Specification quality estimates, on the other hand, target the statement to be proved—the specification—and enable us to structure it in a manner that reduces the cost of proof search. Once a specification has been selected, SPEQUEST is orthogonal to existing proof search methods.

Specification Quality Our work is related to prior literature on measuring and improving specification quality. Endres et al. (2024) generate postconditions from natural language and measure their quality with test cases. Lahiri (2024) extend the approach to verification-aware languages, where quality is evaluated by testing specifications symbolically with the verifier rather than executing tests. SpecGen (Ma et al., 2025) also judges candidate specifications by comparing them to an existing implementation using a verifier, while SpecSyn (Ma et al., 2026) scores specifications on their ability to discriminate the original program from mutants. Richter & Wehrheim (2025) combine verifier results with bug discrimination (whether a specification rejects buggy implementations while accepting correct ones) to evaluate specification quality. Our approach, by contrast, estimates specification quality based on the inference cost of program and proof generation, and it predicts quality before any code or proofs are generated. Perhaps the most conceptually similar work to ours is iRank (Chakraborty et al., 2023), which ranks LLM-generated loop invariants to cut wasted verifier calls. Our work differs from iRank in that we estimate the quality of a specification, not an intermediate invariant, and our estimates account for code generation and proof writing, rather than verifier running time.

3 METHODOLOGY

This section formally describes the two sources of variation in specifications, states the specification cost estimation task, and then describes how we realize estimators for specification cost.

3.1 THEORETICAL MODEL

Definitions Let $\mathcal{V}$ be a space of values, and let Σ be the alphabet of possible program behaviors over $\mathcal{V}$, *i.e.*, $\Sigma = \bigcup_{n,m\in\mathbb{N}} \mathcal{V}^n \times \mathcal{V}^m$ is the set of all input-output pairs. For example, $\mathcal{V}$ may be the set of all Python values. A *formal specification* $S \subset \Sigma$ is simply a subset of the possible behaviors, typically expressed with pre- and post-conditions in first-order logic. In this work, we consider only computable specifications (Hayes & Jones, 1989; Fuchs, 1992).

A verified code generator takes as input a natural language statement of intent, I. We model I as a partition of Σ between correct and incorrect specifications; that is, I specifies a subset $\mathcal{S}_I \subset \Sigma$ of possible program behaviors. A specification S is *correct* if $S \subseteq \mathcal{S}_I$ and *incorrect* otherwise (*e.g.*, S_3 in Figure 1). Since natural language expressions of intent are inherently imprecise, we almost always have multiple correct specifications. The exact boundaries of $\mathcal{S}_I$ for a given intent are difficult to compute in practice, and depend on whether we interpret I more or less permissively. In this work, we treat the theoretical object $\mathcal{S}_I$ as a maximalist interpretation of I, and assume only that we can sample some finite number of specifications $S_1, S_2, \ldots, S_i$ from $\mathcal{S}_I$.

Specification Variation There are two types of variation among specifications. The imprecision of I is the first type: each specification S_i represents a different resolution of the ambiguities of I, represented by $S_1, S_2, \ldots$ in Figure 1. We say that this variation is within the *intent space* of I, as in Figure 1. The "Intent Space" line of Figure 1 gives an example of variation within the intent space of the example intent: both logical statements constrain f to produce an even number, but the two are not equivalent.

The second type of variation in specifications arises from the notation we use to express them. A particular specification S_i corresponding to a set of input-output behaviors can be expressed in multiple equivalent ways using first-order logic; we call this set of possible syntactic encodings the *syntax space* of S_i ($L(S_i)$), as in Figure 1.[1] The syntax space of a particular specification is infinite, so we again assume that it can be sampled to produce a finite set of possible specification expressions $S_{i,1}, S_{i,2}, \ldots$. For ease of notation, we define the set $\Sigma_L = \bigcup_{S\subseteq\Sigma} L(S)$.

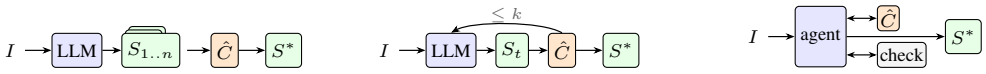
Specification Cost During verified code generation, the generator is given an intent I and tasked with selecting a specification expression $S_{i,j}$, generating a program P, and then proving that P realizes $S_{i,j}$. Both synthesizing P and generating the proof are computationally costly and may fail, but the nondeterminism of LLMs means that the cost is not constant. We therefore model the cost of producing an implementation and a proof with a random function over specification expressions $C : \Omega \times \Sigma_L \to \mathbb{R}^+$, where Ω is the sample space of the generation process's randomness. We can sample C's distribution by performing program and proof generation multiple times on the same specification expression $S_{i,j}$. To ensure that a model does not achieve better results by making unrealistic interpretations of natural language intent, we also transform I by augmenting the natural language intent with a set of correctness criteria $\mathcal{S}_C \subset \mathcal{S}_I$; in practice, these are test cases against which a selected specification is compared. $\mathcal{S}_C$ is generally more constrained than $\mathcal{S}_I$, but still permits some variation within the intent space.

The task addressed by specification quality estimates is then the following:

Task 1 *Given I and $\mathcal{S}_C$, find $S_{i,j}$ with $S_i \subseteq \mathcal{S}_C$ such that $\mathbb{E}[C(S_{i,j})]$ is minimized.*

In practice, the function C may measure the time taken to produce the implementation and proof, the number of input or output tokens used by an agent's LLM calls, or some aggregate metric

[1] If we take specification synthesis as a process of generating specifications from natural language, variation within the intent space corresponds to epistemic uncertainty about I, while syntax space variation corresponds to aleatoric uncertainty (Hüllermeier & Waegeman, 2021; Der Kiureghian & Ditlevsen, 2009).



(a) Selection (b) Feedback loop (c) Free-form agent

Figure 2: Three modes of harnessing specification quality estimates $\hat{C}$; S^* is the specification passed to implementation and proof generation.

that accounts for input, output, and model caching; any such measurement is compatible with our theoretical model.

Estimators We study four classes of estimators for Task 1. First are LLM-based agent estimators: here, we provide the natural language intent description I along with the text of a specification written in formal language as input. Each agent is then tasked with choosing or constructing a specification as described in Section 3.2. The second class of estimators is conventional feature models that operate over a set of structural specification features (for example, the size of the postcondition, the number of quantifiers and logical connectives, and the nesting depth of the specification). Third, we address Task 1 with text models, which treat the Lean specification statement as text, and finally, we use a graph model, a Weisfeiler-Lehman kernel that treats the input Lean text as a chain graph of tokens and summarizes each specification by counts of its local token neighborhoods. Section 4 lists the exact models used and the results obtained for each in detail.

3.2 Harnessing Quality Estimates

We realize Task 1 by constructing estimators $\hat{C}$ for the specification cost function C (the specific estimators we study are described in Section 4.1). Given a particular specification quality estimator $\hat{C}$, we apply it to improve specification quality in one of three ways, as summarized in Figure 2.

Selection (Figure 2a) An LLM generates n candidate specifications for I that vary in both interpretation and syntax (full prompts are given in Section A). Candidates that fail a *correctness check* (compilation in Lean and a check against a benchmark's provided test cases) are discarded, and the estimator selects the candidate with the lowest predicted cost.

Feedback Loop (Figure 2b) An agent generates one specification and then calls the quality estimator $\hat{C}$ in a rigid loop, producing a quality estimate each time. This estimate is fed back to a specification generator, yielding iterative improvement in the quality of the specification. As in the case of selection, the correctness of the specification is checked for compilation and against the provided test cases ($\mathcal{S}_C$). The loop runs for at most k turns. It may also start from a seed specification S_0, such as the benchmark's reference specification.

Free-Form Agent (Figure 2c) An agent receives the estimator and a correctness check as tools and is empowered to use them without restriction. During execution, we measure the number of times the agent calls the quality estimator.

4 Experiments

Our evaluation addresses three research questions:

- **RQ1 Prediction:** Can the cost of verified code generation be accurately predicted from the text of a specification?
- **RQ2 Usefulness:** Are specification quality estimates useful for improving the performance of verified code generation?
- **RQ3 Ablations:** What is the best way to harness specification quality estimates?

4.1 Setup and Implementation

We evaluate our approach on three benchmarks. The first is Verina (Ye et al., 2025), a set of 189 verified code generation tasks focusing on algorithmic problems. We also use AlgoVeri (Zhao et al., 2026), which contains 76 classical algorithm tasks; both Verina and AlgoVeri express specifications and write proofs in Lean 4 (de Moura & Ullrich, 2021). Finally, we use miniCodeProps (Lohn & Welleck, 2024), a set of 201 Lean theorems stating properties of small programs, to evaluate a free-form proving agent (RQ2).

Across all our experiments, code generation and proof writing are done by a standard Claude Code agent with the Opus 4.8 model; the agent is given access to Lean LSP tools (Dressler, 2025) for Verina and AlgoVeri, plus file reading and writing and bash. As listed in Table 1, we evaluate with several different LLM estimator models; these use a single chat completion that provides the model only the text of the specification and a short task context (the benchmark name, or the function signature in the feedback loop). Section B gives the prompts used for the LLM-based estimators.

We also use three classes of conventional regression models. First are three feature models that use 58 structural features of the specification (a random forest, `feat-rf`; a Gaussian process, `feat-gp`; and `pairwise-rf`, a random forest that predicts cost pairwise). We use two text models that embed the specification as TF-IDF word 1–2-grams reduced to 128 dimensions by SVD, followed by ridge regression (`embed-ridge`) or a pairwise classifier (`embed-pairwise`). Finally, we use a Weisfeiler-Lehman graph model with support vector regression (`wl-svr`). Trained models are evaluated with 12 task-grouped 80/20 splits (seeds 0–11), with all representations fit on the training folds only. The zero-shot estimators (LLMs and baselines) are scored once on all tasks.

4.2 RQ1 Prediction

To measure the accuracy of each specification quality estimator, we generated ten candidate specifications for each of the tasks in the Verina and AlgoVeri benchmarks according to the methodology described in Section 3.2; each specification is checked for correctness against the golden test set (*i.e.*, $\mathcal{S}_C$) provided by the benchmark set before estimation is applied. Table 1 reports the correlation between each estimator's prediction and the actual code generation and proof writing cost measured by time and by number of output tokens. We report the Spearman correlation ρ, which measures the correlation between rankings, since a specification's relative quality matters more than the absolute solving time (Spearman, 1904). We do not compute a Pearson correlation for the LLM-based predictors since they only rank the specifications rather than producing time estimates.

Findings We find that implementation generation and proof writing cost is predictable from the text of a specification before any implementation or proof is written: the within-task proof-cost estimates reach a correlation of $\rho = 0.58$ for Verina and $\rho = 0.47$ for AlgoVeri. The quality of predictions is noticeably better for the Verina benchmark; indeed, the baseline specification size (`post-bytes`) performs nearly as well as the trained estimators on AlgoVeri. Across both benchmarks, conventional regression models outperform LLM-based predictors, achieving, for example, $\rho = 0.582$ versus $\rho = 0.477$ on Verina, without any LLM inference cost.

4.3 RQ2 Usefulness

To evaluate whether specification quality estimates can be harnessed to reduce the cost of code generation and proof writing, we connect each estimator model to the selection and feedback loop methods of Figure 2; the results for Verina and AlgoVeri are reported in Table 2. Our results show that all estimator models lead to lower code generation and proof writing cost than the reference specifications in both the selection and loop regimes. For Verina, the best-performing estimator is `wl-svr` (63 seconds) in the selection regime, and `feat-gp` and `feat-rf` (81 seconds) in the loop regime; these times represent more than a $3\times$ improvement over the reference specification, which takes 265 seconds to generate code and proof. The results for LLM-based prediction with Opus 4.8 nearly match the regression models (67 seconds and 85 seconds, respectively). Solving rates are nearly saturated for Verina across all estimators.

For AlgoVeri, all estimator models again outperform the reference specification, which has implementation generation and proof writing cost of 818 seconds on average and solves 71.1% of tasks.

Table 1: Within-task Spearman (ρ) and Pearson (r) correlation between predicted and actual implement-and-prove cost. Standard deviations across splits are given in Section D.

	Verina				AlgoVeri			
	Time		Tokens		Time		Tokens	
model	ρ	r	ρ	r	ρ	r	ρ	r
`post-bytes`	0.454	0.459	0.427	0.414	**0.424**	0.449	0.367	0.363
`pairwise-rf`	**0.582**	**0.601**	**0.520**	**0.518**	0.385	0.433	0.346	0.374
`feat-rf`	0.537	0.566	0.455	0.478	0.297	0.332	0.311	0.310
`feat-gp`	0.522	0.569	0.456	0.482	0.277	0.335	0.266	0.302
`wl-svr`	0.520	0.566	0.442	0.480	0.404	**0.473**	**0.470**	**0.463**
`embed-pairwise`	0.415	0.436	0.383	0.366	0.196	0.224	0.181	0.211
`embed-ridge`	0.361	0.386	0.304	0.313	0.229	0.216	0.242	0.232
Opus 4.8	0.477	—	0.420	—	0.291	—	0.225	—
Qwen3-coder-30B	0.456	—	0.386	—	0.381	—	0.346	—
Qwen3-235B	0.409	—	0.327	—	0.297	—	0.305	—
Haiku 4.5	0.368	—	0.297	—	0.274	—	0.221	—
DeepSeek-V3.2	0.182	—	0.142	—	0.169	—	0.115	—

Table 2: Downstream implement-and-prove cost of the estimator-selected specification, by estimator and regime. The "Sec." columns give average wall-clock time in seconds.

	Verina				AlgoVeri			
	Selection		Loop		Selection		Loop	
Estimator	Solved	Sec.	Solved	Sec.	Solved	Sec.	Solved	Sec.
`post-bytes`	99.5%	101	100%	106	85.5%	422	71.1%	778
`pairwise-rf`	98.9%	68	100%	84	84.2%	**408**	71.1%	742
`feat-rf`	98.9%	68	99.5%	**81**	85.5%	478	72.4%	767
`feat-gp`	99.5%	78	100%	**81**	84.2%	444	71.1%	755
`wl-svr`	98.9%	**63**	100%	82	85.5%	417	69.7%	**622**
`embed-pairwise`	98.9%	86	100%	91	81.6%	482	72.4%	797
`embed-ridge`	100%	105	99.5%	91	82.9%	482	72.4%	754
Opus 4.8	100%	67	100%	85	84.2%	504	73.7%	668
Qwen3-coder-30B	100%	107	100%	109	85.5%	450	73.7%	760
Qwen3-235B	99.5%	116	99.5%	105	82.9%	506	71.1%	691
Haiku 4.5	100%	126	100%	134	84.2%	548	73.7%	675
DeepSeek-V3.2	99.5%	149	99.5%	122	82.9%	552	76.3%	736
Oracle	100%	47			85.5%	281		
Reference spec.		98.4% / 265				71.1% / 818		
First-generated		99.5% / 152				68.4% / 832		

In the selection regime, `pairwise-rf` reduces the generation time to just 408 seconds on average while raising the solved rate to 84.2%. In the loop regime, `wl-svr` reduces time to 622 seconds but also reduces the solved rate to 69.7%. The selection regime performs significantly better than the loop regime for AlgoVeri.

We quantify how much performance is "left on the table" by the estimators by reporting a notional oracle predictor in the selection regime; here, we assume advance knowledge of the generation cost for each of the ten generated specifications and choose the one known to be best. For Verina, the oracle achieves a solving time of 47 seconds; the gap between this average solving time and the 63 seconds achieved by `wl-svr` represents additional performance that could be gained by improving the accuracy of the estimator. For AlgoVeri, the oracle achieves an average generation time of 281 seconds; again, this beats the best estimator model.

Table 3: Results for free-form agentic proving on miniCodeProps with quality estimates.

Estimator tool	Proved	Time (hours)
Baseline	176/201	15.5
`wl-rf`	**182/201**	**5.0**
`struct-rf`	**182/201**	5.5
`char-rf`	**182/201**	6.9
`embed-ridge`	**182/201**	8.4
`embed-hgbr`	181/201	5.5
`embed-rf`	181/201	8.2
`struct-ridge`	180/201	5.1
`char-ridge`	180/201	6.7
`pairwise-rf`	180/201	8.6
`wl-svr`	176/201	9.0
Opus 4.8	169/201	9.9

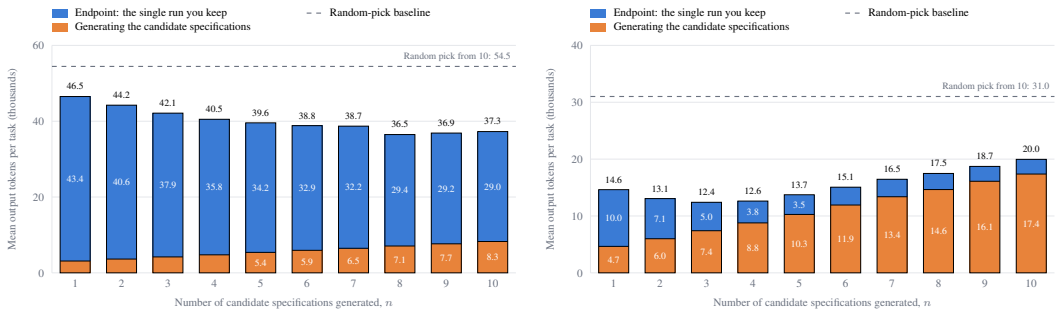
Proof Search We also evaluate the usefulness of specification quality estimates in the context of proof search using the miniCodeProps benchmark. This evaluation differs from those on Verina and AlgoVeri, since it only estimates the cost of writing a proof. The proof-writing agent follows the free-form approach of Figure 2c; it may call the quality estimator for a given Lean theorem to decide where to invest proving resources. The estimators are trained on the agent's own earlier proof attempts, with 5-fold cross-prediction grouped by problem, so no estimate for a problem comes from a model trained on that problem. Since the tasks are far longer-running than in Verina and AlgoVeri, we use a different set of estimators as listed in Table 3; each name combines a representation (`struct` for structural features, `embed` and `char` for TF-IDF word and character n-grams, and `wl` for Weisfeiler-Lehman graph features) with a model (`rf` for random forest, `ridge` for ridge regression, `hgbr` for histogram gradient-boosted regression, and `svr` for support vector regression). The results are given in Table 3. They show that every trained estimator lets the agent solve at least as many problems as the baseline in less time. The best, `wl-rf`, solves 182/201 problems in 5.0 hours, compared with 15.5 hours for the baseline.

Findings Specification quality estimates are useful for reducing the cost of code generation and proof writing. On both the Verina and AlgoVeri benchmark sets, SPEQUEST significantly speeds up downstream code generation and proof writing without a significant decrease in the number of solved instances. In the context of pure proof search (miniCodeProps), specification quality estimates allow an agent to reduce proof-writing time by $3\times$ while *increasing* the number of solved instances.

4.4 RQ3 Ablations

Selection Versus Feedback As we find in Table 2, both selection among a sample of ten specifications and iterative improvement of a specification in a feedback loop significantly improve performance on the Verina and AlgoVeri benchmark sets. For Verina, selection is slightly better: the best predictor achieves an average solving time of 63 seconds, compared to 81 for loop-based refinement. For AlgoVeri, the difference is more pronounced: the best estimator under the selection regime achieves a solving time of 408 seconds, versus 622 in the loop regime. The results suggest that choosing from a sample of possible specifications is a better use of quality estimates than refining specifications in a feedback loop.

Effort Tradeoff While our results suggest that estimating specification quality can significantly reduce downstream code generation and proof writing effort, generating additional specifications to select among is not free. We therefore also evaluate the tradeoff between the number of specifications generated and the benefits of selecting the best one. Figures 3a and 3b show the results of varying the number of candidate specifications n in the selection regime for two different specification generation methods. In Figure 3a, an agent plans ten possible specifications and then generates them in one go; in Figure 3b, an agent generates one specification at a time and is then prompted to produce a syntactically different specification each time. We measure the average number of output



(a) Plan-then-generate mode.

(b) Generate-in-loop mode.

Figure 3: Mean per-task output tokens vs. the number of candidate specifications n.

tokens used to generate n specifications (in orange) and the number used to generate code and write the proof (in blue) from the best of the n specifications available (an oracle, as in Section 4.3). The cost of generating specifications grows linearly with n, while the downstream cost decreases as n grows, since there is a larger set of specifications to choose from. The results indicate that there is a minimum where the tradeoff between these phenomena is best: for plan-then-generate, this minimum point is at $n = 8$, while for the generate-in-loop mode, the minimum occurs at $n = 3$.

Findings Both the selection and feedback loop modes are effective ways to make use of specification quality estimates; our results indicate that the selection regime performs better, especially for the AlgoVeri benchmark set. We also find that there is a tradeoff between the effort required to generate additional specifications and the benefits of choosing among them; the ideal tradeoff point depends on characteristics of the input programs and the method of specification generation.

5 Limitations and Future Work

Several limitations affect the applicability of our results. First, while our results indicate that specification quality estimates are useful in reducing the cost of program generation and proof writing in Lean, this work has not explored whether such estimates could also apply to solver-based verification or to other proof languages such as Rocq. Second, our results for Verina and AlgoVeri suggest that programs with different characteristics, like those in the two benchmark sets, might be more or less amenable to specification quality estimates. Future work could measure what features of an input program—or of candidate specifications—make specification quality estimates more or less useful in reducing downstream generation effort.

Finally, our results are largely limited to the domain of verified code generation. However, LLM-based agents are in wide use in many other domains, and increasingly use natural language plans to execute complex tasks (Wang et al., 2023; Huang et al., 2024). Future work could explore whether estimates of the quality of such plans, analogous to our estimates of specification quality, could reduce downstream reasoning effort in general agentic tasks. Outside of the domain of code, verifying that the result is correct is more complex; future work on estimating plan quality would therefore require new methods of accounting for incorrect results when constructing a training data set.

6 Conclusion

A poorly chosen specification can doom verified code generation. In this paper, we have addressed the task of estimating the quality of specifications before any code is generated from them or any proofs of correctness are written. We developed a theoretical model of specification variation and implemented practical estimators for specification quality. Our results show that the quality of a specification can be accurately predicted given only the text of the specification. We then showed that these estimates can be harnessed to reduce the cost of both verified code generation (by more than $3\times$) and pure proof writing (by up to $3\times$). Future work could explore extending quality estimates to additional verification languages, or to agentic tasks outside the coding domain.

AI USE STATEMENT

Generative AI tools form part of the methodology of our work: we used them to generate and modify specifications, programs, and proofs as described in Section 3 and evaluated their performance. In conducting the research, we also used generative AI tools to help engineer SPEQUEST's implementation and to monitor large-scale experimental runs. We also used AI tools to format figures, graphs, and tables in the submitted paper, to edit text, and to proofread and fix typesetting issues. Finally, we used generative AI tools to broaden our search for related work cited in Section 2. We did not use AI for any other purposes requiring disclosure.

All proofs generated by models were mechanically checked by the Lean kernel. We have manually checked that the implementations produced with the aid of coding assistants are substantially correct, and that the data presented in tables and figures faithfully reflect the results of the experiments we conducted. We have also verified that all descriptions of related work accurately reflect the contents of the cited papers. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We will release our implementation of SPEQUEST. The implementation of SPEQUEST consists of 9,000 lines of Python code implementing each estimator model, the agent estimators, and evaluation harnesses for measuring performance on the underlying benchmark sets as described in Section 3. Each trained model can be rebuilt from scratch, and, for experiments where we used selection among ten specifications, the implementation includes each of the ten generated specifications.

REFERENCES

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. In *Proceedings of the 42nd International Conference on Machine Learning*, pp. 587–615. PMLR, 2025. URL https://proceedings.mlr.press/v267/aggarwal25a.html.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Investigating Test Overfitting on SWE-bench. In *International Symposium on Foundations of Software Engineering*, 2026. URL https://arxiv.org/abs/2511.16858.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced Coding Benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024. URL https://arxiv.org/abs/2410.06992.

David Brandfonbrener, Simon Henniger, Sibi Raja, Tarun Prasad, Chloe R. Loughridge, Federico Cassano, Sabrina Ruixin Hu, Jianang Yang, William E. Byrd, Robert Zinkov, and Nada Amin. VerMCTS: Synthesizing Multi-Step Programs using a Verifier, a Large Language Model, and Tree Search. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024. URL https://openreview.net/forum?id=HmB9uZTzaD.

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

Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 9164–9175, 2023. URL https://arxiv.org/abs/2310.09342.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K. Lahiri, Tao Xie, and Lidong Zhou. SAFE: Automated Proof Generation for Rust Code via Self-Evolution. In *The 13th International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.15756.

Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Armen Der Kiureghian and Ove Ditlevsen. Aleatory or epistemic? Does it matter? *Structural Safety*, 31(2):105–112, 2009. doi: 10.1016/j.strusafe.2008.06.020.

Nongyu Di, Tianyu Chen, Shan Lu, Shuai Lu, Yeyun Gong, Peng Cheng, Jacob R. Lorch, Yuan Yao, and Xiaoxing Ma. Reducing the Costs of Proof Synthesis on Rust Systems by Scaling Up a Seed Training Set. *arXiv preprint arXiv:2602.04910*, 2026. URL https://arxiv.org/abs/2602.04910.

Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972. doi: 10.1145/355604.361591.

Oliver Dressler. lean-lsp-mcp: Lean theorem prover MCP server. https://github.com/oOo0oOo/lean-lsp-mcp, 2025.

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*, 1(FSE):1889–1912, 2024. URL https://arxiv.org/abs/2310.01831.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1229–1241, 2023. URL https://arxiv.org/abs/2303.04910.

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, pp. 328–343, 2017. doi: 10.1145/3064176.3064183.

Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5): 323–334, 1992.

Vladimir Gladshtein, Vitaly Kurin, Yueyang Feng, Dipesh Kafle, George Pîrlea, Qiyuan Zhao, and Ilya Sergey. Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean. In *International Conference on Computer Aided Verification*, 2026. URL https://ilyasergey.net/assets/pdf/papers/velvet-cav26.pdf.

Ian J. Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the planning of LLM agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.

Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. *Machine Learning*, 110(3):457–506, 2021. doi: 10.1007/s10994-021-05946-3.

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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Shuvendu K. Lahiri. Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages. In *Formal Methods in Computer-Aided Design*, pp. 142–147. IEEE, 2024. URL https://arxiv.org/abs/2406.09757.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=QQhQIqons0.

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical Proof Search for Open State-of-the-Art Code Verification. In *Conference on Language Modeling (COLM)*, 2026. URL https://arxiv.org/abs/2603.19329.

Evan Lohn and Sean Welleck. miniCodeProps: a Minimal Benchmark for Proving Code Properties. *arXiv preprint arXiv:2406.11915*, 2024. URL https://arxiv.org/abs/2406.11915.

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. ISSN 2835-8856. URL https://openreview.net/forum?id=yBgTVWccIx.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *IEEE/ACM 47th International Conference on Software Engineering*, pp. 16–28. IEEE, 2025. URL https://arxiv.org/abs/2401.08807.

Lezhi Ma, Shangqing Liu, Yi Li, Qiong Wu, Han Wang, and Lei Bu. SpecSyn: LLM-based Synthesis and Refinement of Formal Specifications for Real-world Program Verification. *arXiv preprint arXiv:2604.21570*, 2026. URL https://arxiv.org/abs/2604.21570.

Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-Assisted Synthesis of Verified Dafny Methods. In *Proceedings of the ACM on Software Engineering*, pp. 812–835, 2024. URL https://arxiv.org/abs/2402.00247.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024. doi: 10.1145/3660810. URL https://arxiv.org/abs/2310.10996.

Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. doi: 10.2307/1990888.

Cedric Richter and Heike Wehrheim. Beyond Postconditions: Can Large Language Models Infer Formal Contracts for Automatic Software Verification? *arXiv preprint arXiv:2510.12702*, 2025. URL https://arxiv.org/abs/2510.12702.

C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. ISSN 00029556. URL http://www.jstor.org/stable/1412159.

Elias Stengel-Eskin, Kyle Rawlins, and Benjamin Van Durme. Zero and Few-shot Semantic Parsing with Ambiguous Inputs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2306.00824.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A Curated Benchmark for Formally Verified Code Generation. *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2505.13938.

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 *IEEE/ACM 47th International Conference on Software Engineering*, 2025. doi: 10.1109/ICSE55347.2025.00161. URL https://arxiv.org/abs/2412.14063.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively Building Formal Proofs with Informal Reasoning. *arXiv preprint arXiv:2509.22819*, 2025. URL https://arxiv.org/abs/2509.22819.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, 2023.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data. *arXiv preprint arXiv:2405.14333*, 2024. URL https://arxiv.org/abs/2405.14333.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2):3454–3482, 2025. URL https://arxiv.org/abs/2409.13082.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, 2023. URL https://arxiv.org/abs/2306.15626.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation. *arXiv preprint arXiv:2505.23135*, 2025. URL https://arxiv.org/abs/2505.23135.

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. AlgoVeri: An Aligned Benchmark for Verified Code Generation on Classical Algorithms. *arXiv preprint arXiv:2602.09464*, 2026. URL https://arxiv.org/abs/2602.09464.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library Generation from Scratch. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' Propensity of Exploiting Test Cases. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.20270.

Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Qiaochu Jiang, and Sean Welleck. Premise Selection for a Lean Hammer. *arXiv preprint arXiv:2506.07477*, 2025. URL https://arxiv.org/abs/2506.07477.

A Specification Generation Prompts

The prompts below generate the $n = 10$ diverse candidate specifications per Verina task used in the selection regime. Generation is plan-first: a single PLAN turn enumerates 10 genuinely different readings, then one GENERATE turn per entry realizes it as concrete Lean 4. The generator sees only the natural language description and the Lean signature—never a reference implementation or reference specification.

System Prompt

```
You are an expert in Lean 4 and formal specification.  You write formal
specifications (preconditions and postconditions) that capture the intended
behavior of a program described in natural language.

You will be given a natural-language task description and the Lean 4 function
SIGNATURE (name, argument types, return type) -- but NOT any reference
implementation and NOT any reference specification.  You must invent the
specification yourself, from the description and signature alone.

A specification here has this shape (Verina benchmark convention):
- `precond`  : a `Prop` over the inputs -- the precondition.  Make it as
  PERMISSIVE as the description allows (do not over-constrain the inputs).
- `postcond` : a `Prop` over the inputs AND the `result` -- the postcondition,
  a sound and complete relationship between input and output.
- `precond_aux` / `postcond_aux` : OPTIONAL auxiliary `def`s you may introduce
  when a condition cannot be expressed inline.  Add `@[reducible, simp]`
  to them.

Lean 4 rules:
- Lean 4 syntax only (no Lean 3).  Do NOT import Std or Init;
  `import Mathlib` is already available.
- Use `a[i]!` (not `a[i]`) to index arrays/lists.
- The postcondition is elaborated with `result` bound to the function's
  return type, and the preconditions in scope.
```

Plan Turn

````
Here is the task.

===== NATURAL-LANGUAGE DESCRIPTION =====
{description}

===== LEAN 4 SIGNATURE (this is all the formal context you get) =====
```lean4
{scaffold}
```

STEP 1 -- PLAN {n} DIFFERENT SPECIFICATIONS.

The natural-language description is ambiguous and under-determined in several
ways.  Your job now is NOT to write Lean yet, but to plan {n} genuinely
DIFFERENT specifications of this task.  Each one must be:
  - FAITHFUL: a defensible reading of the natural-language description; and
  - DISTINCT from the others along at least one of these axes:
      * INTERPRETATION -- it resolves an ambiguity in the description
        differently (e.g. behavior on empty input, ties, duplicates,
        negative/zero, ordering, off-by-one boundaries, over/underflow,
        what counts as "valid" input);
      * FORM -- it states the same intent in a different logical shape
        (e.g. relational vs. functional/closed-form, existential vs.
        universal quantification, characterizing the result directly vs.
        via a uniqueness/optimality property, strong vs. permissive
        precondition, constructive vs. declarative).

First, briefly list the specific ambiguities and formal-form choices this task
admits.  Then output the plan as a fenced ```json block with EXACTLY this
shape:
````

````
```json
{
  "ambiguities": ["...", "..."],
  "specs": [
    {"id": 1, "interpretation": "<one-line reading being committed to>",
      "form": "<the logical form/shape>",
      "differs_by": "<how this differs from the others>"},
    ... {n} entries, ids 1..{n} ...
  ]
}
```

Make the {n} entries as diverse as the description honestly allows.  Do not
produce {n} cosmetic variants of one idea; if the task is nearly unambiguous,
say so in "ambiguities" and still find {n} distinct FORMS.
````

Generate Turn (Once per Plan Entry)

```
STEP 2 -- GENERATE SPEC #{i} of {n}.

Realize plan entry #{i} as concrete Lean 4.  Recall it:
  interpretation: {interpretation}
  form:           {form}
  differs_by:     {differs_by}

It must stay FAITHFUL to the natural-language description and DISTINCT from
the specs you already produced above.  You may use the lean-lsp tools
(lean_run_code, lean_verify, lean_diagnostic_messages, leansearch/loogle) to
check that your precond/postcond elaborate against the signature before you
commit -- a spec that does not typecheck is worthless.

Output ONLY this fenced ```json block, nothing else:
```

````
```json
{
  "id": {i},
  "precond_aux": "<Lean 4 aux defs for the precondition, or empty string>",
  "precond": "<Lean 4 Prop for the precondition>",
  "postcond_aux": "<Lean 4 aux defs for the postcondition, or empty string>",
  "postcond": "<Lean 4 Prop over inputs and `result`>"
}
```
````

Test-in-the-Loop Generation In test-in-the-loop generation (Figure 3b), the system prompt and Plan turn are unchanged, but the Generate turn tells the model that each candidate will be compiled and tested:

```
STEP 2 -- GENERATE SPEC #{i} of {n}.

Realize plan entry #{i} as concrete Lean 4.  Recall it:
  interpretation: {interpretation}
  form:           {form}
  differs_by:     {differs_by}

It must stay FAITHFUL to the natural-language description and DISTINCT from the
specs you already produced above.  Your precond/postcond will be (a) compiled
against the real signature and (b) exercised against a hidden battery of
example input/output pairs; you will get feedback and may revise.  Use the
```

```
lean-lsp tools (lean_run_code, lean_verify, lean_diagnostic_messages,
leansearch/loogle) to sanity-check that it elaborates before you commit.

Output ONLY this fenced ```json block, nothing else:
{same JSON block as the Generate turn above}
```

A candidate that does not compile or that disagrees with the benchmark's test cases receives one of the following repair messages, for at most three attempts per candidate. The {failures} field names up to three failing checks, each with a single counterexample input; the model never sees the full test suite.

```
Spec #{i} attempt {attempt} DID NOT COMPILE against the real signature.

Lean reported:
```
{stderr}
```

Fix the specification so it elaborates.  Keep the SAME interpretation
("{interpretation}"); only repair the Lean.  Re-emit the full JSON block.

Spec #{i} attempt {attempt} COMPILES but is NOT FAITHFUL to the intended
behavior -- it disagrees with worked examples of the task.

{failures}
Revise precond/postcond so they agree with these examples while remaining a
defensible reading of the description and DISTINCT from your earlier specs.
These are individual examples, not the whole spec -- reason about the general
rule they imply.  Re-emit the full JSON block.
```

If fewer than n distinct candidates pass these checks, the model is asked for additional candidates, up to five times:

```
BACKFILL -- you have {have} faithful, distinct specs so far but we want {n}.

Produce ONE more specification that is (a) FAITHFUL to the description,
(b) DISTINCT from every spec above (a new interpretation or a new logical
form -- not a cosmetic rewrite), and (c) will pass the example checks.  It will
be compiled and tested the same way; you may revise on feedback.

Output ONLY the JSON block (use id {i}).
```

AlgoVeri For AlgoVeri, the scaffold is the task's Lean file with the bodies of the reference precondition and postcondition replaced by `True`, and candidates are compiled but not tested. The system prompt is adapted accordingly:

```
You are an expert in Lean 4 and formal specification.  You write formal
specifications (preconditions and postconditions) that capture the intended
behavior of a program described in natural language.

You will be given a natural-language task description and the task's Lean 4
file with the reference precondition and postcondition bodies BLANKED (replaced
by `True`) -- you see the imports, helper structures/definitions, and the exact
function signatures, but NOT the reference specification.  You must invent the
specification yourself, from the description and this scaffold alone.

A specification here has this shape (AlgoVeri benchmark convention):
- `precond`  : a `Prop` over the inputs -- the body of the `<name>_precond`
  definition.  Make it as PERMISSIVE as the description allows (do not
  over-constrain the inputs).
```

```
- `postcond` : a `Prop` over the inputs AND the `result` -- the body of the
  `<name>_postcond` definition; a sound and complete relationship between
  input and output.  Note `h_precond` is in scope there.
- `precond_aux` / `postcond_aux` : OPTIONAL auxiliary `def`s you may introduce
  when a condition cannot be expressed inline; they are inserted above the
  definition that uses them.  Add `@[reducible, simp]` to them.

Lean 4 rules:
- Lean 4 syntax only (no Lean 3).  Do NOT import Std or Init; `import Mathlib`
  is already available.
- Your Props are spliced verbatim into the marked region bodies -- write bare
  Prop expressions (no `def`, no `:=`).
- The file already contains helper defs/structures; you may reference them.
```

The Plan turn is the same except that the scaffold is headed `LEAN 4 SCAFFOLD (reference spec bodies blanked; this is all the formal context you get).` In the Generate turn, the sentence about compilation and testing is replaced by `Your precond/postcond will be compiled against the real signature; you will get the Lean errors back and may revise.`, and the backfill message asks for candidates that are well-formed and compile rather than candidates that pass the example checks.

Feedback Loop The feedback loop uses the same system prompt as generation. Its first turn is:

````
Write a formal Lean 4 specification (precondition + postcondition) for the task
below.  You will NOT write the implementation or proof -- a separate automated
agent will later have to (a) write an implementation of the signature and
(b) prove it satisfies your specification.

## Task description
{description}

## Lean scaffold (signature the spec constrains)
```lean
{scaffold}
```

Your spec must be FAITHFUL to the description: it will be compiled and then
exercised against a hidden battery of worked examples; you will get feedback
and may revise.

After each FAITHFUL candidate you produce, the harness will report the output
of a DIFFICULTY PREDICTOR: an estimate of how hard your spec will be for the
downstream agent to implement-and-prove (lower = cheaper), plus where it sits
among your previous candidates.  Your goal is to MINIMIZE predicted difficulty
while staying faithful.  You will get {k} scored attempts.

Use the lean-lsp tools (lean_run_code, lean_diagnostic_messages, leansearch,
loogle) to sanity-check that your Lean elaborates before you commit.

Output ONLY this fenced ```json block, nothing else:
{JSON block with precond_aux, precond, postcond_aux, postcond}
````

Candidates that do not compile or that disagree with the test cases receive repair messages like those of test-in-the-loop generation. After each candidate that passes, the model receives the estimate:

```
Your candidate #{idx} is FAITHFUL.  Difficulty predictor says:

  predicted difficulty: {score}
  vs your previous faithful candidates: {comparison}
```

```
{best note}You have {left} scored attempt(s) left.  Produce a REVISED spec you
believe the predictor will score LOWER (easier), while remaining faithful to
the description.  Meaningful restructuring (different logical form, simpler
obligations, more computational phrasing) beats cosmetic edits.  If you think
this candidate is already minimal, you may re-emit it unchanged and say so.
Re-emit the full JSON block.
```

Here {comparison} states whether the candidate is the best, tied with the best, or the worst so far, or how many previous candidates it beats, and {best note} gives the best score so far when the candidate did not improve on it. The loop stops after k scored candidates or after two consecutive candidates that do not improve the best score. For AlgoVeri, which has no test cases, "FAITHFUL" is replaced by "COMPILES" throughout, and the instructions additionally ask the model not to weaken the specification into something the description does not support.

B Specification Quality Estimator Prompts

The LLM-based estimators score one candidate specification per call with a single chat completion (no tools and no agent loop). The model sees only a short task context and the specification text; it never sees the natural language description, the benchmark's tests, or a reference implementation. In the selection mode the task context is the benchmark name; in the feedback loop it is the task's Lean scaffold (the function signature). If the reply does not parse, the model receives one repair message.

System Prompt

```
You are an expert in formal verification, judging how DIFFICULT a candidate
formal specification will be for a downstream automated agent, which must
(a) produce an implementation/model of its own choosing and (b) formally
prove it satisfies the specification. The agent may pick any implementation
it likes -- a spec is easy if SOME easily-provable implementation satisfies it.

Judge ONLY from the specification text and the task context you are shown:
logical structure, quantifiers, connectives, breadth of library surface,
likely induction/invariant burden. Reply with ONLY a JSON object
{"score": <integer 0-100, higher = harder>, "rationale": "..."}.
```

User Message

````
## Task context
```
{task context}
```

## Candidate specification
```
{precondition helpers, precondition, postcondition helpers, postcondition}
```
Score this candidate (0-100 difficulty). JSON only.
````

Repair Message (Sent Only If the Reply Does Not Parse)

```
Invalid ({parse error}). Reply with ONLY {"score": <integer 0-100>,
"rationale": "..."}.
```

Free-Form Proof Search Agent In the proof search experiments on miniCodeProps (Table 3), the baseline agent receives the following prompt, where {check command} is the command that runs the Lean checker:

```
Prove the Lean 4 theorem in ./problem.lean.

The file contains the definitions it depends on (the header) followed by the
theorem statement ending in `:= by` and a `sorry`. Replace the `sorry` with a
complete proof. You may add helper lemmas ABOVE the theorem, but you must NOT
modify the header definitions or the theorem statement itself.

Environment facts (important):
- There is NO lake project and NO `lean` CLI here. The ONLY way to check your
  file is: {check command}
  It prints PASS / PASS_WITH_SORRY / FAIL with error messages (Mathlib v4.15).
  Check frequently -- after every tactic step or lemma change.
- No internet access. Mathlib v4.15 only (some newer lemma names don't exist).
- Read ./.claude/skills/lean-proof/SKILL.md first and follow its methodology
  (one step at a time, hardest case first, error priority).

You are done ONLY when the checker prints PASS (no sorry anywhere). Work until
it does or you run out of ideas. Do not ask questions; just work.
```

When an estimator is available, the prompt continues:

```
You also have an MCP tool `estimate_proving_time` (from the `estimator`
server): give it a complete standalone Lean 4 theorem statement and it
returns the predicted proving wall-time in seconds plus a difficulty band,
from a model trained on measured proving times for exactly this corpus.
Tool calls are instant and free -- use them liberally to avoid wasting time:
- Before committing to a proof plan that needs helper lemmas, write each
  candidate lemma statement and estimate it. Prefer plans whose subgoals are
  all cheap; a `very_hard` subgoal is a strong signal to find another route.
- When two decompositions could work, estimate both and take the cheaper one.
- If a goal you are grinding on is predicted expensive, step back and look
  for an easier reformulation instead of pushing on.
Log of your estimates is kept automatically; call the tool with just the
statement text.
```

The tool is described to the agent as follows:

```
Estimate how long it will take to prove a Lean 4 theorem statement, using a
model trained on measured proving times for this corpus. Input the COMPLETE
standalone theorem statement (with binders and type ascriptions, ending in
':= by sorry' or ':= by'). Returns predicted wall-clock seconds and a
difficulty band. Use it to compare alternative helper lemmas / subgoals
BEFORE investing effort: prefer decompositions whose subgoals are all cheap,
and avoid grinding on predicted-expensive intermediate goals.
```

The tool returns the predicted time in seconds and minutes together with a difficulty band (easy, moderate, hard, or very hard). When the Opus 4.8 estimator backs the tool, it receives the statement with the following prompt, and its reply is returned to the agent in the same format:

```
You estimate proving cost for an automated LLM-based prover (Claude-class
agent with Lean 4 + Mathlib, checking via a compiler loop).

Given the Lean 4 theorem statement below, predict the wall-clock seconds the
prover will need to produce a complete verified proof. Calibration guide: a
statement closed by rfl/simp/omega in one or two attempts ~30-60s; easy
structural induction ~120-300s; multi-lemma inductions with some search
~300-1200s; deep invariants, nested recursion/termination reasoning, or
statements needing many auxiliary lemmas ~1200-3600s or more.

Statement:
```

```
```lean
{statement}
```

Output ONLY a JSON object: {"estimated_seconds": <number>}. No other text.
```

C Estimator and Training Details

This appendix describes the conventional estimators of Section 4.1 and the protocols used to train and evaluate them. The LLM-based estimators are described in Section B.

Structural Features The feature models (`feat-rf`, `feat-gp`, and `pairwise-rf`) use 58 numeric features computed from the specification text after Lean comments and string literals are removed. The features fall into eight groups:

- *Size* (17): the number of bytes, characters, non-empty lines, and tokens in the precondition, the postcondition, the auxiliary definitions, and the whole specification, plus the maximum line length.
- *Lexical* (5): the number of identifiers, distinct identifiers, qualified (dotted) identifiers, numeric literals, and Lean keywords.
- *Logical structure* (12): the number of universal and existential quantifiers and their total, the number of quantifier alternations, the number of conjunctions, disjunctions, negations, implications, and bi-implications and their total, the number of top-level conjuncts in the postcondition, and whether the precondition is trivial.
- *Relations and arithmetic* (6): the number of equalities, disequalities, inequalities, membership tests, arithmetic operators, and index accesses.
- *Nesting* (7): the maximum bracket depth and the number of parentheses, square brackets, lambdas, and `let`, `if`, and `match` expressions.
- *Auxiliary definitions* (3): the number of auxiliary definitions, whether any of them is recursive, and the number of extra imports.
- *Library surface* (5): the number of references to list operations, order relations (*e.g.*, `Sorted`, `Perm`), arithmetic functions, string operations, and type names.
- *Ratios* (3): the ratio of postcondition to precondition tokens, the fraction of characters in auxiliary definitions, and the density of quantifiers and connectives per token.

The logical-structure, relation, and nesting features are computed over the precondition and postcondition only; the size and library-surface features also include the auxiliary definitions.

Models Each regression model predicts $\log(1 + y)$, where y is either the wall-clock time or the number of tokens of implementation generation and proof writing; a separate model is trained for each target.

- `feat-rf`: a random forest regressor with 300 trees over the 58 features.
- `feat-gp`: Gaussian process regression over the standardized features, with an RBF kernel (initial length scale $\sqrt{58}$) plus a white-noise kernel and normalized targets.
- `pairwise-rf`: a random forest classifier with 200 trees, trained on every ordered pair of specifications of the same task. The input is the difference of the two feature vectors and the label is whether the first specification is cheaper; pairs with equal cost are discarded. A specification's score is its mean predicted probability of being cheaper than each of the other specifications of its task.
- `embed-ridge` and `embed-pairwise`: the specification text is embedded as TF-IDF word unigrams and bigrams (minimum document frequency 2, at most 20,000 terms) reduced to 128 dimensions by truncated SVD. `embed-ridge` applies ridge regression ($\alpha = 1$); `embed-pairwise` applies logistic regression to the same within-task pairs as `pairwise-rf`.

- `wl-svr`: the first 400 tokens of the specification text form a chain graph, which is relabeled for two Weisfeiler-Lehman iterations; the counts of the resulting labels are the input to a support vector regressor with a linear kernel.
- `post-bytes`: the number of bytes in the postcondition and its auxiliary definitions.

Evaluation Protocol For the correlations of Table 1, each trained model is evaluated on 12 random 80/20 splits of the tasks (seeds 0–11), so that all specifications of a task fall on the same side of a split; the TF-IDF, SVD, and Weisfeiler-Lehman vocabularies are fit on the training tasks only. For each held-out task with at least three specifications, we compute the Spearman and Pearson correlations between predicted and measured cost, average them across tasks after a Fisher z-transform, and then average across splits. `post-bytes` and the LLM estimators require no training; they are scored once and averaged over all tasks. For the selection results of Table 2, each task's specifications are scored by a model trained on the other tasks using 5-fold cross-prediction grouped by task. In the feedback loop, each trained model is refit once on all of the benchmark's specifications and then frozen; pairwise models score a candidate by its mean predicted probability of being cheaper than the loop's earlier candidates, with a neutral score for the first candidate.

Proof Search Estimators The estimators of Table 3 predict $\log(1 + t)$, where t is the agent's measured wall-clock time on a problem; for failed attempts, t is clipped to between 900 and 3,600 seconds. The training data are the theorem statements that the agent submitted to the estimator tool in earlier runs, each labeled with the outcome of the problem it was submitted for. The representations are

- `struct`: 15 structural counts of the statement (*e.g.*, its length, number of quantifiers, arrows, and comparisons, and references to `List` and `Nat`);
- `embed`: TF-IDF word unigrams and bigrams (minimum document frequency 3) reduced to at most 256 dimensions by truncated SVD;
- `char`: TF-IDF character 3–5-grams (at most 20,000 terms, sublinear term frequency);
- `wl`: Weisfeiler-Lehman features as for `wl-svr` above.

The models are a random forest with 200 trees (`rf`), ridge regression with $\alpha = 1$ (`ridge`), histogram gradient-boosted regression with default settings (`hgbr`), and a linear support vector regressor (`svr`). `pairwise-rf` scores a statement by its mean predicted probability of being harder than a fixed set of anchor statements from the training data. The problems are divided into five folds with a fixed random seed, and during evaluation each problem is served by the model trained on the other four folds.

D Correlation Variance

Table 4 repeats Table 1 with the standard deviation of each trained model's correlation across the 12 task-grouped splits described in Section C. `post-bytes` and the LLM estimators are scored once on all tasks and therefore have no split-to-split variation.

Table 4: Within-task Spearman (ρ) and Pearson (r) correlation between predicted and actual implement-and-prove cost, with standard deviations across the 12 task-grouped splits. Standard deviations are not available for the token-based Pearson correlations.

	Verina				AlgoVeri			
	Time		Tokens		Time		Tokens	
model	ρ	r	ρ	r	ρ	r	ρ	r
`post-bytes`	0.454	0.459	0.427	0.414	**0.424**	0.449	0.367	0.363
`pairwise-rf`	**0.582** $\pm$ 0.035	**0.601** $\pm$ 0.033	**0.520** $\pm$ 0.042	**0.518**	0.385 $\pm$ 0.075	0.433 $\pm$ 0.090	0.346 $\pm$ 0.078	0.374
`feat-rf`	0.537 $\pm$ 0.035	0.566 $\pm$ 0.039	0.455 $\pm$ 0.057	0.478	0.297 $\pm$ 0.125	0.332 $\pm$ 0.133	0.311 $\pm$ 0.085	0.310
`feat-gp`	0.522 $\pm$ 0.035	0.569 $\pm$ 0.024	0.456 $\pm$ 0.036	0.482	0.277 $\pm$ 0.105	0.335 $\pm$ 0.121	0.266 $\pm$ 0.080	0.302
`wl-svr`	0.520 $\pm$ 0.037	0.566 $\pm$ 0.039	0.442 $\pm$ 0.065	0.480	0.404 $\pm$ 0.096	**0.473** $\pm$ 0.098	**0.470** $\pm$ 0.083	**0.463**
`embed-pairwise`	0.415 $\pm$ 0.041	0.436 $\pm$ 0.037	0.383 $\pm$ 0.047	0.366	0.196 $\pm$ 0.128	0.224 $\pm$ 0.098	0.181 $\pm$ 0.115	0.211
`embed-ridge`	0.361 $\pm$ 0.040	0.386 $\pm$ 0.047	0.304 $\pm$ 0.044	0.313	0.229 $\pm$ 0.088	0.216 $\pm$ 0.091	0.242 $\pm$ 0.083	0.232
Opus 4.8	0.477	—	0.420	—	0.291	—	0.225	—
Qwen3-coder-30B	0.456	—	0.386	—	0.381	—	0.346	—
Qwen3-235B	0.409	—	0.327	—	0.297	—	0.305	—
Haiku 4.5	0.368	—	0.297	—	0.274	—	0.221	—
DeepSeek-V3.2	0.182	—	0.142	—	0.169	—	0.115	—