Intent-aligned Formal Specification Synthesis via Traceable Refinement

Zhe Ye[†1] **Aidan Z.H. Yang**[2] **Huangyuan Su**[†3] **Zhenyu Liao**[2] **Samuel Tenka**[2] **Zhizhen Qin**[2] **Udaya Ghai**[2] **Dawn Song**[1] **Soonho Kong**[2]

Abstract

Large language models are increasingly used to generate code from natural language, but ensuring correctness remains challenging. Formal verification offers a principled way to obtain such guarantees by proving that a program satisfies a formal specification. However, specifications are frequently missing in real-world codebases, and writing high-quality specifications remains expensive and expertise-intensive. We present VERISPECGEN, a traceable refinement framework that synthesizes intent-aligned specifications in Lean through requirement-level attribution and localized repair. VERISPECGEN decomposes natural language into atomic requirements and generates requirement-targeted tests with explicit traceability maps to validate generated specifications. When validation fails, traceability maps attribute failures to specific requirements, enabling targeted clause-level repairs. VERISPECGEN achieves 86.6% on VERINA SpecGen task using Claude Opus 4.5, improving over baselines by up to 31.8 points across different model families and scales. Beyond inference-time gains, we generate 343K training examples from VERISPECGEN refinement trajectories and demonstrate that training on these trajectories substantially improves specification synthesis by 62–106% relative and transfers gains to general reasoning abilities.

1. Introduction

Large language models (LLMs) are increasingly used to generate code from natural language instructions (Chen et al., 2021; Roziere et al., 2023), enabling rapid implementation while introducing new challenges for correctness assurance. Recent studies have shown that LLM-generated code frequently contains functional errors (Wang et al., 2025) and security vulnerabilities (Guo et al., 2024; Yang et al., 2024), and these issues typically require costly human review to resolve. Formal verification offers a principled approach to automated correctness assurance by proving that a program satisfies a *formal specification*—a machine-checkable behavioral contract usually expressed as preconditions and postconditions (Hoare, 1969). Recent work demonstrates that LLMs can assist verification workflows by generating specifications or proofs (Sun et al., 2024; Misu et al., 2024; Aggarwal et al., 2024; Ye et al., 2025), suggesting the potential for end-to-end verifiable code generation.

However, the strength of verification guarantees is fundamentally bounded by specification quality, yet synthesizing correct specifications remains challenging even for human experts (Woodcock et al., 2009). First, specifications require precise intent alignment between informal descriptions and formal contracts. Weak specifications permit buggy implementations to pass verification, while incorrect ones reject correct implementations, undermining the value of verification in either case. Achieving this alignment demands translating potentially ambiguous natural language into unambiguous preconditions and postconditions capturing exactly the intended behavior. Second, training data for specification synthesis is scarce because formal specifications are rarely present in real-world codebases and remain expensive to write. Recent benchmarks show that even state-of-the-art models struggle to reliably synthesize specifications (Sun et al., 2024; Ye et al., 2025; Thakur et al., 2025), demonstrating that specification generation remains a substantial bottleneck for verifiable code generation. Existing approaches to specification synthesis, as discussed in Section 2, either anchor specification correctness on program artifacts (Ernst et al., 2007; Ma et al., 2025; Jin et al., 2024; Sun et al., 2025) or fail to effectively repair misaligned specifications (Ghosh et al., 2016; Cosler et al., 2023; Endres et al., 2024; Sun et al., 2024; Cao et al., 2025).

We address this challenge with VERISPECGEN, a traceable refinement framework that synthesizes intent-aligned formal specifications in Lean (de Moura et al., 2015) through requirement-level attribution and localized repair. Given a natural language problem description, VERISPECGEN

[†]Work done during internship at Amazon Web Services. [1]UC Berkeley [2]Amazon Web Services [3]Harvard University. Correspondence to: Soonho Kong <soonho@amazon.com>.

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

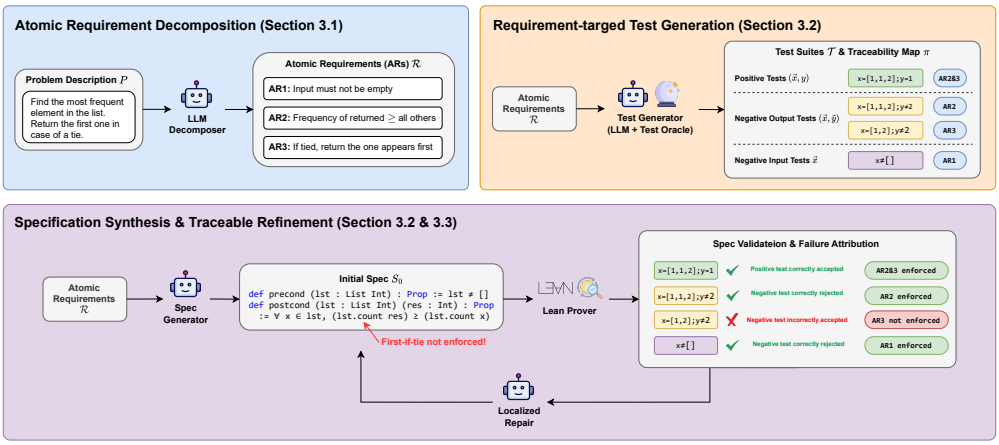

Figure 1. **VERISPECGEN traceable refinement workflow.** Given a natural language problem description (e.g., "Find the most frequent element..."), VERISPECGEN synthesizes formal specifications through three stages: (1) **Atomic Requirement Decomposition**: An LLM decomposes the description into testable atomic requirements (AR1: non-empty input, AR2: frequency constraint, AR3: tie-breaking rule). (2) **Requirement-targeted Test Generation**: For each requirement, the system generates positive tests (valid input-output pairs), negative-output tests (invalid outputs for valid inputs), and negative-input tests (invalid inputs), creating an explicit traceability map π linking each test to its validating requirements. (3) **Specification Synthesis & Traceable Refinement**: An LLM generates an initial Lean specification from the atomic requirements. A Lean prover validates the specification against all tests. When validation fails (e.g., test $\langle[1, 2], y{\neq}2\rangle$ incorrectly accepted), the traceability map attributes the failure to specific requirements (AR3 via π), enabling **localized repair** that modifies only the affected postcondition clause rather than rewriting the entire specification. This process iterates until all tests pass, producing intent-aligned specifications.

decomposes it into informal atomic requirements and generates requirement-targeted test cases for each requirement, including positive examples that should satisfy the specification and negative examples that should be rejected. Test outputs are labeled by executing a reference implementation as a black-box oracle, without inspecting code structure or deriving specification content from the implementation. Each test is paired with an explicit traceability map that links back to the requirements it validates. VERISPECGEN generates an initial specification from the atomic requirements and validates it by instantiating each test into a Lean proposition and attempting to prove or disprove it using proof automation. When validation fails, the traceability map attributes failures to specific requirements, enabling localized clause-level repairs that modify only the affected contract components. This process iterates until all tests pass and adversarial testing does not reveal missing constraints, yielding specifications that align with natural language intent through targeted, requirement-grounded refinement.

To address the scarcity of training data for specification synthesis, we generate a large-scale dataset from VERISPECGEN's refinement process. We execute VERISPECGEN with Claude Sonnet 4.5 (Anthropic, 2025b) on TACO-verified (Li, 2024), producing structured trajectories containing requirement decompositions, traceability maps, intermediate specifications, and localized repairs. We distill these trajectories into supervised fine-tuning examples, yielding 343,827 high-quality instruction-response pairs.

We evaluate VERISPECGEN on the SpecGen task of the VERINA benchmark. Across multiple model scales, traceable refinement improves VERINA SpecGen scores by up to 31.8 points over the benchmark baselines, reaching 86.6 points with Claude Opus 4.5 (Anthropic, 2025a) and achieving state-of-the-art performance. Ablation studies show that all framework components are necessary for effective refinement, with requirement decomposition as the most critical component. Beyond inference-time improvements, we demonstrate that training on trajectory-distilled data substantially improves base model capabilities. Fine-tuning Qwen3-4B-Instruct-2507 and Qwen3-Coder-30B-A3B (Yang et al., 2025) on our dataset improves both VERINA SpecGen scores by 15.6–18.2 points (62–106% relative) and VERINA CodeGen scores by 18.2–21.5 points (54–72% relative), and transfers improvements to out-of-domain math reasoning and general coding benchmarks.

In summary, our contributions are:

- **Traceable refinement for specification synthesis.** We introduce traceable refinement that enables targeted, localized specification repairs. By explicitly mapping validation failures to specific natural language requirements, VERISPECGEN achieves state-of-the-art specification synthesis, improving over direct generation baselines by up to 31.8 percentage points across model scales.
- **Scalable dataset generation via trajectory distillation.** We generate the first large-scale and high-quality dataset of traceable refinement trajectories with 343K examples. Training on these trajectories improves specification synthesis by up to 106% relatively while transferring to

out-of-domain reasoning tasks, providing a scalable path to improve specification synthesis models.

2. Background and Related Works

Specification synthesis paradigms. Specification synthesis approaches fall into two paradigms. Program-behavior-based methods infer specifications from code via dynamic analysis (Ernst et al., 2007) or LLMs (Ma et al., 2025; Wen et al., 2024; Sun et al., 2025), excelling when implementations are trusted ground truth. Intent-anchored methods synthesize specifications from natural language (Ghosh et al., 2016; Cosler et al., 2023; Endres et al., 2024; Sun et al., 2024; Mugnier et al., 2025; Mukherjee & Delaware, 2024; Cao et al., 2025), matching workflows where LLM-generated code may be buggy. However, prior intent-anchored systems cannot determine whether specifications capture all intended requirements and rely on coarse refinement signals (type errors, proof failures) that cannot localize failures to specific requirements. VERISPECGEN follows the intent-anchored paradigm, deriving specification content from natural language while using implementations solely as black-box oracles for test labeling rather than extracting specification logic. VERISPECGEN addresses the refinement gap through traceable refinement: decomposing natural language into atomic requirements, generating requirement-targeted tests, and using traceability maps for requirement-level attribution and localized repair.

Refinement mechanisms and trajectory-based learning. Most prior work employs coarse refinement signals such as verifier errors or global test outcomes (Madaan et al., 2023; Chen et al., 2023b) that indicate failure without localizing the source of misalignment. Fine-grained signals like mutation testing (Sun et al., 2025) provide counterexamples but do not map directly to requirements in the natural-language description. Recent work demonstrates the effectiveness of learning from refinement trajectories for general reasoning tasks (Zelikman et al., 2022; Chen et al., 2023a; Shinn et al., 2023) and formal artifact generation (Lin et al., 2024). To the best of our knowledge, VERISPECGEN is the first to provide requirement-level attribution through requirement decomposition, enabling targeted clause-level repairs. VERISPECGEN further generates a large-scale trajectory dataset for training purposes that provides process-level supervision for achieving intent alignment.

3. Traceable Refinement for Specification Synthesis

We propose VERISPECGEN, an agentic framework for synthesizing intent-aligned formal specifications through traceable refinement. The workflow is illustrated in Figure 1.

Task definition. Given a natural-language problem description P and a function signature σ that defines the input/output types, our goal is to synthesize an intent-aligned Lean 4 specification $S = \langle \mathsf{pre}(\vec{x}), \mathsf{post}(\vec{x}, y) \rangle$. The precondition $\mathsf{pre}(\vec{x})$ constrains the set of valid inputs $\vec{x}$ and should properly reject invalid inputs according to P. The postcondition $\mathsf{post}(\vec{x}, y)$ characterizes the set of correct outputs y for each valid input and should properly reject undesired outputs according to P.

Framework overview. Specification synthesis requires high-quality refinement signals to guide iterative improvement, but such signals are difficult to obtain. Type checking and proof failures provide only coarse feedback indicating global inconsistency, while direct LLM-based judging on specification is unreliable and may hallucinate alignment. VERISPECGEN addresses this through *traceable refinement*. We decompose the natural-language description into *atomic requirements*, which are informal statements each expressing a single testable behavioral property (Section 3.1). For each requirement, we generate targeted test cases that establish an explicit *traceability map* linking tests to requirements (Section 3.2). When tests fail, we trace failures back to implicated requirements and perform localized repairs on the specification (Section 3.3). This process continues until all tests pass, followed by adversarial testing to expose missing constraints. During refinements, we collect the trajectories for fine-tuning dataset construction (Section 3.4).

3.1. Decomposition into Atomic Requirements

To enable fine-grained refinement feedback, VERISPECGEN decomposes the natural-language description P into a set of atomic requirements (ARs) $\mathcal{R} = \{r_i\}_{i=1}^{m}$ using an LLM. An atomic requirement r_i is an informal natural-language statement expressing a *single* behavioral property that can be validated on concrete input-output examples, rather than bundling multiple properties in one requirement. We prompt the model to extract such requirements from P, explicitly requesting both functional requirements and edge-case requirements including boundary conditions and tie-breaking rules. VERISPECGEN further leverages an LLM-based judge to review and improve $\mathcal{R}$. The resulting requirement set $\mathcal{R}$ serves as the unit of traceability for test generation (Section 3.2) and failure attribution (Section 3.3). For example in Figure 1, the example description decomposes into three atomic requirements.

3.2. Requirement-Targeted Tests Generation and Specification Validation

VERISPECGEN uses requirement-targeted tests as *grounded* validation signals for a candidate specification S, instead of purely subjective LLM judgment directly. For each requirement $r \in \mathcal{R}$, we construct a set of labeled test cases intended to validate the behavioral property r. Concretely, we gener-

ate three types of tests: (i) *positive tests* $(\vec{x}, y)$ that should be accepted by the candidate specification, (ii) *negative-input tests* $\vec{x}$ that should be rejected by the candidate precondition, and (iii) *negative-output tests* $(\vec{x}, \tilde{y})$ that should be rejected by the candidate postcondition for valid inputs.

Test oracles for output labeling. To label test outputs, we require a test oracle $\mathcal{O}$ that provides ground-truth outputs for given inputs. We instantiate $\mathcal{O}$ as a black-box execution oracle using a reference implementation, either provided with P or generated by an LLM. $\mathcal{O}$ is agnostic to individual atomic requirements. Mining specifications directly from implementations (e.g., via symbolic execution or program analysis) would anchor validation to potentially buggy code rather than the user's intent. Instead, we use the oracle only for labeling concrete test outputs, not for extracting specifications, and therefore maintain intent-anchored validation where specification correctness is judged against natural-language requirements.

Test generation. The inputs $\vec{x}$ for positive and negative-input tests are generated by prompting an LLM with the specific requirement r to match its intent, emphasizing boundary values and adversarial cases. To label outputs for positive and negative-output tests, we query the oracle $y := \mathcal{O}(\vec{x})$ and perturb y into different $\tilde{y}$s using an LLM to violate requirement-relevant constraints. We collect all tests into a test suite $\mathcal{T}$ and annotate each test $t \in \mathcal{T}$ with its target requirement(s) via a traceability map $\pi : \mathcal{T} \to 2^{\mathcal{R}}$, enabling requirement-level attribution of failing validations during refinement. For instance, in Figure 1, test$\langle[1, 2], 2\rangle$ maps to AR3, enabling failure attribution.

Validating specifications using tests. Given a candidate specification S and a test suite $\mathcal{T}$, we must determine whether S correctly enforces each requirement. Following VERINA (Ye et al., 2025), we construct a Lean proposition for each test $t \in \mathcal{T}$ by instantiating S with the concrete values from t, and attempt to prove or disprove it using proof automation such as the `grind` tactic (Lean Community, 2026). We use proof automation rather than execution because specifications are logical predicates, not executable functions, and the prover can check whether the proposition is logically entailed by the specification.

For a candidate specification S, we define three propositions ϕ^+, $\phi^{\text{in}-}$, and $\phi^{\text{out}-}$ based on test type:

- **Positive test** $\phi^+(\vec{x}, y)$: $\phi^+(S, t) := \mathsf{pre}(\vec{x}) \wedge \mathsf{post}(\vec{x}, y)$, which checks that the specification accepts an intended input-output pair.
- **Negative-input test** $\vec{x}$: $\phi^{\text{in}-}(S, t) := \neg\mathsf{pre}(\vec{x})$, which checks that the precondition rejects an invalid input.
- **Negative-output test** $(\vec{x}, \tilde{y})$: $\phi^{\text{out}-}(S, t) := \mathsf{pre}(\vec{x}) \wedge \neg\mathsf{post}(\vec{x}, \tilde{y})$, which checks that the postcondition rejects an incorrect output for a valid input.

We define t as a `pass` on S if Lean automation can prove the instantiated proposition $\phi(S, t)$, and a `fail` if Lean proof automation proves its negation. If automation is inconclusive, we invoke an LLM judge *only for that test* to label the concrete test as `pass` or `fail`. The judge is used purely to guide refinement in case proof automation fails; after each update we re-run Lean on the full suite. Our analysis in Section B.3 shows 88.4% of validations are resolved by Lean automation in practice.

We summarize the validation results using an evaluation function $\text{EVAL}(S, \mathcal{T})$ that partitions tests by their outcomes:

$$\text{EVAL}(S, \mathcal{T}) := \langle \mathcal{T}^{\text{Lean}}_{\text{PASS}}(S),\ \mathcal{T}^{\text{Lean}}_{\text{FAIL}}(S),\ \mathcal{T}^{\text{Judge}}_{\text{PASS}}(S),\ \mathcal{T}^{\text{Judge}}_{\text{FAIL}}(S) \rangle \tag{1}$$

where $\mathcal{T}^{\text{Lean}}_{\text{PASS}}(S)$ and $\mathcal{T}^{\text{Lean}}_{\text{FAIL}}(S)$ are tests whose corresponding propositions are proved true or false by Lean. $\mathcal{T}^{\text{Judge}}_{\text{PASS}}(S)$ and $\mathcal{T}^{\text{Judge}}_{\text{FAIL}}(S)$ are the remaining tests labeled by the judge due to proof automation being inconclusive.

For refinement, we use the failing tests

$$\mathcal{F}(S) := \mathcal{T}^{\text{Lean}}_{\text{FAIL}}(S) \cup \mathcal{T}^{\text{Judge}}_{\text{FAIL}}(S) \tag{2}$$

and lift them to linked requirements via traceability map π:

$$\mathcal{R}_{\text{FAIL}}(S) := \bigcup_{t \in \mathcal{F}(S)} \pi(t) \tag{3}$$

$\mathcal{R}_{\text{FAIL}}(S)$ identifies which natural-language requirements are implicated by validation failures, enabling requirement-localized clause repair in Section 3.3.

3.3. Specification Synthesis and Requirement-Localized Repair

Iterative refinement via requirement-localized feedback. We first generate an initial specification S_0 from the atomic requirement set $\mathcal{R}$ by prompting an LLM to translate $\mathcal{R}$ into Lean 4 preconditions and postconditions. The LLM is prompted to structure the specification such that individual requirements correspond to separate clauses. Given candidate specification S_k at iteration k, we evaluate it on test suite $\mathcal{T}$ (Equation (1)) to obtain failing tests $\mathcal{F}(S_k)$ (Equation (2)) and implicated requirements $\mathcal{R}_{\text{FAIL}}(S_k)$ (Equation (3)). The traceability map π attributes each test failure to specific requirements, enabling targeted feedback.

We then construct a repair prompt containing the current specification S_k, the implicated failing requirements, and their corresponding representative failing tests. The LLM generates an updated specification S_{k+1} that addresses the implicated requirements while preserving correct aspects of S_k. Iteration continues until all tests pass or a maximum iteration budget is exhausted. Empirically, this localized repair process shows requirement-level generalization rather than test-specific patching, with most successful synthesis converging within a single iteration (see Section B.2).

Adversarial testing. When a candidate specification S_k passes all tests in $\mathcal{T}$ (i.e., $\mathcal{F}(S_k) = \emptyset$), we further apply adversarial testing to detect incorrect or under-constrained specifications. We prompt an LLM to generate adversarial test cases in all three test types based on the atomic requirements $\mathcal{R}$ that are designed to break the current formal specification S_k. These adversarial tests aim to expose edge cases, missing constraints, or overly permissive specifications that allow unintended behavior. For each generated adversarial test t_{adv}, we validate S_k on it using the same validation procedure as in Section 3.2. If the adversarial test successfully breaks the specification (i.e., S_k incorrectly accepts or rejects the test), we add t_{adv} to the test suite $\mathcal{T}$ and use an LLM to map it to its target requirements via the traceability map π. The newly discovered failures trigger additional refinement iterations, returning to the requirement-localized repair process with the expanded test suite.

3.4. Trajectory Recording

We record the complete synthesis trajectory:

$$\tau = (P, \sigma, \mathcal{R}, \mathcal{T}, \pi, S_0, \mathcal{F}_0, \Delta_0, S_1, \mathcal{F}_1, \Delta_1, \ldots, S_K) \quad (4)$$

where Δ_k denotes the localized feedback provided at iteration k containing the implicated requirements and their associated failing tests. These trajectories provide process-level supervision that bridges informal intent and formal contracts through intermediate reasoning steps. Unlike end-to-end specification examples, trajectories expose how to achieve intent alignment: the decomposition from P to $\mathcal{R}$ demonstrates requirement extraction, the traceability map π provides requirement-to-test alignment, and the repair sequence $\langle S_k, \mathcal{F}(S_k), \Delta_k \rangle$ shows how to perform localized repair. We leverage these trajectories for fine-tuning data construction detailed in Section 4.

4. Training Data Construction via Trajectory Distillation

High-quality training data for specification synthesis is scarce. Formal specifications rarely exist in real-world codebases, and writing intent-aligned contracts remains expensive even for experts. To improve model performance beyond inference-time techniques, we need scalable methods to synthetically generate specification synthesis data.

Prior work treats specification generation as direct natural-language-to-specification translation (Endres et al., 2024; Cao et al., 2025), providing problem descriptions paired with ground-truth contract labels. While this establishes correctness targets, it offers no supervision for *how* to achieve intent alignment. These datasets lack guidance on which requirements to extract, how to validate alignment, or how to repair mis-specifications when validation fails.

VERISPECGEN's agentic workflow naturally produces this missing supervision. Each synthesis trajectory contains intermediate reasoning steps such as requirement decompositions, traceability mappings, and failure attributions that bridge informal intent and formal contracts. We leverage this structure through *agentic trajectory distillation* (Zelikman et al., 2022; Chen et al., 2023a; Shinn et al., 2023), executing VERISPECGEN on Python problems from TACO-Verified to construct supervised training data. Each trajectory is distilled into task-specific training examples for decomposition, validation, attribution, and targeted repair. This provides process-level supervision beyond end-to-end examples. We use Claude Sonnet 4.5 as the teacher model to ensure dataset quality, given its adequate performance on VERINA SpecGen task. Total API cost for trajectory collection and distillation was approximately $7,000. See Section A for detailed statistics of dataset construction pipeline

Source data and Lean translation. We source problems from TACO-Verified (Li, 2024), an MIT-licensed collection of Python competitive programming problems with reference solutions and tests. We verify *zero* 10-gram overlap with VERINA to prevent evaluation leakage. For each problem, we use the teacher model to generate the Lean signature from its Python reference solution and construct a literal translator to convert Python test values to type-correct Lean literals. We successfully process 11,981 of 12,901 problems.

Trajectory collection. We execute VERISPECGEN on each translated problem, using the problem description and Lean signature as inputs. After decomposing requirements, we reuse the translated Python test suites as positive tests and map them to atomic requirements using the teacher model. We prune tests based on requirement coverage to control test suite size. Following VERISPECGEN's workflow, we then construct negative tests and execute specification refinement for up to 10 iterations per problem. Each execution produces a comprehensive test suite including adversarial tests, a complete refinement trace, and a final specification when synthesis succeeds. Out of 11,981 problems, we successfully synthesize 6,842 specifications.

SFT dataset distillation. We distill each trajectory checkpoint from the 6,842 successful synthesis runs into instruction-response pairs aligned with VERISPECGEN's modular components. We define 11 tasks spanning the full synthesis pipeline, including requirement decomposition, test generation and mapping, specification generation, validation, failure attribution, and localized repair. Each training example contains instructions based on its task with the teacher model's output as the target. For instance, repair examples include failing requirements, failing tests, and the current specification. We further construct three SFT dataset variants. *SFT Full* contains 343,827 examples spanning all tasks. *SFT No-Test* contains 121,531 examples, excluding

test construction and mapping tasks. *SFT Spec-Only* contains 6,842 end-to-end NL-to-specification pairs.

5. Evaluation

To validate our approach, we measure both inference-time gains from traceable refinement and whether the refinement process itself can be distilled into effective training supervision by investigating three research questions:

RQ1 Does traceable refinement improve specification synthesis compared to direct generation? We compare VERISPECGEN against baselines across multiple model scales. (Section 5.1)

RQ2 Which components of traceable refinement are necessary for effective refinement? We isolate the contribution of requirement decomposition, traceability mapping, and adversarial testing through systematic ablations. (Section 5.2)

RQ3 Can refinement trajectories be distilled into training data that improves base model capabilities? We investigate whether structured trajectories provide richer supervision than end-to-end specification examples, enabling smaller models to internalize refinement strategies and generalize to out-of-domain tasks. (Section 5.3)

General setup. For all experiments, we use Lean v4.24.0 with the `grind` tactic for proof automation (120s timeout). For evaluation on VERINA (Ye et al., 2025), we use the official harness and report both CodeGen and SpecGen pass@1 scores. Unless otherwise specified, we use temperature $T = 0.3$ and a maximum of 10,000 tokens for all LLM generation. All inference and training are performed on AWS p5en.24xlarge instances with 8×H200 GPUs, 192 CPU cores, and 2TB memory.

5.1. RQ1: Does Traceable Refinement Improve Specification Synthesis?

Setup. We evaluate on the VERINA benchmark SpecGen task and compare with the official baseline. The baseline uses 2-shot prompting and measures pass@1 averaged over 5 runs. For VERISPECGEN, we only use the problem description and Lean signature provided by VERINA, and generate Python program from them using the evaluated LLM to obtain the test oracle. The VERINA test suite is private and not used in VERISPECGEN synthesis in all stages. We run the traceable refinement pipeline with up to 25 iterations and up to 60 tests generated per problems using LLM generated program as the test oracle, terminating early when a specification passes all internal tests and adversarial checks. We evaluate 5 models: two closed-source models (Claude Opus 4.5, Claude Sonnet 4.5) and three open-weight models (Qwen3-Coder-480B-A35B, Qwen3-Coder-30B-A3B, Qwen3-4B-Instruct-2507) to test whether improvements generalize across model families and scales.

Table 1. **Specification synthesis results on VERINA.** VERISPECGEN consistently outperforms VERINA baseline across all models, achieving 42–87% relative improvement.

Model	Baseline	VERISPECGEN	Δ Abs. (Rel.)
CLOSED-SOURCE MODELS			
Claude Opus 4.5	59.0	**86.6**	↑27.6 (+46.8%)
Claude Sonnet 4.5	50.7	72.0	↑21.3 (+42.0%)
OPEN-WEIGHT MODELS			
Qwen3-Coder-480B-A35B	36.7	68.5	↑31.8 (+86.7%)
Qwen3-Coder-30B-A3B	29.3	50.3	↑21.0 (+71.7%)
Qwen3-4B-Instruct-2507	15.8	23.0	↑7.2 (+45.6%)

Table 2. **Ablation study on VERINA.** All variants improve over direct generation on Opus 4.5, and removing the decomposer causes the largest performance drop, showing that decomposition is critical for effective specification synthesis.

	Claude Opus 4.5		Qwen3-Coder-30B-A3B	
Variant	Score	Δ Abs. (Rel.)	Score	Δ Abs. (Rel.)
Direct generation	59.0	–	29.3	–
Direct refinement	72.2	↑13.2 (+22.4%)	27.0	↓2.3 (-7.8%)
Test-driven refinement	73.8	↑14.8 (+25.1%)	20.9	↓8.4 (-28.7%)
w/o adversarial testing	81.0	↑22.0 (+37.3%)	48.9	↑19.6 (+66.9%)
VERISPECGEN	**86.6**	↑27.6 (+46.8%)	**50.3**	↑21.0 (+71.7%)

Findings. Table 1 shows that VERISPECGEN consistently improves specification synthesis across all evaluated models. Claude Opus 4.5 achieves the strongest absolute performance at 86.6% pass@1 (+27.6 points over baseline), while mid-tier open-weight models show largest relative gains, with Qwen3-Coder-480B-A35B improving by 86.7% relative (+31.8 points). These improvements generalize across model families, scales, and training paradigms (general purpose and code-specialized), demonstrating that traceable refinement provides robust benefits regardless of model characteristics.

However, the magnitude of improvement correlates with model capability. Mid-tier models (Qwen3-Coder-480B-A35B, Qwen3-Coder-30B-A3B) gain 21.0–31.8 points, while the smallest model (Qwen3-4B-Instruct-2507) gains by only 7.2 points. This pattern suggests that exploiting requirement-level feedback requires sufficient reasoning capability to trace failures to specific constraints and perform targeted repairs. Nevertheless, even weaker models benefit substantially in relative terms, and Section 5.2 shows that traceable refinement substantially outperforms multiple baselines.

5.2. RQ2: Which Components Enable Effective Refinement?

Setup. To isolate component contributions, we conduct ablations on two models representing different capability levels: Claude Opus 4.5 and Qwen3-Coder-30B-A3B. All variants use the same iteration budget (25 refinements) and

Table 3. **Evaluation of trajectory-distilled training variants on VERINA**. Training on specification synthesis trajectories improves both CodeGen and SpecGen tasks. *SFT No-Test* achieves the best specification performance (+15.6–18.2 SpecGen@1), while *SFT Full* yields the strongest code generation transfer (+18.2–21.5 CodeGen@1). Δ shows both absolute and relative improvements over the base model.

	Qwen3-4B-Instruct-2507				Qwen3-Coder-30B-A3B			
	CodeGen		SpecGen		CodeGen		SpecGen	
Training Variants	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10
Base Model	25.3	28.3	14.7	16.3	39.7	53.5	29.4	41.7
SFT Spec-Only	18.0	26.7	9.5	16.6	41.5	68.4	30.4	50.0
Δ Abs. (Rel.)	↓7.3 (-28.9%)	↓1.6 (-5.7%)	↓5.2 (-35.4%)	↑0.3 (+1.8%)	↑1.8 (+4.5%)	↑14.9 (+27.9%)	↑1.0 (+3.4%)	↑8.3 (+19.9%)
SFT Full	**43.5**	**72.7**	22.8	38.8	**61.2**	79.7	43.5	58.0
Δ Abs. (Rel.)	↑18.2 (+71.9%)	↑44.4 (+156.9%)	↑8.1 (+55.1%)	↑22.5 (+138.0%)	↑21.5 (+54.2%)	↑26.2 (+49.0%)	↑14.1 (+48.0%)	↑16.3 (+39.1%)
SFT No-Test	35.4	66.8	**30.3**	**45.2**	58.7	**84.0**	**47.6**	**63.9**
Δ Abs. (Rel.)	↑10.1 (+39.9%)	↑38.5 (+136.0%)	↑15.6 (+106.1%)	↑28.9 (+177.3%)	↑19.0 (+47.9%)	↑30.5 (+57.0%)	↑18.2 (+61.9%)	↑22.2 (+53.2%)

Table 4. **Transfer to out-of-domain benchmarks for Qwen3-4B-Instruct-2507**. Linear merging (α=0.2) successfully transfers specification-synthesis capabilities to math reasoning and general code generation tasks.

	Math Reasoning			Coding		VERINA	
Model variant	**GSM8K**	**AIME24**	**AIME25**	**HumanEval**	**MBPP**	**CodeGen**	**SpecGen**
Base Model	66.3	60.0	46.7	63.7	75.2	25.3	14.7
Merged (α=0.2)	68.3	**63.3**	50.0	**76.1**	**78.2**	22.9	15.6
Δ Abs. (Rel.)	↑2.0 (+3.0%)	↑3.3 (+5.5%)	↑3.3 (+7.1%)	↑12.4 (+19.5%)	↑3.0 (+4.0%)	↓2.4 (-9.5%)	↑0.9 (+6.1%)
SFT No-Test	**80.7**	40.0	23.3	70.1	70.2	**35.4**	**30.3**
Δ Abs. (Rel.)	↑14.4 (+21.7%)	↓20.0 (-33.3%)	↓23.4 (-50.1%)	↑6.4 (+10.0%)	↓5.0 (-6.6%)	↑10.1 (+39.9%)	↑15.6 (+106.1%)

test generation capacity (60 tests per problem).

Ablations. We compare VERISPECGEN against progressively structured baselines (Table 2). **Direct generation** is the VERINA pass@1 baseline without any refinement. **Direct refinement** adds iterative refinement using only LLM self-judgment: the model generates a specification, judges whether it satisfies the description, and revises for up to 25 iterations without test-based validation. **Test-driven refinement** generates requirement-targeted tests and refines based on test outcomes, but without requirement decomposition or failure-attributed feedback. **w/o adversarial testing** removes the adversarial test component only. **VERISPECGEN** includes all components described in Section 3.

Findings. Table 2 reveals a clear hierarchy of component importance. First, iteration without grounded validation is insufficient or harmful. Direct refinement improves Opus 4.5 by 13.2 points but degrades Qwen3-Coder-30B-A3B by 2.3 points, demonstrating that naive LLM self-judgment can be counterproductive for weaker models. Second, test-driven refinement without requirement attribution substantially underperforms. While achieving 73.8% on Opus 4.5, it only reaches 20.9% on Qwen3-Coder-30B-A3B (worse than both direct refinement and baseline), demonstrating that test-based validation without proper attribution confuses weaker models. The full pipeline outperforms this variant by 12.8 and 29.4 points respectively, indicating that grounded validation alone is insufficient without requirement-level attribution to guide localized repairs. Third, removing adversarial testing reduces performance by 5.6 points on Opus 4.5 and 1.4 points on Qwen3-Coder-30B-A3B. Adversarial testing effectively exposes under-constrained or incorrect specifications that pass requirement-targeted tests but miss edge cases, and this capability benefits more from stronger models' ability to generate diverse challenging cases. In summary, effective refinement requires both grounded validation using test and requirement-level attribution. The full VERISPECGEN pipeline achieves consistent improvements across model scales.

5.3. RQ3: Can Trajectory Distillation Improve Models?

Setup. We fine-tune Qwen3-4B-Instruct-2507 and Qwen3-Coder-30B-A3B on variants of our dataset (Section 4) and evaluate the resulting models. We assess improvement in two settings: in-domain performance on VERINA CodeGen and SpecGen tasks, and out-of-domain transfer to general math and coding benchmarks.

5.3.1. IN-DOMAIN IMPROVEMENT: LEAN CODE AND SPECIFICATION SYNTHESIS

Training settings. We compare three variants derived from the same trajectory source. *SFT Spec-Only* uses only final specifications as targets, representing standard end-to-end NL-to-specification supervision as previous work suggested (Cao et al., 2025). *SFT Full* uses complete trajectory-distilled supervision spanning all tasks defined in Section 4. *SFT No-Test* excludes test-related tasks from *SFT Full*. We train for 4 epochs on Qwen3-4B-Instruct-2507 and 5 epochs on Qwen3-Coder-30B-A3B, using a learning rate of 2×10^{-6} with a cosine schedule. We evaluate fine-tuned models using VERINA direct generation baselines on CodeGen and SpecGen tasks and report both pass@1 and pass@10, where pass@10 measures the model's ability to generate diverse candidates for downstream best-of-N se-

lection (Gui et al., 2024) or reinforcement learning (Setlur et al., 2025).

Findings. Table 3 shows that trajectory distillation substantially improves both tasks. Notably, all models are trained on Lean specification synthesis trajectories only, yet we observe significant transfer gains to Lean code generation, demonstrating that reasoning about specifications benefits code synthesis as well. *SFT No-Test* achieves the best SpecGen performance, improving SpecGen pass@1 by 15.6–18.2 points across models, with particularly strong SpecGen@10 gains (+177% on Qwen3-4B-Instruct-2507), indicating improved candidate distribution beyond greedy quality. *SFT Full* achieves the best CodeGen performance, improving CodeGen pass@1 by 18.2–21.5 points.

The supervision variant analysis reveals clear patterns. First, *SFT Spec-Only* fails despite using correct specifications as targets, providing minimal gains on Qwen3-Coder-30B-A3B and degrading Qwen3-4B-Instruct-2507. This demonstrates that end-to-end supervision is likely insufficient without the intermediate reasoning steps. Second, comparing *SFT Full* and *SFT No-Test* reveals that test-construction supervision primarily benefits code generation while specification synthesis benefits from focused training on decomposition and repair. These results validate trajectory distillation as an effective approach, with process-level supervision substantially outperforming end-to-end examples and providing strong transfer to related tasks.

5.3.2. Out-of-Domain: Transfer to General Benchmarks

Training settings. To assess transfer to out-of-domain tasks, we evaluate the Qwen3-4B-Instruct-2507 base model and the *SFT No-Test* checkpoint from Section 5.3 on general math and coding benchmarks: GSM8K (Cobbe et al., 2021), AIME24 (Zhang & Math-AI, 2024), AIME25 (Zhang & Math-AI, 2025), HumanEval (Chen et al., 2021), and MBPP (Austin et al., 2021). To understand how trajectory data performs when mixed with general-purpose training, we apply linear model merging using merge-kit (Goddard et al., 2024), which interpolates parameters between the base and fine-tuned models. We use an interpolation weight of α=0.2 for the fine-tuned model, simulating training on a mixture of general and specification-synthesis data.

Findings. Table 4 shows that trajectory-distilled training transfers to out-of-domain tasks. The linear merged model (with α=0.2) achieves consistent improvements across all out-of-domain benchmarks, with particularly strong gains on HumanEval, demonstrating that formal reasoning data benefits both math reasoning and code generation through improved structured problem-solving. The full fine-tuned model on *SFT No-Test* shows signs of overfitting to the specification-synthesis domain as expected. While it strongly improves GSM8K, likely due to structured reasoning patterns transferring to step-by-step word problems, it degrades on competition mathematics.

We note that linear merging does not work reliably for Qwen3-Coder-30B-A3B due to its Mixture-of-Experts (MoE) architecture (Zhou et al., 2025), and we leave MoE-specific merging strategies to future work.

6. Conclusion and Discussion

We introduced VeriSpecGen, a traceable refinement framework that synthesizes intent-aligned formal specifications through requirement-level attribution and localized repair, achieving 86.6% pass@1 on Verina and establishing state-of-the-art results. Ablation studies confirm that requirement decomposition and localized repair are critical for effective refinement. Beyond inference-time gains, distilling VeriSpecGen's trajectories into 343K training data improves base model specification synthesis by 62–106% while transferring to out-of-domain reasoning tasks, demonstrating that process-level supervision from traceable refinement teaches generalizable problem-solving strategies.

Limitations and future directions. While VeriSpecGen advances state-of-the-art specification synthesis, several limitations direct the future work. First, the framework requires a test oracle for output labeling. Exploring alternative oracle mechanisms like LLM-based prediction or human-in-the-loop feedback could generalize to settings when reference implementations are unavailable while maintaining intent-anchored validation. Second, validation falls back to an LLM judge when proof automation is inconclusive. Advances in proof automation and emerging LLM-based provers can help reduce reliance on LLM judge for complex specifications, enabling more robust specification evaluation. Third, extending our methodology beyond Lean 4 to verification frameworks like Dafny (Leino, 2010) or Coq (Barras et al., 1997) could show the generality of test-driven specification synthesis and broaden its impact across the formal methods community.

References

Aggarwal, P., Parno, B., and Welleck, S. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. *arXiv preprint arXiv:2412.06176*, 2024.

Anthropic. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, 2025a.

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, 2025b.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

Cao, J., Lu, Y., Li, M., Ma, H., Li, H., He, M., Wen, C., Sun, L., Zhang, H., Qin, S., et al. From informal to formal–incorporating and evaluating LLMs on natural language requirements to verifiable formal proofs. *arXiv preprint arXiv:2501.16207*, 2025.

Chen, B., Shu, C., Shareghi, E., Collier, N., Narasimhan, K., and Yao, S. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023b.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., and Trippel, C. nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification (CAV)*, 2023.

de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. The Lean theorem prover (system description). In *International Conference on Automated Deduction (CADE)*, 2015.

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

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 2007.

Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., and Steiner, W. Arsenal: automatic requirements specification extraction from natural language. In *NASA Formal Methods Symposium*, pp. 41–46. Springer, 2016.

Goddard, C., Siriwardhana, S., Ehghaghi, M., Meyers, L., Karpukhin, V., Benedict, B., McQuade, M., and Solawetz, J. Arcee's MergeKit: A toolkit for merging large language models. In Dernoncourt, F., Preoţiuc-Pietro, D., and Shimorina, A. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 477–485, Miami, Florida, US, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-industry.36. URL https://aclanthology.org/2024.emnlp-industry.36.

Gui, P. G. S., Curi, S., Krause, A., and Levy, K. Y. BOND: Aligning LLMs with best-of-N distillation. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://arxiv.org/abs/2407.14622.

Guo, C., Liu, X., Xie, C., Zhou, A., Zeng, Y., Lin, Z., Song, D., and Li, B. Redcode: Risky code execution and generation benchmark for code agents. *Advances in Neural Information Processing Systems*, 37:106190–106236, 2024.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

Jin, S., Yan, F. Y., Tan, C., Kalia, A., Foukas, X., and Mao, Z. M. Autospec: Automated generation of neural network specifications. *arXiv preprint arXiv:2409.10897*, 2024.

Lean Community. *The `grind` Tactic*. Lean FRO, 2026. URL https://lean-lang.org/doc/reference/latest/The--grind--tactic/. The Lean Language Reference.

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

Li, K. Verified taco problems. https://huggingface.co/datasets/likaixin/TACO-verified, 2024. URL https://huggingface.co/datasets/likaixin/TACO-verified.

Lin, H., Sun, Z., Yang, Y., and Welleck, S. Lean-STaR: Learning to interleave thinking and proving. *arXiv preprint arXiv:2407.10040*, 2024.

Ma, L., Liu, S., Li, Y., Xie, X., and Bu, L. SpecGen: Automated generation of formal program specifications via large language models. In *International Conference on Software Engineering (ICSE)*, 2025.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

Misu, M. R. H., Lopes, C. V., Ma, I., and Noble, J. Towards AI-assisted synthesis of verified Dafny methods. *Proceedings of the ACM on Software Engineering*, 2024.

Mugnier, E., Gonzalez, E. A., Polikarpova, N., Jhala, R., and Zhou, Y. Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM on Programming Languages*, 9(POPL), 2025.

Mukherjee, P. and Delaware, B. Towards automated verification of llm-synthesized c programs. *arXiv preprint arXiv:2410.14835*, 2024.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Setlur, A., Geng, C., Hu, X., Levine, S., and Kumar, A. Inference-aware fine-tuning for best-of-N sampling in large language models. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2412.15287.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop verifiable code generation. In *International Symposium on AI Verification*, 2024.

Sun, C., Agashe, V., Chakraborty, S., Taneja, J., Barrett, C., Dill, D., Qiu, X., and Lahiri, S. K. Classinvgen: Class invariant synthesis using large language models. In *International Symposium on AI Verification*, pp. 64–96. Springer, 2025.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., and Chaudhuri, S. Clever: A curated benchmark for formally verified code generation. *arXiv preprint arXiv:2505.13938*, 2025.

Wang, Z., Zhou, Z., Song, D., Huang, Y., Chen, S., Ma, L., and Zhang, T. Towards Understanding the Characteristics of Code Generation Errors Made by Large Language Models . In *International Conference on Software Engineering (ICSE)*, 2025.

Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.-C., and Tian, C. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification (CAV)*, 2024.

Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

Xiao, S., Liu, Z., Zhang, P., and Xing, X. Lm-cocktail: Resilient tuning of language models via model merging. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2474–2488, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, Y., Nie, Y., Wang, Z., Tang, Y., Guo, W., Li, B., and Song, D. SecCodePLT: A unified platform for evaluating the security of code GenAI. *arXiv preprint arXiv:2410.11096*, 2024.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Zelikman, E., Wu, Y., Mu, J., and Goodman, N. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Zhang, Y. and Math-AI, T. American invitational mathematics examination (aime) 2024, 2024.

Zhang, Y. and Math-AI, T. American invitational mathematics examination (aime) 2025, 2025.

Zhou, Y., Karamanolakis, G., Soto, V., Rumshisky, A., Kulkarni, M., Huang, F., Ai, W., and Lu, J. Mergeme: Model merging techniques for homogeneous and heterogeneous moes. *arXiv preprint arXiv:2502.00997*, 2025.

Table 5. **Dataset statistics from trajectory distillation.** Starting from 12,901 TACO-verified problems (decontaminated with zero 10-gram overlap with VERINA), we successfully translate 11,981 to Lean and synthesize 6,842 specifications, from which we distill three SFT dataset variants.

Dataset	#Instances
Source and Processing	
TACO-verified (decontaminated)	12,901
Translated to Lean with test suites	11,981
Successfully synthesized specifications	6,842
Distilled Training Examples	
SFT Full	343,827
SFT No-Test	121,531
SFT Spec-Only	6,842

Table 6. Task distribution in the full SFT dataset. Test-related tasks comprise 64.6% of the dataset, while core specification tasks account for 35.4%.

Task	Count	%	Description
Core Specification Tasks			
`ar_decomposition`	6,927	2.0	Decompose description into atomic requirements
`direct_specgen`	6,842	2.0	End-to-end specification generation
`spec_refinement`	55,270	16.1	Repair failed specifications with localized feedback
`feedback_generation`	42,330	12.3	Generate requirement-attributed failure analysis
`signature_generation`	10,162	3.0	Generate Lean function signatures
Test-Related Tasks			
`positive_test_gen`	8,054	2.3	Generate positive tests (valid input-output pairs)
`negative_input_test_gen`	7,089	2.1	Generate negative-input tests (invalid inputs)
`negative_output_test_gen`	7,612	2.2	Generate negative-output tests (wrong outputs)
`adversarial_test_gen`	1,759	0.5	Generate adversarial tests for edge cases
`test_mapping`	79,639	23.2	Map tests to atomic requirements
`verdict_unknown`	118,143	34.4	Validate specifications on concrete test cases
Total	**343,827**	**100.0**	

A. Dataset Statistics

A.1. Training Data Construction

As detailed in Section 4 and shown in Table 5, we generate training data by executing VERISPECGEN on TACO-verified problems (Li, 2024) and distilling the resulting trajectories into supervised fine-tuning examples. Starting from 12,901 decontaminated problems with zero 10-gram overlap with VERINA, we successfully translate 11,981 to Lean and synthesize 6,842 specifications.

A.2. Task Distribution

The complete filtered dataset contains 343,827 examples spanning 11 distinct tasks extracted from VERISPECGEN's refinement pipeline. Table 6 shows the distribution of tasks in the full dataset. Tasks fall into two categories: *core specification tasks* that handle requirement decomposition, synthesis, and refinement; and *test-related tasks* that generate and validate requirement-targeted tests.

A.3. Training Variants

To investigate the contribution of different task types, we construct three dataset variants from the same trajectory source. Table 7 summarizes the three variants and their task composition.

SFT Full contains all 343,827 examples spanning the complete traceable refinement pipeline. This variant provides comprehensive supervision for all components of VERISPECGEN, including requirement decomposition, test generation, specification synthesis, validation, and localized repair.

Table 7. Training dataset variants. All variants are derived from the same high-quality checkpoints but differ in which tasks are included.

Variant	Examples	Description
SFT Full	343,827	All 11 tasks, complete refinement pipeline
SFT No-Test	121,531	5 core specification tasks only
SFT Spec-Only	6,842	End-to-end NL-to-specification pairs only

Table 8. Task inclusion and percentage composition across training variants. Percentages show the proportion of each task within its variant. SFT No-Test concentrates supervision on refinement tasks (80.3%), while SFT Spec-Only provides only end-to-end targets.

Task	Full	No-Test	Spec-Only
`direct_specgen`	2.0%	5.6%	100%
`ar_decomposition`	2.0%	5.7%	—
`signature_generation`	3.0%	8.4%	—
`spec_refinement`	16.1%	45.5%	—
`feedback_generation`	12.3%	34.8%	—
`positive_test_gen`	2.3%	—	—
`negative_input_test_gen`	2.1%	—	—
`negative_output_test_gen`	2.2%	—	—
`adversarial_test_gen`	0.5%	—	—
`test_mapping`	23.2%	—	—
`verdict_unknown`	34.4%	—	—
Total Examples	**343,827**	**121,531**	**6,842**

SFT No-Test excludes all six test-related tasks, retaining 121,531 examples focused on core specification synthesis. We remove `positive_test_gen`, `negative_input_test_gen`, `negative_output_test_gen`, `adversarial_test_gen`, `test_mapping`, and `verdict_unknown`. This variant concentrates supervision on requirement decomposition, specification generation, and requirement-attributed refinement.

SFT Spec-Only contains only 6,842 end-to-end examples from the `direct_specgen` task, representing standard natural-language-to-specification supervision without intermediate reasoning steps. This variant serves as a baseline to measure the value of process-level supervision.

A.4. Task Composition by Variant

Table 8 shows which tasks are included in each training variant and their relative proportions. In SFT No-Test, refinement-related tasks (`spec_refinement` and `feedback_generation`) comprise 80.3% of the dataset, providing concentrated supervision for requirement-attributed repair. In contrast, SFT Spec-Only provides only final specification targets without any process-level supervision.

A.5. Key Observations

Our dataset construction reveals several insights about effective training for specification synthesis.

First, test-related tasks dominate the full dataset at 64.6%, yet removing them (SFT No-Test) achieves the best specification synthesis performance (Table 3). This suggests that focused training on core specification tasks is more effective than including test-generation supervision, possibly because test generation is a distinct capability that does not directly transfer to specification synthesis.

Second, end-to-end examples alone are insufficient for learning specification synthesis. Despite containing correct final specifications as targets, SFT Spec-Only substantially underperforms both SFT Full and SFT No-Test. This demonstrates that process-level supervision from intermediate reasoning steps is essential for teaching models how to achieve intent alignment.

Third, refinement supervision is critical for effective specification synthesis. In SFT No-Test, `spec_refinement` and `feedback_generation` comprise 80.3% of examples, providing explicit supervision for requirement-attributed repair. The strong performance of models trained on SFT No-Test validates that learning to perform localized repairs based on requirement-level feedback is a key capability for specification synthesis.

B. Additional Experiment Results

B.1. Refinement Budget Analysis

To characterize the accuracy-cost trade-off of traceable refinement, we evaluate VERISPECGEN and its ablation variants under varying refinement budgets on the VERINA SpecGen task with Claude Opus 4.5. Table 9 reports SpecGen pass@1 at 1, 2, and 5 refinement rounds.

Table 9. **SpecGen pass@1 at increasing refinement budgets** on VERINA with Claude Opus 4.5. VERISPECGEN at a single round (78.9%) already exceeds test-driven refinement at five rounds (73.8%) and direct refinement at five rounds (72.2%), indicating that requirement-level attribution contributes more than additional iterations alone.

Method	1 round	2 rounds	5 rounds
VERISPECGEN (full)	78.9%	82.7%	**86.6%**
Test-driven (no attribution)	71.4%	73.6%	73.8%
Direct refinement	62.6%	69.8%	72.2%

Two observations follow. First, VERISPECGEN at a single refinement round (78.9%) already exceeds both test-driven refinement (73.8%) and direct refinement (72.2%) at five rounds, showing that requirement-level attribution provides a stronger lift than additional iterations without attribution. Second, all variants saturate after five rounds, so the practical refinement budget is bounded; the dominant cost driver is iteration count, which is shared across all variants.

B.2. Repair Generalization and Convergence

We further analyze whether repairs in VERISPECGEN address the underlying logical issue or merely patch the specific failing tests shown in the repair prompt. We assess this on the Claude Opus 4.5 configuration by measuring two complementary properties: (i) whether repairs generalize from shown failing tests to withheld tests targeting the same requirement, and (ii) the distribution of refinement iterations required for successful synthesis.

Repair generalization. Each repair prompt contains up to three representative failing tests per implicated requirement. We define *generalization* as the fraction of withheld tests for the same requirement that pass after repair, indicating whether the repair fixed the underlying logical issue rather than only the shown tests. This metric is measurable only for requirements with more than three failing tests, biasing the population toward harder cases; reported numbers are therefore lower bounds. Table 10 summarizes the results.

Table 10. **Repair generalization on VERINA with Claude Opus 4.5.** 75.8% of repair events show at least partial generalization to withheld tests for the same requirement, evidence that repairs target the underlying logical issue rather than patching only shown tests.

Metric	Value
Mean withheld-test pass rate after repair	36.1%
Repairs with full generalization (100%)	13.6%
Repairs with partial generalization (>0%)	75.8%
Repairs with no generalization	24.2%

Convergence behavior. Table 11 reports the distribution of refinement iterations needed for successful synthesis. 76.2% of problems converge on the first attempt and 92.3% within three iterations. For converged problems, failing requirements are resolved in a mean of 2.3 repair rounds, compared to 21.3 for non-converged problems.

B.3. Lean Validation Coverage

Section 3 introduces an LLM judge fallback for tests where Lean proof automation is inconclusive within the timeout when evaluating specification quality. We characterize how often this fallback fires in practice on the Claude Opus 4.5 configuration. Table 12 reports validation rates aggregated over all test evaluations during synthesis.

The judge handles only the 11.6% of evaluations where `grind` cannot decide a proposition within the 120s timeout, primarily on semantically correct specifications using constructs that exceed proof-automation capability. The per-problem

Table 11. **Refinement convergence on VERINA with Claude Opus 4.5.** The 1-round convergence rate of 76.2% and the gap between mean repair rounds for converged (2.3) and non-converged (21.3) problems support the interpretation of refinement as requirement-level repair rather than test-level patching.

Metric	Value
Successfully converged problems	94.5%
Converged in 1 iteration (first attempt)	76.2%
Converged in 2 iterations	13.8%
Converged in 3 iterations	2.2%
Converged in $\leq$3 iterations	92.3%
Mean repair rounds (converged problems)	2.3
Mean repair rounds (non-converged problems)	21.3

Table 12. **Lean validation coverage on VERINA with Claude Opus 4.5.** 88.4% of test evaluations are formally resolved by Lean proof automation, with the LLM judge fallback handling only the 11.6% residue. The per-problem median of 95.9% indicates that most problems are nearly fully Lean-validated.

Metric	Value
Lean-resolved	88.4%
LLM judge fallback	11.6%
Per-problem Lean rate (mean)	88.5%
Per-problem Lean rate (median)	95.9%
Problems with $\geq$80% Lean validation	78.3%

median of 95.9% indicates that this residue is concentrated on a small set of grind-hostile problems rather than uniformly distributed across the benchmark; 78.3% of problems have at least 80% of their tests resolved by Lean.

B.4. Failure Mode Analysis

To characterize the boundaries of the current approach, we analyze the 13.4% of problems on which Claude Opus 4.5 fails to produce a passing specification. Table 13 reports the distribution of failure categories with their root causes.

Table 13. **Failure mode breakdown on the 13.4% of VERINA problems where the best configuration (Claude Opus 4.5) fails.** 69% of failures stem from Lean code-generation capability (pipeline early stop and compilation errors), while only 4% reflect genuine specification reasoning errors. The dominant bottleneck is base-model Lean fluency on harder problems, not the framework itself.

Failure category	%	Root cause
Pipeline early stop	39	Inability to generate type-checked tests
Lean compilation error	30	Cannot produce valid Lean 4 syntax after exhausting repair budget
Decidability timeout	9	Spec semantically correct but `grind` times out and judge fails
Genuine semantic failure	4	Spec logic error persisting after exhausting repair budget
Other	17	Signature type mismatches, quality-check rejections

The dominant bottleneck is Lean code-generation capability on harder problems: 39% of failures are pipeline early stops where the model cannot produce a type-checked test suite, and 30% are Lean compilation errors persisting after the repair budget is exhausted. Only 4% of failures reflect genuine semantic specification errors, suggesting that improvements to base-model Lean fluency would close most of the remaining gap rather than improvements to the refinement framework itself.

B.5. SFT Ablation Study

We conduct a comprehensive ablation study to understand how training data composition affects specification generation performance. We fine-tune both Qwen3-4B-Instruct-2507 and Qwen3-Coder-30B-A3B for 5 epochs and evaluate specification generation performance across sampling temperatures $T \in \{0.0, 0.3, 0.6, 1.0\}$ on the VERINA SpecGen task.

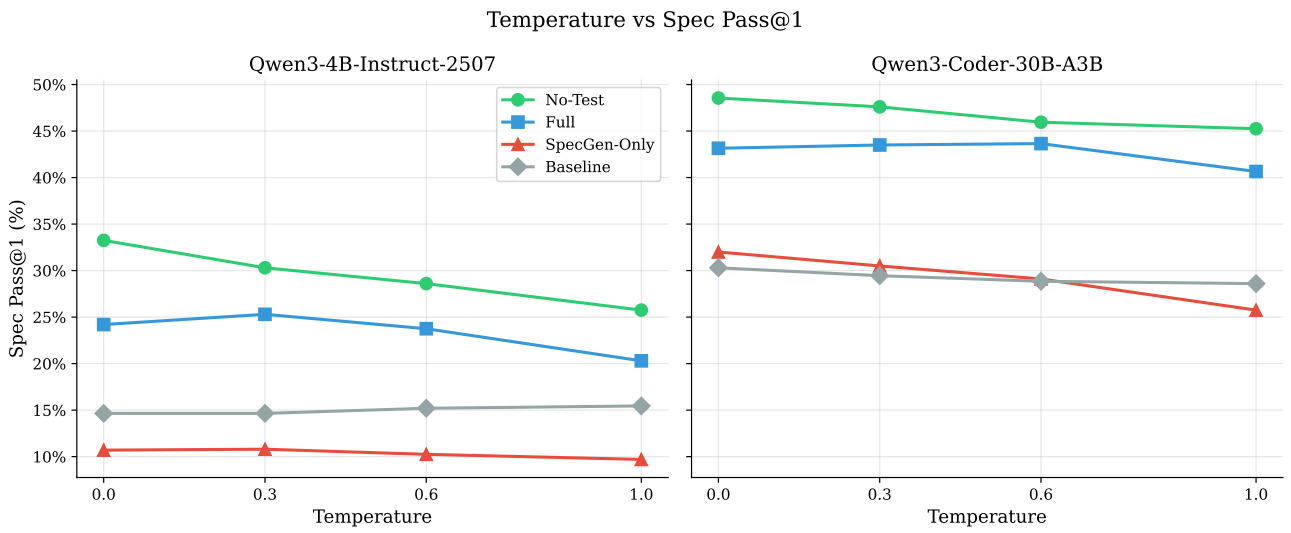

Figure 2. Temperature vs. Spec Pass@1 for different SFT configurations. Lower temperatures consistently yield better performance across all configurations. SFT Spec-Only drops below the base model at higher temperatures for the 30B model, demonstrating that single-task training provides insufficient learning signal and multi-task supervision from trajectory distillation is essential for robust specification synthesis.

B.5.1. Temperature Sensitivity

Figure 2 shows the effect of sampling temperature on specification generation performance across training configurations.

- **Lower temperature yields optimal performance**: Specification generation accuracy decreases monotonically with increasing temperature for all configurations. This pattern suggests that specification synthesis benefits from deterministic decoding, as greedy sampling ($T = 0.0$) consistently achieves the best results. Models remain reasonably robust at moderate temperatures ($T \leq 0.6$), with performance degrading more sharply at $T = 1.0$.

- **SFT No-Test consistently outperforms SFT Full**: Excluding test-related tasks improves specification generation by 2–5% absolute across temperatures. This finding aligns with our observation in Section 5.3.1 that test-construction supervision primarily benefits code generation rather than specification synthesis, suggesting potential interference between test generation and specification generation learning objectives.

- **Multi-task training provides essential learning signal**: SFT Spec-Only, despite being trained specifically on specification generation, performs significantly worse than multi-task configurations. At high temperatures, the 30B model trained on SFT Spec-Only falls below the untrained base model, corroborating the finding in Table 3 that end-to-end supervision without intermediate reasoning steps is insufficient for learning specification synthesis.

B.5.2. Code vs. Specification Trade-off

Figure 3 visualizes the relationship between code generation and specification generation performance across training epochs. Points above the diagonal indicate configurations that achieve relatively stronger specification performance compared to code performance.

- **SFT No-Test achieves optimal specification-code trade-off**: This configuration achieves the highest specification performance while maintaining code generation capability comparable to SFT Full, consistent with the results in Table 3.

- **Larger models exhibit clearer configuration separation**: The 30B model shows more distinct clustering between training configurations, suggesting that increased model capacity amplifies the effects of training data composition. This observation implies that careful curation of training tasks becomes increasingly important at larger scales.

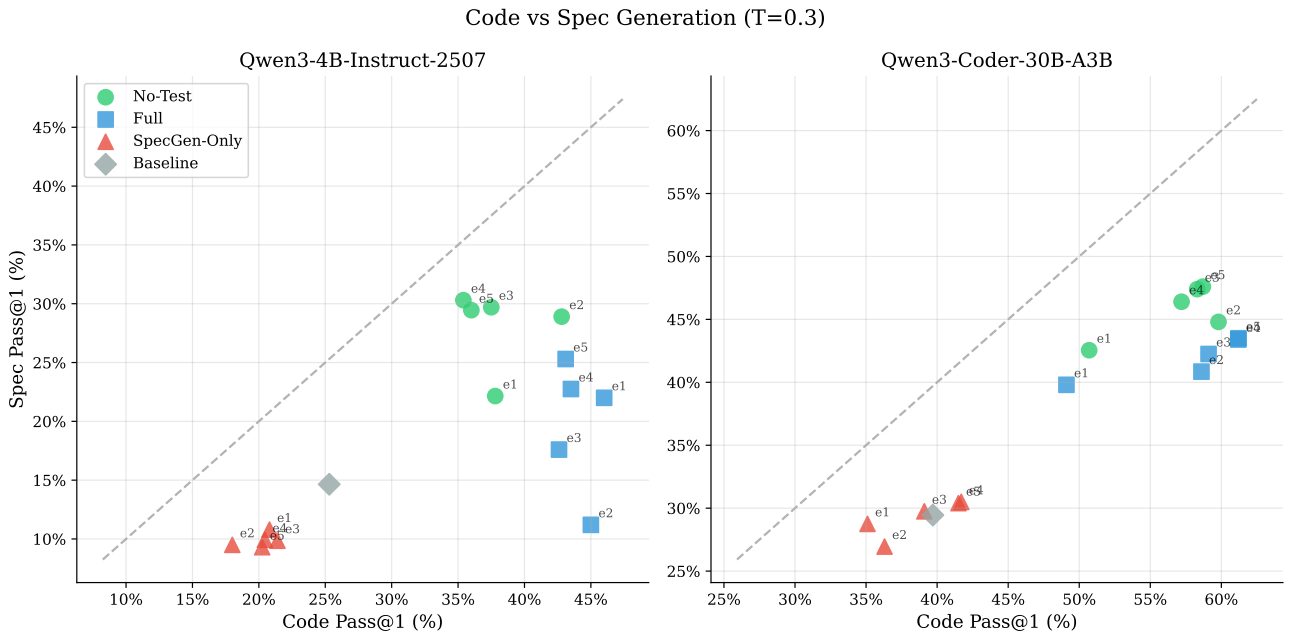

Figure 3. Code Pass@1 vs. Spec Pass@1 at $T = 0.3$. Each point represents a training epoch (e1–e5). SFT No-Test achieves the best specification performance while maintaining competitive code generation capability. SFT Spec-Only clusters in the lower-left quadrant, exhibiting degraded performance on both tasks despite being trained specifically for specification generation.

B.5.3. Training Dynamics Across Epochs and Temperatures

Figure 4 presents a detailed analysis of how specification performance varies across training epochs and sampling temperatures for SFT No-Test, the best-performing configuration identified above.

- **Consistent improvement with training epochs**: Both models show sustained improvement with additional training, though with diminishing returns after epoch 3–4. This suggests that 3–5 epochs provide sufficient exposure to the trajectory-distilled supervision without overfitting in-domain tasks.

B.6. Full Out-of-Domain Transfer Results

To assess whether trajectory-distilled training generalizes beyond formal verification, we evaluate Qwen3-4B-Instruct-2507 on out-of-domain benchmarks as described in Section 5.3. We compare the base model, full fine-tuning on SFT No-Test, and linear model merging (Xiao et al., 2024), which interpolates parameters $\theta_{\text{merged}} = (1-\alpha)\theta_{\text{base}} + \alpha\theta_{\text{ft}}$ to balance specialization with general capabilities. We evaluate three interpolation weights $\alpha \in \{0.1, 0.2, 0.3\}$, where smaller α preserves base model capabilities and larger α incorporates more fine-tuned behavior.

Table 14 shows that linear merging at α=0.2 successfully transfers specification-synthesis capabilities to out-of-domain tasks, improving 6 of 7 benchmarks over the base model with particularly strong gains on HumanEval (+19.5% relative). The interpolation weight α controls the specialization-generalization trade-off: α=0.1 achieves the best AIME25 (53.3) but minimal SpecGen gains (+0.8), while α=0.3 improves SpecGen (+5.6) at the cost of MBPP degradation (-6.6%). Full fine-tuning (SFT No-Test) achieves the strongest in-domain performance but overfits to verification objectives. Interestingly, GSM8K improves substantially (+21.7%) even with full fine-tuning, likely because structured reasoning patterns in specification refinement transfer to step-by-step problem solving. These results demonstrate that α=0.2 merging provides a practical method to incorporate specification-synthesis capabilities into general-purpose models, while full fine-tuning is preferable for specialized verification applications.

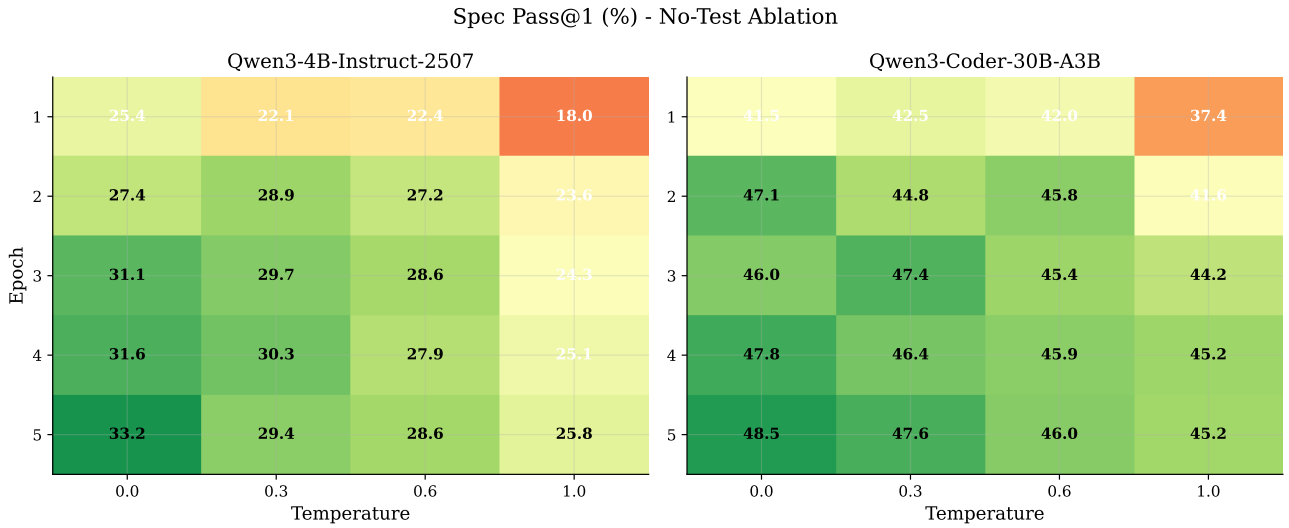

Figure 4. Spec Pass@1 (%) across epochs and temperatures for SFT No-Test. Performance improves consistently with additional training epochs, with diminishing returns after epoch 3–4. At $T = 0.3$, the best configuration achieves 30.3% for the 4B model and 47.6% for the 30B model, representing 15.6 and 18.2 points absolute improvement over the respective base models (14.7% and 29.4%).

Table 14. **Transfer to out-of-domain benchmarks for Qwen3-4B-Instruct-2507**. Linear merging with varying α values transfers specification-synthesis capabilities to math reasoning and general code generation tasks. α=0.2 achieves the best balance across out-of-domain benchmarks.

	Math Reasoning			Coding		VERINA	
Model variant	**GSM8K**	**AIME24**	**AIME25**	**HumanEval**	**MBPP**	**CodeGen**	**SpecGen**
Base Model	66.3	60.0	46.7	63.7	75.2	25.3	14.7
Merged (α=0.1)	67.6	56.7	**53.3**	74.2	76.1	21.4	15.5
Δ Abs. (Rel.)	↑1.3 (+2.0%)	↓3.3 (-5.5%)	↑6.6 (+14.1%)	↑10.5 (+16.5%)	↑0.9 (+1.2%)	↓3.9 (-15.4%)	↑0.8 (+5.4%)
Merged (α=0.2)	68.3	**63.3**	50.0	**76.1**	**78.2**	22.9	15.6
Δ Abs. (Rel.)	↑2.0 (+3.0%)	↑3.3 (+5.5%)	↑3.3 (+7.1%)	↑12.4 (+19.5%)	↑3.0 (+4.0%)	↓2.4 (-9.5%)	↑0.9 (+6.1%)
Merged (α=0.3)	67.1	56.7	46.7	70.1	70.2	28.9	20.3
Δ Abs. (Rel.)	↑0.8 (+1.2%)	↓3.3 (-5.5%)	0.0 (+0.0%)	↑6.4 (+10.0%)	↓5.0 (-6.6%)	↑3.6 (+14.2%)	↑5.6 (+38.1%)
SFT No-Test	**80.7**	40.0	23.3	70.1	70.2	**35.4**	**30.3**
Δ Abs. (Rel.)	↑14.4 (+21.7%)	↓20.0 (-33.3%)	↓23.4 (-50.1%)	↑6.4 (+10.0%)	↓5.0 (-6.6%)	↑10.1 (+39.9%)	↑15.6 (+106.1%)

C. SFT Data Prompts

We include the SFT dataset prompts and construction method from trajectories in the followings:

Prompt 1 (AR Decomposition)

System Prompt

```
You are an expert in Lean 4 formal verification. Your task is to analyze
programming problems and identify the key requirements that a correct Lean 4
implementation must satisfy.

Requirements fall into two categories:
- **Preconditions**: Constraints on the input that must be true before the
function executes (e.g., "the list must not be empty", "n must be positive")
- **Postconditions**: Properties that the output must satisfy after the
function executes (e.g., "the result is the sum of all elements", "the
returned list is sorted")

Each requirement should be atomic, precise, and independently verifiable as
a Lean 4 proposition.
```

User Prompt

```
Analyze the following programming problem and identify all the requirements
(preconditions and postconditions) that a correct implementation must
satisfy.

## Problem Description
{problem_description}

## Function Signature
{signature}

Please identify all preconditions (input constraints) and postconditions
(output properties) for this problem. For each requirement, provide:
- A unique identifier (e.g., "R1", "R2")
- Whether it's a "precondition" or "postcondition"
- A clear description of the requirement
```

Response (from trajectory data)

Response uses `checkpoint.informal_requirements` (the decomposed IRs), formatted as JSON with a natural language introduction stating the count of preconditions and postconditions.

```
I've analyzed the problem and identified {ir_count} requirements
({precond_count} preconditions and {postcond_count} postconditions).

'''json
{requirements_json}
'''
```

Prompt 2 (Test Mapping)

System Prompt

```
You are an expert in Lean 4 formal verification. Your task is to analyze
test cases and determine which requirements they help verify for a Lean 4
specification.

For each requirement, you should assess:
- Whether the test case validates or covers this requirement
- Your confidence level in this assessment
- A brief explanation of your reasoning
```

User Prompt

```
Analyze which requirements this test case validates or covers.

## Requirements
{requirements_json}

## Test Case
Input: {test_input}
Expected Output: {test_output}

For each requirement, determine whether this test case validates it and
explain your reasoning.
```

Response (from trajectory data)

Response uses `checkpoint.testcase_ar_mapping_detailed[tc.id]` which contains per-requirement mappings with `validates`, `confidence`, and `reason` fields for each test case.

```
I'll analyze which requirements this test case validates:

**{req_id}**: This test {status} this requirement ({confidence} confidence).
{reason}

```json
{detailed_mapping}
```
```

Prompt 3 (Positive Test Generation)

System Prompt

```
You are an expert in Lean 4 software testing. Your task is to generate test
cases for programming problems that will be implemented in Lean 4.

**Positive test cases** are tests with valid inputs that satisfy all input
constraints (preconditions), paired with the correct expected output. These
tests verify that a correct Lean 4 implementation produces the right results
for valid inputs.
```

User Prompt

```
Generate positive test cases for the following problem. These should be
valid inputs with correct expected outputs.

## Problem Description
{problem_description}

## Function Signature
{signature}

## Requirements
{requirements_json}

Please generate test cases that cover different scenarios and edge cases.
For each test case, provide:
- The input values
- The expected output
- A brief description of what the test validates
```

Response (from trajectory data)

Response uses `checkpoint.testcases` formatted as JSON objects with `input`, `output`, and `description` fields.

```
I'll generate {test_count} positive test cases covering different scenarios
for this problem.

```json
{testcases_json}
```
```

Prompt 4 (Negative Input Test Generation)

System Prompt

```
You are an expert in Lean 4 software testing. Your task is to generate test
cases for programming problems that will be implemented in Lean 4.

**Negative input test cases** are tests with invalid inputs that violate the
input constraints (preconditions). For example, if a function requires a
non-empty list, a negative input test would pass an empty list. These tests
help verify that a Lean 4 specification correctly identifies invalid inputs.
```

User Prompt

```
Generate negative input test cases for the following problem. These should
have invalid inputs that violate the input constraints.

## Problem Description
{problem_description}

## Function Signature
{signature}

## Requirements
{requirements_json}

Please generate test cases with invalid inputs that violate the
preconditions (input constraints). For each test case, provide:
- The invalid input values
- A description of which constraint is violated and why
```

Response (from trajectory data)

Response uses `checkpoint.negative_input_testcases` formatted as JSON objects with `input` and `description` fields.

```
I'll generate {test_count} negative input test cases that violate the input
constraints.

```json
{testcases_json}
```
```

Prompt 5 (Negative Output Test Generation)

System Prompt

```
You are an expert in Lean 4 software testing. Your task is to generate test
cases for programming problems that will be implemented in Lean 4.

**Negative output test cases** are tests with valid inputs but intentionally
wrong outputs that violate the expected behavior (postconditions). For
example, if the correct output for input [1,2,3] is 6 (the sum), a negative
output test might use output 5 (incorrect sum). These tests help verify that
a Lean 4 specification correctly identifies wrong outputs.
```

User Prompt

```
Generate negative output test cases for the following problem. These should
have valid inputs but intentionally wrong outputs.

## Problem Description
{problem_description}

## Function Signature
{signature}

## Requirements
{requirements_json}

## Reference Test Cases (with correct outputs)
{positive_tests_json}

Please generate test cases by taking valid inputs and pairing them with
wrong outputs. For each test case, provide:
- The input values (valid inputs)
- A wrong output value that violates the expected behavior
- A description explaining why this output is wrong and which requirement it
violates
```

Response (from trajectory data)

Response uses `checkpoint.negative_output_testcases` formatted with `input`, `wrong_output`, `correct_output` (if available), and `description` fields.

```
I'll generate {test_count} negative output test cases with valid inputs but
wrong outputs.

```json
{testcases_json}
```
```

Prompt 6 (Adversarial Test Generation)

System Prompt

```
You are an expert in Lean 4 software testing and verification. Your task is
to generate adversarial test cases designed to find potential bugs in Lean 4
specifications.

Adversarial tests are carefully crafted inputs that target:
- Edge cases and boundary conditions (e.g., empty arrays, maximum values,
zero)
- Corner cases that might be overlooked
- Inputs that might expose logical errors in the specification

These tests help ensure the Lean 4 specification is robust and correctly
handles all valid inputs.
```

User Prompt

```
Generate adversarial test cases to find potential bugs in this
specification.

## Problem Description
{problem_description}

## Function Signature
{signature}

## Requirements
{requirements_json}

## Current Specification
```lean
{spec}
```

Please generate test cases that target edge cases and boundary conditions.
Focus on inputs that might expose specification bugs. For each test, explain
why it might find a bug.
```

Response (from trajectory data)

Response uses `checkpoint.adversarial_testcases` with `input`, `expected_output`, and `description`. Metadata tracks if any test found a bug via `adversarial_failure` flag.

```
I'll generate {test_count} adversarial test cases targeting edge cases and
boundary conditions that might expose specification bugs.

```json
{testcases_json}
```
```

Prompt 7 (Specification Refinement)

System Prompt

```
You are a formal verification expert. Your task is to fix a Lean 4
specification based on the provided feedback.

The feedback describes issues found during evaluation:
- Syntax or type errors from Lean compilation
- Test case failures with specific conditions
- Logic errors in preconditions or postconditions

Generate a corrected specification that addresses all the feedback.
```

User Prompt

```
Fix this Lean 4 specification based on the feedback.

## Problem Description
{problem_description}

## Function Signature
{signature}

## Requirements
{requirements_json}

## Current (Failed) Specification
```lean
{failed_spec}
```

Feedback
{feedback}

Generate a corrected Lean 4 specification with both precondition and
postcondition functions.
```

Response (from trajectory data)

Response uses `checkpoint.spec` (the final successful specification) as the corrected output. The input uses a failed attempt's spec and feedback from `checkpoint.spec_attempts`.

```
Based on the feedback, I'll fix the specification to address the issues.

```lean
{corrected_spec}
```
```

Prompt 8 (Feedback Generation)

System Prompt

```
You are a formal verification expert. Your task is to analyze a failed Lean
4 specification and generate detailed feedback explaining what's wrong and
how to fix it.

Your feedback should:
- Identify the root cause of failures
- Point to specific issues in preconditions or postconditions
- Suggest concrete fixes
- Be actionable and specific
```

User Prompt

```
Analyze this failed specification and generate feedback for improvement.

## Problem Description
{problem_description}

## Requirements
{requirements_json}

## Failed Specification
```lean
{failed_spec}
```

Quality Score
{quality_score:.2f}

Task
Generate detailed feedback explaining what's wrong and how to fix it.
```

Response (from trajectory data)

Response uses `attempt.informalized_feedback` or `attempt.feedback` from `checkpoint.spec_attempts`—the LLM-generated feedback explaining spec failures.

```
[Generated from checkpoint data as described above]
```

Prompt 9 (Signature Generation)

System Prompt

```
You are a Lean 4 programming expert. Your task is to generate a function
signature in JSON format based on the problem description.

The signature should include:
- function_name: A descriptive name for the function
- input_params: A list of input parameters, each with a name and Lean 4 type
- output_type: The Lean 4 return type of the function
```

User Prompt

```
Generate a Lean 4 function signature for the following problem.

## Problem Description
{problem_description}

Output the signature as a JSON object with fields:
- function_name: string
- input_params: array of objects with "name" and "type" fields
- output_type: string
```

Response (from trajectory data)

Response uses `checkpoint.signature` (JSON format) with `function_name`, `input_params`, and `output_type` fields.

```
Based on the problem description, I'll create a Lean 4 function signature.

The function `{func_name}` takes {num_params} parameter(s) and returns
`{output_type}`.

```json
{signature_json}
```
```

Prompt 10 (Verdict Unknown)

System Prompt

```
You are a Lean 4 formal verification expert. Your task is to determine
whether a Lean 4 specification condition will evaluate to TRUE or FALSE for
a given test case.

Analyze the Lean 4 specification logic carefully and determine the
evaluation result.
Provide clear reasoning for your conclusion.
```

User Prompt

```
Determine if this {component} will evaluate to TRUE or FALSE.

## Specification
```lean
{spec}
```

Test Case
Type: {test_type}
Input: {test_input}
Output: {test_output}

Test Type Explanation:
- **positive**: Valid inputs that should produce correct outputs
- **neg_input**: Invalid inputs that violate preconditions (should fail
precondition check)
- **neg_output**: Valid inputs with wrong outputs (should pass precondition
but fail postcondition)

Task
Analyze whether the {component} condition evaluates to TRUE or FALSE for
this test case. Explain your reasoning step by step, then provide your final
answer.
```

Response (from trajectory data)

Response uses `result.llm_reasoning` from `checkpoint.spec_eval.results` where `result_status='unknown'`, combined with the computed TRUE/FALSE verdict.

```
Let me analyze the {component} for this test case.

{reasoning}

**Answer: {eval_result}**
```

Prompt 11 (Direct Specification Generation)

System Prompt

```
Your input fields are:
1. `task_description` (str): The specification task
2. `task_template` (str): Lean 4 code snippet with placeholders
3. `precond_desc` (str): Natural language precondition description
4. `postcond_desc` (str): Natural language postcondition description
Your output fields are:
1. `imports` (str): Required imports (optional)
2. `precond_aux` (str): Auxiliary precondition definitions
3. `precond` (str): Generated precondition code
4. `postcond_aux` (str): Auxiliary postcondition definitions
5. `postcond` (str): Generated postcondition code
All interactions will be structured in the following way, with the
appropriate values filled in.

[[ ## task_description ## ]]
{task_description}
[[ ## task_template ## ]]
{task_template}
[[ ## precond_desc ## ]]
{precond_desc}
[[ ## postcond_desc ## ]]
{postcond_desc}
[[ ## imports ## ]]
{imports}
[[ ## precond_aux ## ]]
{precond_aux}
[[ ## precond ## ]]
{precond}
[[ ## postcond_aux ## ]]
{postcond_aux}
[[ ## postcond ## ]]
{postcond}
[[ ## completed ## ]]

In adhering to this structure, your objective is:
        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 (wrapped with {{}}) for the spec to be generated.
        The precondition should be as permissive as possible, and the
postcondition should model a sound and 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
```

Prompt 12 (Direct Specification Generation (Continued))

User Prompt

```
[[ ## task_description ## ]]
{task_description}

[[ ## task_template ## ]]
{task_template}

[[ ## precond_desc ## ]]
{precond_desc}

[[ ## postcond_desc ## ]]
{postcond_desc}

Respond with the corresponding output fields, starting with the field `[[ ##
imports ## ]]`, then `[[ ## precond_aux ## ]]`, then `[[ ## precond ## ]]`,
then `[[ ## postcond_aux ## ]]`, then `[[ ## postcond ## ]]`, and then
ending with the marker for `[[ ## completed ## ]]`.
```

Response (from trajectory data)

Response uses parsed components from `checkpoint.spec`: `imports`, `precond_aux`, `precond`, `postcond_aux`, `postcond` extracted via `parse_spec_components()`.

```
[[ ## imports ## ]]
{imports}

[[ ## precond_aux ## ]]
{precond_aux}

[[ ## precond ## ]]
{precond}

[[ ## postcond_aux ## ]]
{postcond_aux}

[[ ## postcond ## ]]
{postcond}

[[ ## completed ## ]]
```