AUXSPEC: LEARNING TO AUTOFORMALIZE PROGRAM SPECIFICATIONS VIA EXECUTABLE POSTCONDITIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal specifications define intended program behavior and underpin verification, but writing them in theorem provers such as Lean is slow and expertise-intensive, making specification a major bottleneck to scaling formally correct code. Autoformalization from natural-language problems is promising, yet training specification models requires scalable feedback on whether generated specifications capture intended behavior. Prior approaches obtain this feedback through explicit proofs or separately implemented code with formal correspondence proofs, both expensive. We introduce AUXSPEC, a Lean framework that pairs declarative postconditions with executable counterparts through typeclass resolution. Test cases can then evaluate declarative specifications by execution, letting us use agreement as a reinforcement-learning reward. Using AUXSPEC, we synthesize 3,037 LeetCode problems with proved equivalence theorems and expand training to 19.5K samples through expert iteration on Codeforces. Experiments show that training Qwen3.8-27B and Qwen3.5-9B with the execution-based reward raises SpecGen pass@4 on AUXSPEC-Test 0.7% $\rightarrow$ 60.3% and 0.7% $\rightarrow$ 40.3%. On the out-of-domain Verina benchmark, both models improve 21.9% $\rightarrow$ 84.7% and 7.0% $\rightarrow$ 69.8%, and the 27B model surpasses VeriSpecGen-30B-A3B, the prior state of the art among open-source specification models, on both pass@1 (47.6% $\rightarrow$ 60.2%) and pass@4 (63.9% at pass@10 $\rightarrow$ 84.7%). We release the 3,037 problems with test cases and equivalence proofs, the weights of the trained 9B and 27B models, and the 300-problem AUXSPEC-Test benchmark with expert review and correctness proofs for future evaluation.

1 INTRODUCTION

Formal program specifications define the correct behavior of a program over all valid inputs, providing the reference against which implementations can be proved correct or tested. Yet writing such specifications in an interactive theorem prover requires substantial expertise and effort, making specification a fundamental barrier to scaling formal code verification and generation. As AI models make rapid progress in generating code and proving its correctness, this barrier becomes increasingly important: realizing the promise of formally correct code requires automating specification itself. Autoformalization from natural-language programming problems offers a path forward, but immediately raises a central question: *when does a generated formal specification faithfully capture the intended problem?*

This question is particularly important in modern code-generation pipelines. Large language models (LLMs) generate code for programming problems (e.g., LeetCode) from natural-language descriptions and example input-output pairs, while candidate solutions are evaluated on test cases that provide scalable reward signals for reinforcement learning (Wang et al., 2025a; Ruan et al., 2026). However, test cases only sample program behavior (Xie et al., 2026), allowing incorrect solutions to pass when corner cases are missing (Shi et al., 2026). Formal verification in an interactive theorem prover such as Lean (de Moura et al., 2015; de Moura & Ullrich, 2021), by contrast, establishes that an implementation satisfies its specification on every permitted input, with a proof checked by a small trusted kernel (Hoare, 1969). The specification translates the natural-language problem into a precondition on valid inputs and a postcondition on correct outputs (Endres et al., 2024; Chen et al., 2026). Yet manual curation limits the scale of Clever and Verina to 161 and 189 problems (Thakur et al., 2025; Ye et al., 2026a), while errors persist both in curated references (Feng et al., 2026)

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.

Table 1. Comparison of AUXSPEC with closely related work on specification generation. Specification validation checks whether a specification matches the source problem. [1] $\vee$ indicates that the check falls back to the second method when the first is inconclusive, and $\rightarrow$ that the second method follows the first. PBT denotes property-based testing. The equivalence proof of AUXSPEC shows that the executable postcondition agrees with the postcondition on all valid inputs and outputs. Correctness marks whether every implementation is proved to satisfy the specification (✓), only some are (Partial), or the work has no implementation task (–). For AUXSPEC, the proofs cover the 300 test problems. SFT, EI and RL denote supervised fine-tuning, expert iteration and reinforcement learning.

Work	Lang.	Specification validation		Correctness	Manual review	Training
		Reference	Method[1]			
Verina (Ye et al., 2026a)	Lean	Expert specification	Proof $\vee$ testing	Partial	Benchmark (189)	None
Clever (Thakur et al., 2025)	Lean	Expert specification	Proof	Partial	Benchmark (161)	None
VeriSpecGen (Ye et al., 2026b)	Lean	Test cases	Proof $\vee$ LLM judge	–	Not reported	SFT
Coins (Yang et al., 2026)	Rocq	Test cases	Proof, LLM judge	–	Benchmark (164)	None
ATLAS (Baksys et al., 2025)	Dafny	Test cases	Verifier $\rightarrow$ LLM judge	✓	Not reported	SFT
Verus-SpecGym (Agarwal et al., 2026)	Verus	Test cases	Verifier $\vee$ execution	–	Not reported	None
LeetProof (Feng et al., 2026)	Lean	Test cases	LLM judge $\rightarrow$ PBT	Partial	Unproved (38)	None
AUXSPEC (ours)	Lean	Test cases	Equivalence proof $\rightarrow$ execution	✓	Test set (300)	SFT, EI, RL

and in specifications produced by automated pipelines (Dougherty & Mehta, 2025; Bursuc et al., 2026; Dougherty et al., 2026). Scaling specification therefore requires more than generating formal statements: it requires an efficient, scalable signal for determining whether a generated specification faithfully captures the source problem, analogous to the test-case signals that enable large-scale training for code generation.

Training specification generators has been explored in auto-active verification languages, where a verifier, like an SMT solver, provides the reward by checking candidate specifications on test cases (Yan et al., 2026; Huang et al., 2026; Baksys et al., 2025). However, SMT solvers limit the complexity of specifications that can be checked (Yang et al., 2026; Ye et al., 2026a), and interactive theorem provers are more expressive but provide less automation (Feng et al., 2026), which makes each proof costly. A generated specification in Lean can compile even when incorrect, and prior work checks each test case by proof, a method known as symbolic specification testing (Lahiri, 2024; Ye et al., 2026b; Yang et al., 2026), which leaves test cases without a proof to an LLM judge or unresolved and repeats the proof search for every revised or newly generated specification. The proofs are unavoidable as a declarative postcondition states the intended behavior independently of a checking procedure, and quantification over unbounded domains does not itself provide an executable check. As a result, the test-level signal that trains code generation (Wang et al., 2025a; He et al., 2026) is unavailable for training specification generation at scale (Table 1).

To address this issue, we propose AUXSPEC, a framework that trains specification generation in Lean 4 with executable feedback at the specification level. Each declarative postcondition is paired with an executable counterpart proved equivalent once per problem, and executing the counterpart on test cases both decides the declarative postcondition and provides the reward for reinforcement learning, which brings the test-based reward of code generation to specification generation. We build a pipeline that synthesizes the components in separate stages with test cases labeled by reference solutions, apply the pipeline to LeetCode to obtain 3,037 problems with all equivalence theorems proved, and scale the training data to 19.5K samples with expert iteration and Codeforces problems.

We fine-tune Qwen3.5-9B and Qwen3.8-27B on these samples and apply reinforcement learning with the execution-based reward, obtaining AUXSPEC-9B and AUXSPEC-27B for specification and implementation generation and repair. In performance, the 27B model raises SpecGen pass@4 from 0.7% to 60.3% on AUXSPEC-Test and from 21.9% to 84.7% on Verina without test cases in the prompt, relative to the backbone. The 9B model improves from 0.7% to 40.3% and from 7.0% to 69.8% under the same settings, and reinforcement learning with the execution-based reward raises pass@4 over the fine-tuned models at both scales (Section 5.4).

Our contributions are threefold.

- **Executable specification format.** We introduce a specification format that pairs a declarative postcondition with an executable Boolean postcondition on input-output pairs, obtained through type class resolution in Lean when a decidability instance exists and defined separately otherwise. An equivalence theorem proves that both postconditions agree on valid inputs, while the correctness theorem keeps the specification independent of the implementation (Section 2).
- **AUXSPEC dataset and AUXSPEC-Test.** We release the 3,037 LeetCode problems with specifications, implementations, frozen test cases and equivalence proofs, including AUXSPEC-Test, an

expert-reviewed benchmark of 300 problems for specification generation in Lean with executable postconditions (Sections 3 and 3.3).
- **AUXSPEC-9B-RL and AUXSPEC-27B-RL.** With a reward from test case execution, we train and release open-weight models that generate Lean specifications and implementations from natural-language descriptions. The models are competitive with frontier models in domain and on Verina.

2 PROBLEM FORMULATION

We augment the declarative postcondition of a specification with an executable postcondition, which decides test cases by execution. This section describes the format, tasks and acceptance criteria.

Format. A problem provides a description, input constraints and test cases, and asks for a Lean file with specification, implementation and 2 theorems (Figure 9). Within the specification, the precondition restricts valid inputs, while the postcondition and the executable postcondition describe correct outputs, declaratively and by computation. The equivalence theorem relates the 2 postconditions on valid inputs, and the correctness theorem asserts the implementation satisfies the postcondition. We illustrate the format with 2 synthesized problems, *Count Number of Special Subsequences*, whose executable postcondition enumerates candidate index sets (Appendix E.2.1), and *Continuous Subarray Sum*, whose executable postcondition decides the postcondition through type class resolution (Appendix E.2.1).

We validate the postcondition by executing the test cases on the executable postcondition, and the equivalence theorem transfers the results to the postcondition. Prior work instead proves the declarative postcondition on every instantiated test case (Section 1). For the correctness theorem, we use the declarative postcondition to prevent the specification from restating the implementation, which would reduce the proof to unfolding definitions (Thakur et al., 2025; Zeng et al., 2025).

Tasks. We focus on specification generation (SpecGen) and code generation (CodeGen) as in prior work (Ye et al., 2026a), and add the repair tasks SpecGen-Repair and CodeGen-Repair (Table 2). The synthesis pipeline performs these tasks to build the training data and the test set.

Table 2. Input and output of each task. Broken is the rejected output that a repair task revises, and Diagnostics are the compiler errors and failed test cases of that output. Test layers check the output (Table 7).

Task	Input					Output	
	Description	Constraints	Cases	Broken	Diagnostics	Component	Test layers
SpecGen	✓	✓	✓			Specification	$L1^a$ $L1^b$ $L2^a$ $L2^b$
SpecGen-Repair	✓	✓	✓	✓	✓	Repaired specification	$L1^a$ $L1^b$ $L2^a$ $L2^b$
CodeGen	✓		✓			Implementation	L3
CodeGen-Repair	✓		✓	✓	✓	Repaired implementation	L3

Acceptance criteria. A generated output is accepted only if the assembled file compiles, the test layers of the task pass and LLM judges approve (Zheng et al., 2023). The same criteria apply to data filtering, sample selection in expert iteration, the reward in reinforcement learning and evaluation.

Test cases are fixed before synthesis (frozen) and executed as `#guard` commands, which fail at compilation when the expression is false (Ye et al., 2026a). The commands form layers by the component and the case under test (Table 7). Outputs transcribe the frozen test cases into the layers of the task, and test case alignment checks the transcription line by line. Beyond the test cases, the judges check faithfulness (the postcondition keeps all conditions of the description), non-tautology (the postcondition states a property rather than an algorithm), completeness (the bounds of the executable postcondition keep correct outputs beyond the tested inputs), and termination (the implementation terminates within the stated input bounds).

3 DATA SYNTHESIS

To prevent synthesized components from fitting the test cases or weakening the accepted specification (Baksys et al., 2025), we synthesize the components in isolated and ordered stages (Figure 1, Appendix E). The executable postcondition is synthesized after the postcondition is accepted, and

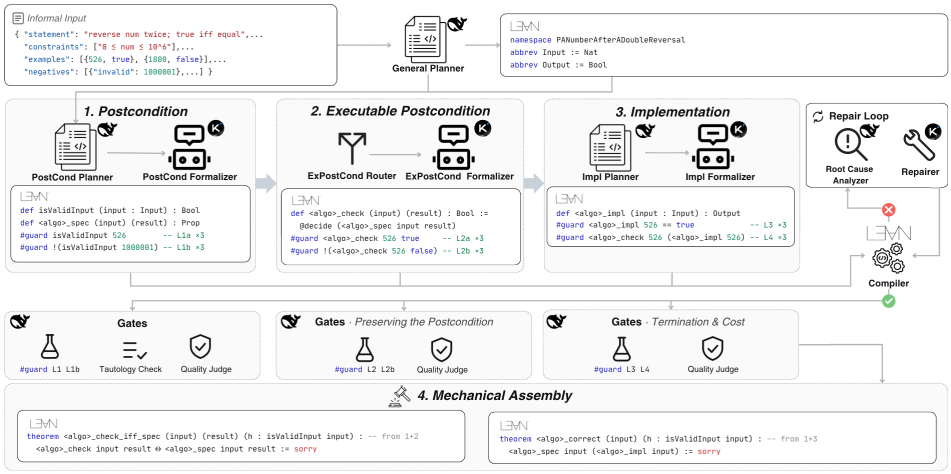


Figure 1. Data synthesis pipeline for AUXSPEC. Stages (1) to (3) synthesize the postcondition, the executable postcondition, and the implementation under separate plans, and stage (4) assembles the components with the theorem statements.

the implementation after both postconditions. We select the models by a pilot comparison of synthesis success and reasoning quality (Appendix E.8).

3.1 SYNTHESIS PIPELINE

Seed data. We collect 3,109 LeetCode problems (Xia et al., 2025) with typed function signatures, reference solutions and stored tests (Table 3).

Table 3. Number of problems and samples for each source. Seeds are the problems retained after filtering.

Source	Seeds	Difficulty			SFT		RL		AUXSPEC-Test
		Easy	Medium	Hard	Problems	Samples	Problems	Samples	Problems
LeetCode	3,109	719	1,622	768	2,701	14,928	63	107	300
Codeforces	3,285	1,281	1,140	864	1,292	4,564	1,986	2,760	0
AtCoder	623	241	322	60	0	0	623	927	0
Total	7,017	2,241	3,084	1,692	3,993	19,492	2,672	3,794	300

Test cases. The reference solution labels expected outputs independently of the specifications under synthesis (Wang et al., 2025a; Fu et al., 2025; He et al., 2026). Positive test cases pair valid inputs from the examples and stored tests with these outputs. Negative test cases are invalid inputs and valid inputs paired with outputs that the reference solution confirms as incorrect (Appendix E.1.1). Both fill the test layers of Table 7 and are shown to the model during generation and repair.

Specifications. To assemble the separately synthesized components, we fix the input and output types from the function signature before any component is generated (Figure 1), and synthesis retrieves declarations from the pinned Mathlib version (The mathlib Community, 2020) to prevent calls to unknown identifiers. The executable postcondition is first set to the decidable form, the postcondition decided through decidability instances, which serves 35.5% of the synthesized problems, and a separate executable postcondition is generated when no instance can be synthesized (Appendix E.2). Judges review each accepted specification for the properties of Section 2, and each rejection includes a counterexample input.

Implementation. Given the accepted specification, we plan the implementation to reuse existing declarations, and quality review targets properties that small test inputs cannot show, such as nontermination hidden behind a fuel parameter and cost beyond the stated input bounds (Appendix E.3.1).

Repair. Components that fail compilation or a test case are repaired from compiler and REPL feedback (Lean community, 2023) for up to 18 rounds. To prevent repair from weakening a specification to fit the test cases, each round rewrites the single block that holds the first error and keeps accepted components and all other blocks fixed, and a root-cause analyzer names the block to rewrite, or the plan for replanning, when an error cannot be placed by line (Figure 1). The analyzer can discard

a frozen test case only when the case contradicts a constraint quoted from the problem, and no test case was discarded (Appendix E.4.1). The recorded repairs supply the repair samples of Section 4.1.

3.2 RESULTS

The pipeline synthesizes 3,037 of the 3,109 problems (97.7%, 99.3% on easy and 95.7% on hard problems, Appendix E.5). Test cases on small inputs admit false positives (Li et al., 2022), and the implementation judge rejects 19.6% of the implementations that pass all test cases (Appendix E.3.1). The decidable form passes all checks at the first attempt for 98.1% of the problems admitting the form, against 52.4% for separately generated executable postconditions (Appendix E.2.2).

Only 34.8% of the seeds pass all components at the first attempt, and repair recovers most others within 2 rounds (Appendix E.4.1). We prove the equivalence theorem for all 3,037 problems. The proof attempts exposed statement errors in 31 problems, which had all passed the test layers and judges and were repaired (Appendix F). For the remaining 99.0% of accepted problems, the theorem is provable as synthesized, which supports applying the acceptance criteria without proofs in training and evaluation. Of the 31 problems, 19 have errors in the postcondition, which execution never evaluates, and 10 in the executable postcondition beyond the tested outputs (Table 31).

3.3 AUXSPEC-TEST

AUXSPEC-Test consists of 300 problems across 4 tasks, with 72 easy, 155 medium and 73 hard problems. Evaluation splices a generated component into the reference file, joining a generated specification with the reference implementation and a generated implementation with the reference specification, and scores the file under the criteria of Section 2. To keep errors in reference components from misjudging generated outputs, every component of the 300 problems is manually checked against the problem description, finding 21 errors (Appendix G.1), and proving the correctness theorem for every problem finds 1 more (Appendix F.3). In total, 278 problems (92.7%) pass the review and are proved without changes, and the 22 with errors are manually revised. Scoring through frozen test cases and judges lets AUXSPEC-Test accept any correct formulation of a specification, including ones other than the reference (Ye et al., 2026a; Thakur et al., 2025).

4 TRAINING

4.1 TRAINING DATA

Each synthesized problem contributes samples to the tasks of Section 2. The generation tasks take accepted components as targets, and the repair tasks take components rejected during synthesis as inputs. Each sample carries a reasoning trace rewritten from the synthesis output, which records the planning behind a generated component and the diagnosis behind a repair (Appendix E.6).

Increasing the number of problems improves SpecGen performance more than continued pretraining on a Mathlib corpus (Appendix D.2). We therefore source further problems from Codeforces and AtCoder through TACO (Li et al., 2023), and Table 3 lists the sources of the supervised fine-tuning (SFT) and reinforcement learning (RL) data. To eliminate the risk of data contamination, the final SFT and RL data are disjoint from each other and from the evaluation benchmarks (Appendix C).

4.2 SUPERVISED FINE-TUNING

We grow the SFT data to 19.5K training samples over 3 rounds of expert iteration (Polu et al., 2023), each fine-tuning Qwen3.8-27B from the base model on the accumulated samples and adding the outputs that meet the acceptance criteria of Section 2. SpecGen performance rises as the data grow, with no sign of saturation so far (Appendix D.1). Fine-tuning Qwen3.5-9B (Qwen Team, 2026a) and Qwen3.8-27B (Qwen Team, 2026b) on these samples yields AUXSPEC-9B-SFT and AUXSPEC-27B-SFT, with the settings of Appendix B.

4.3 REINFORCEMENT LEARNING

We further apply RL to the SFT models on 2,672 problems disjoint from the SFT data (Table 3), yielding AUXSPEC-9B-RL and AUXSPEC-27B-RL. The recipe is developed on the 9B model and applied to the 27B model for 1 epoch (Appendix D.3).

Reward design. A binary reward requiring all acceptance criteria has low variance on SpecGen, where the SFT models pass 11.8% and 23.2% at pass@1 (Section 5.2), and a low-variance reward would slow optimization even when accurate (Razin et al., 2025b). We therefore score each rollout with a 6-stage progressive reward that follows the order of the acceptance criteria of Section 2 and credits each stage passed (Le et al., 2022; Huang et al., 2026), namely format (0.1), compilation (0.1), test execution (0.1), test case alignment (0.3), non-tautology (0.2) and overall quality (0.2). Stages count only if all earlier stages pass, and the LLM-judged stages come last. With $c_k(y) = 1$ if rollout y passes stage k and 0 otherwise, and Δ_k the value of stage k, the reward is

$$r(y) = \sum_{k=1}^{6} \Delta_k \prod_{j=1}^{k} c_j(y). \tag{1}$$

To prevent reward hacking, only tests that call a declaration of the output count toward the required tests, and rollouts with fewer such tests earn only Δ_1. Outputs that assume unproved facts or bypass the kernel check count as compilation failures, and unchanged repair rollouts receive 0 (Appendix D.3.1).

Self-repair. Besides generation, we train on repair tasks built from failed rollouts of the current policy, which avoids distribution shift (Kumar et al., 2025). We keep these rollouts, paired with compiler messages, in a buffer that fills 40% of each batch with single-turn repair tasks (Table 2). Failed rollouts return to the buffer at most twice, and each return scales the reward down by 0.1.

Policy optimization. Our RL algorithm extends Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO; Yu et al., 2025) by normalizing the advantage by the standard deviation over the batch instead of the group (Hu et al., 2025), adding a KL penalty against the SFT model π_{ref} (Liu et al., 2025), which delays collapse (Section 5.4), and masking the clipped surrogate $\ell_{i,t}$ of token t in rollout i with $m_{i,t} \in \{0, 1\}$. Combining these changes, we maximize

$$\mathcal{J}(\theta) = \mathbb{E}\Big[\frac{\sum_{i,t} m_{i,t}\, \ell_{i,t}(\theta)}{\sum_{i,t} m_{i,t}}\Big] - \beta\, \mathbb{D}_{\text{KL}}\big[\pi_\theta \,\|\, \pi_{\text{ref}}\big], \tag{2}$$

which reduces to the original algorithm with $m_{i,t} = 1$, $\beta = 0$ and group normalization. The mask $m_{i,t}$ is 0 on all tokens of repetitive rollouts and 1 otherwise, which removes the gradient of such rollouts but keeps the reward in the group mean. A rollout is classified as repetitive if a 40-character n-gram occurs at least 20 times and over 40% of n-grams are repeats (Welleck et al., 2020). We set $\beta = 0.04$ (Shao et al., 2024) and list the remaining hyperparameters in Appendix B.

5 EXPERIMENTS

5.1 SETUP

Training settings, including hyperparameters and the Lean environment, are listed in Appendix B, and the sampling settings in Appendix B.2. We report pass@k (Chen et al., 2021) for $k \in \{1, 4\}$, the fraction of problems with at least one of k samples meeting the acceptance criteria of Section 2 (Appendix B.3), and the evaluated models use the same prompts (Appendix I). The baselines are our backbones, frontier models and prior work on specification generation.

- **Instruct Models:** To isolate the effect of fine-tuning, we evaluate the backbones Qwen3.5-9B (Qwen Team, 2026a) and Qwen3.8-27B (Qwen Team, 2026b).
- **Frontier Models:** Claude-Opus-5 (Anthropic, 2026a), Claude-Sonnet-5 (Anthropic, 2026b), GPT-5.6-Sol and GPT-5.6-Luna (OpenAI, 2026), Kimi-K3 (Kimi Team, 2026), and DeepSeek-V4-Pro and DeepSeek-V4-Flash (DeepSeek-AI, 2026).
- **Prior Work:** We compare against VeriSpecGen (Ye et al., 2026b), which fine-tunes Qwen3-4B and Qwen3-Coder-30B-A3B on refinement trajectories distilled from a closed teacher, using the reported results as the models are not publicly available.

5.2 MAIN RESULTS

We report the performance on the in-domain test set in Table 4. Training raises SpecGen pass@4 from 0.7% for both backbones to 40.3% (9B RL) and 60.3% (27B RL). The resulting models are competitive with large-scale frontier models, as AUXSPEC-27B-RL trails only Claude-Opus-5 and Kimi-K3 in SpecGen pass@1 and the 9B RL model outperforms Claude-Sonnet-5 and GPT-5.6-Luna. On both repair tasks, the 27B RL model comes within 3 points of the best frontier model at pass@4. To assess false positives in the automatic SpecGen evaluation, we manually review one accepted output for each of 15 problems counted as pass@4 successes per model, with the 9B RL model audited at step 120, and the review confirms 55 of the 60 outputs as faithful (Appendix G.2).

Table 4. Main results on AUXSPEC-Test. Underline marks the best result within each group, and **bold underline** the best overall. Green numbers in parentheses give the gain in points over the backbone of the same size.

Model	SpecGen		CodeGen		SpecGen-Repair		CodeGen-Repair	
	pass@1	pass@4	pass@1	pass@4	pass@1	pass@4	pass@1	pass@4
Frontier Models								
Claude-Opus-5	**52.5**	73.7	64.0	82.7	**89.3**	**96.9**	**73.5**	**86.4**
Claude-Sonnet-5	13.0	36.7	30.4	53.0	72.3	86.3	69.2	80.6
GPT-5.6-Sol	28.1	55.0	**72.8**	**91.7**	66.2	82.4	72.8	80.6
GPT-5.6-Luna	8.9	26.7	5.7	19.0	54.2	74.8	63.1	76.7
Kimi-K3	42.2	**81.3**	6.8	23.0	80.7	95.4	70.6	**86.4**
DeepSeek-V4-Pro	20.4	53.0	31.7	60.7	63.4	90.8	62.1	83.5
DeepSeek-V4-Flash	33.0	68.7	34.2	66.0	71.9	93.1	57.3	79.6
Instruct Models								
Qwen3.5-9B	0.0	0.7	0.7	5.7	9.9	26.7	8.7	31.1
Qwen3.8-27B	0.0	0.7	8.7	12.7	29.8	48.1	50.5	71.8
Ours								
AUXSPEC-9B-SFT	11.8 (+11.8)	27.3 (+26.6)	14.5 (+13.8)	31.0 (+25.3)	59.4 (+49.5)	81.7 (+55.0)	58.3 (+49.6)	76.7 (+45.6)
AUXSPEC-9B-RL	20.2 (+20.2)	40.3 (+39.6)	30.2 (+29.5)	50.7 (+45.0)	67.6 (+57.7)	87.8 (+61.1)	54.1 (+45.4)	70.9 (+39.8)
AUXSPEC-27B-SFT	23.2 (+23.2)	47.0 (+46.3)	28.3 (+19.6)	50.0 (+37.3)	76.1 (+46.3)	90.8 (+42.7)	67.7 (+17.2)	84.5 (+12.7)
AUXSPEC-27B-RL	35.6 (+35.6)	60.3 (+59.6)	38.1 (+29.4)	58.3 (+45.6)	81.3 (+51.5)	94.7 (+46.6)	72.3 (+21.8)	83.5 (+11.7)

5.3 OUT-OF-DOMAIN EVALUATION

To assess generalization beyond the synthesized data, we evaluate on Verina (Ye et al., 2026a), a 189-problem manually curated benchmark with reference specifications, implementations and test cases. Following the original evaluation, generated specifications must be sound and complete with respect to the reference specification on the test cases, and generated implementations must pass all positive test cases. To match the training inputs, we also evaluate with the test cases in the prompt (w/ tests) besides the original prompt (w/o tests). Table 5 reports pass@1 and pass@4. Both models are trained on the synthesized problems only, yet AUXSPEC-27B-RL scores 60.2% SpecGen pass@1 without test cases in the prompt, trailing only Claude-Opus-5 and Kimi-K3, and both RL models outscore every frontier model on CodeGen, where the 27B model reaches 75.9% pass@1 against 46.3% for the best frontier model and the 9B model reaches the highest pass@4 of all models (94.2%). Test cases in the prompt raise SpecGen pass@4 from 84.7% to 92.6% for the 27B RL model and from 69.8% to 81.0% for the 9B RL model, and the frontier models gain 5 to 24 points under the same change. With the same evaluator, the strongest trained VeriSpecGen model reports 47.6% SpecGen pass@1 and 63.9% pass@10, below the 60.2% pass@1 of the 27B RL model and the 69.8% pass@4 of the 9B RL model.

5.4 RL ANALYSIS

Reinforcement learning from the fine-tuned checkpoints raises SpecGen and CodeGen pass@4 for both models in domain (Table 4), and the gains transfer to Verina, where SpecGen pass@4 without test cases rises from 60.3% to 69.8% and from 78.8% to 84.7% for the 9B and 27B models (Table 5). The reward is computed on the synthesized problems only, yet Verina scores each specification against the reference specification of the benchmark on test cases absent from the prompt, and the gains there show that the policy learns faithfulness to the description beyond the frozen test cases.

Separating test execution from compilation reduces zero-variance groups, in which all rollouts receive equal rewards and which dynamic sampling discards (Yu et al., 2025). Under a reward that

Table 5. Out-of-domain evaluation on Verina. w/o tests and w/ tests indicate whether the Verina test cases are included in the input. Underlines, bold and green numbers as in Table 4. †SpecGen results as reported in the original paper (Ye et al., 2026b), scored by the same Verina evaluator with the Verina prompt template. *CodeGen results as reported in the original paper. In these rows, – marks settings the paper does not report, and italic values are pass@10 in place of pass@4 and are not underlined.

Model	SpecGen				CodeGen			
	w/o Tests		w/ Tests		w/o Tests		w/ Tests	
	pass@1	pass@4	pass@1	pass@4	pass@1	pass@4	pass@1	pass@4
Frontier Models								
Claude-Opus-5	**77.8**	**89.9**	**86.2**	95.2	46.3	75.7	41.5	74.6
Claude-Sonnet-5	57.8	85.7	64.8	90.5	17.5	42.3	15.1	38.6
GPT-5.6-Sol	42.1	73.5	85.1	**97.9**	11.2	33.3	11.6	31.7
GPT-5.6-Luna	50.8	86.2	67.3	94.7	17.5	48.7	22.4	55.0
Kimi-K3	69.2	89.4	77.6	96.3	32.7	69.8	31.9	70.4
DeepSeek-V4-Pro	44.7	76.7	53.8	82.5	24.7	55.0	31.7	66.1
DeepSeek-V4-Flash	51.9	84.7	60.2	92.6	6.5	21.2	5.8	18.5
Instruct Models								
Qwen3.5-9B	2.0	7.0	7.0	18.5	25.1	48.1	23.3	49.2
Qwen3.8-27B	7.2	21.9	28.8	55.0	4.1	15.3	3.0	12.2
Prior Work								
VeriSpecGen-4B	30.3†	*45.2*†	–	–	35.4*	*66.8**	–	–
VeriSpecGen-30B-A3B	47.6†	*63.9*†	–	–	58.7*	*84.0**	–	–
Ours								
AuXSpec-9B-SFT	32.5 (+30.5)	60.3 (+53.3)	39.9 (+32.9)	66.1 (+47.6)	56.7 (+31.6)	79.9 (+31.8)	58.9 (+35.6)	83.6 (+34.4)
AuXSpec-9B-RL	40.2 (+38.2)	69.8 (+62.8)	53.8 (+46.8)	81.0 (+62.5)	74.2 (+49.1)	**94.2** (+46.1)	75.8 (+52.5)	93.1 (+43.9)
AuXSpec-27B-SFT	49.2 (+42.0)	78.8 (+56.9)	56.1 (+27.3)	84.1 (+29.1)	68.9 (+64.8)	89.4 (+74.1)	69.0 (+66.0)	87.8 (+75.6)
AuXSpec-27B-RL	60.2 (+53.0)	84.7 (+62.8)	75.9 (+47.1)	92.6 (+37.6)	**75.9** (+71.8)	93.7 (+78.4)	**77.0** (+74.0)	**94.7** (+82.5)

assigns 0.1 to every rollout failing test execution, no rollout passes this stage in most SpecGen groups of the 9B model, and scoring the same rollouts with Eq. (1) more than doubles the fraction of groups with nonzero variance (19.5% to 51.2% for SpecGen). Rollouts that compile but fail tests or alignment remain the second largest class after compilation failures throughout training (Figure 2c and 2d). A reward proportional to passed tests adds no variance and would reward edits of the expected outputs, which the policy generates in the same rollout (Appendix D.3.1).

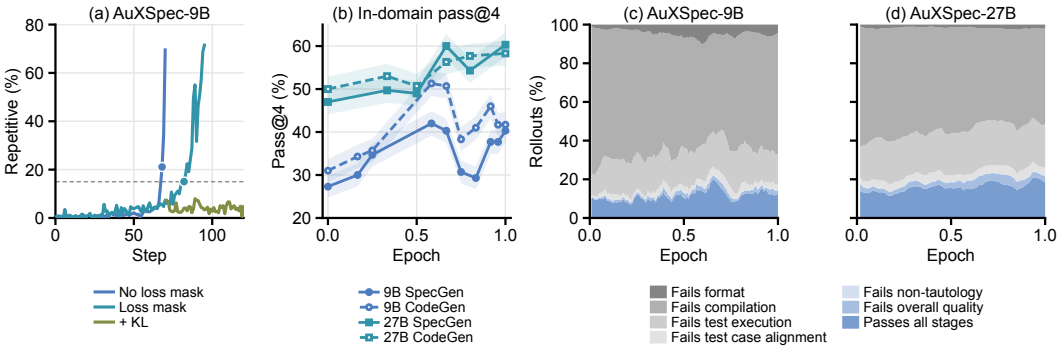


Figure 2. Training dynamics of RL. (a) Repetitive rollouts of the 9B model under the recipe changes, where the loss mask and the KL penalty keep repetition low. Dots mark collapse, the first step above 15%. (b) Pass@4 on the in-domain test set at the evaluated checkpoints, with bands of 1 standard error. (c, d) Fraction of rollouts ending at each stage of Eq. (1) in the final runs, which last 1 epoch, 5-step moving average.

The loss mask and the KL penalty largely resolve repetitive and overlong rollouts, the form collapse takes on the 9B model, where rollouts repeat a span of the thinking segment until the length limit (Figure 2a). A negative reward on repetitive rollouts advances collapse, whereas removing the gradient of these rollouts with the loss mask delays it, and among the changes tried, only the KL penalty toward the fine-tuned model holds repetition low through the epoch (Appendix D.3.2). Truncated rollouts, entropy and gradient norm rise before the reward falls (Appendix D.3.3).

We transfer the recipe from the 9B model to the 27B model, where repetitive and truncated rollouts stay rare but malformed rollouts, which end before closing the thinking segment or the code block, grow although none receives a positive advantage. At the positions where these rollouts end, the probability of the end-of-sequence token rises from below 1e-8 under the fine-tuned model to 7e-3 at step 60, where the token ranks second at most of these positions, and we hypothesize that negative gradients on failed rollouts displace probability to tokens outside the sampled rollouts (Gao et al.,

2025; Deng et al., 2025), the end-of-sequence token among them, consistent with the negative reward that advances collapse on the 9B model (Appendix D.3.3). Token-level masks act on this hypothesis by confining the loss of malformed rollouts to the end-of-sequence token and removing the negative gradient from structural tokens of other failed rollouts (Deng et al., 2025; Guo et al., 2026), after which malformed rollouts fall to 1.6% of rollouts over steps 41 to 60, against 3.2% in the same steps without the masks, and pass@4 rises through the epoch (Figure 2b).

6 Related work

Specification generation. Specification autoformalization translates requirements into formal specifications. Prior work studies postcondition generation from natural language (Endres et al., 2024), executable Python specifications (Chen et al., 2026) and user intent in Dafny (Lahiri, 2024).

Training for specification generation. Training for program specification generation has concentrated on auto-active verification languages (Section 1). In interactive theorem provers, checking a candidate specification on a test needs a proof, and reinforcement learning from verifier feedback has targeted correctness proofs (Li et al., 2026b) but remains underexplored for specification generation (Baksys et al., 2025). Specification generators there are trained instead on synthesized and repaired examples, where a reference implementation labels tests, a proof attempt checks each labeled test, and LLM judgments guide refinement when proof automation is inconclusive (Ye et al., 2026b). AuXSpec also labels tests with reference solutions, while the test results of the executable postcondition filter the training data and define the reinforcement-learning reward (Section 2).

Specification validation. Beyond proving each instantiated proposition (Section 1), a test case is checked by searching for counterexamples to propositions constructed from test cases (Feng et al., 2026). We add an executable postcondition and an equivalence theorem, which, once proved, transfers each test result to the declarative postcondition without a separate proof (Section 2). Equivalence of the two postconditions does not imply agreement with the problem requirements, since both may encode the same misunderstanding. We assess this agreement through tests prepared before specification synthesis, description-based checks during data construction, and expert review of all held-out problems (Sections 2 and 3.1, Appendix G.1).

Adjacent settings. Generating specifications for existing implementations utilizes a verifier to check the specification against the code but assumes the code is correct (Chen et al., 2025; Yan et al., 2026). In this setting, reinforcement learning has included executable checks on mutated-output tests in the reward (Huang et al., 2026). Our SpecGen task instead starts from the problem description, input constraints and tests, without access to an implementation. Vericoding benchmarks evaluate the reverse task, generating implementations and proofs from a given formal specification (Zhao et al., 2026; Bursuc et al., 2026).

7 Conclusion

We presented AuXSpec, a framework that turns the validation of Lean program specifications from a proof per test case into the execution of test cases, which enables reinforcement learning with the test-based reward of code generation. By augmenting each declarative postcondition with an executable counterpart and an equivalence theorem, we check specifications by execution and keep the declarative postcondition for the correctness proof. With this framework, we synthesize 3,037 LeetCode problems with all equivalence theorems proved and the AuXSpec-Test benchmark of 300 expert-reviewed problems with every correctness theorem proved. The same acceptance criteria filter the training data and define the reinforcement-learning reward, and the resulting 9B and 27B models are competitive with large-scale frontier models in domain and on the out-of-domain Verina. A staged reward that credits each acceptance criterion in order turns failing rollouts into gradient, whereas a binary reward leaves most groups without signal. Degenerate rollouts are contained by removing the gradient of such rollouts and anchoring the policy to the fine-tuned model, whereas penalizing them accelerates collapse, and the recipe transfers from the 9B model to the 27B model. The gains of training on the synthesized problems alone carry to Verina, which shows that the reward teaches faithfulness to the description beyond the test cases the reward executes. We discuss the limitations of this work in Appendix H.

AI USE STATEMENT

We used generative AI tools to synthesize the dataset and to produce proofs. The pipeline uses LLMs to synthesize, repair, judge and rewrite specifications and implementations (Section 3, Appendices E and I), and proof agents built on LLMs produce the equivalence and correctness proofs, which the Lean kernel checks (Appendix F). We did not use generative AI tools to develop the conceptual framework or to interpret results, and translation and qualitative data analysis do not apply to this work. We also used generative AI tools to edit the text, create figures, search the literature and format references. We reviewed all AI-assisted work, ran the AI-generated code, checked every number against the script that computed it and every citation against the original publication, and take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

One of the authors carried out the expert review of AUXSPEC-Test and the audit of accepted outputs (Appendix G), and the work involves no other human participants and no personal data. The problems come from LeetCodeDataset (Xia et al., 2025) and TACO (Li et al., 2023), both released under the Apache-2.0 license, with the web-crawled problem statements of TACO under CC BY 4.0. We use them for research with attribution and will release our data and models under compatible licenses. The released models generate specifications and implementations for programming problems, and every accepted implementation remains checkable by proof against the postcondition.

REPRODUCIBILITY STATEMENT

The synthesis pipeline is described in Section 3, with every stage in Appendix E, the counts at each step in Appendix E.5, the models used in Appendix E.8, condensed prompts of the stages and judges in Appendix I, and the proving of the equivalence and correctness theorems in Appendix F. Training is described in Section 4, with the training and sampling settings in Appendices B and B.2, expert iteration in Appendix D.1, the metrics in Appendix B.3 and the reinforcement-learning reward and ablations in Appendix D.3. We will release the synthesized problems with test cases and equivalence proofs, AUXSPEC-Test with correctness proofs, the evaluation code and the model weights.

REFERENCES

Anmol Agarwal, Natalie Neamtu, Pranjal Aggarwal, Seungone Kim, Jannis Limperg, Cedric Flamant, Kanna Shimizu, Bryan Parno, and Sean Welleck. Verus-specgym: An agentic environment for evaluating specification autoformalization. *arXiv preprint arXiv:2605.26457*, 2026.

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations (ICLR)*, 2024.

Zeyuan Allen-Zhu and Yuanzhi Li. Physics of language models: Part 3.1, knowledge storage and extraction. In *International Conference on Machine Learning (ICML)*, 2024.

Anthropic. Introducing claude opus 5. https://www.anthropic.com/news/claude-opus-5, 2026a.

Anthropic. Introducing claude sonnet 5. https://www.anthropic.com/news/claude-sonnet-5, 2026b.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics. In *International Conference on Learning Representations (ICLR)*, 2024.

Mantas Baksys, Stefan Zetzsche, Olivier Bouissou, Remi Delmas, Soonho Kong, and Sean B. Holden. Atlas: Automated toolkit for large-scale verified code synthesis. *arXiv preprint arXiv:2512.10173*, 2025.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, et al. A benchmark for vericoding: Formally verified program synthesis. *Transactions on Machine Learning Research*, 2026.

Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025.

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

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, et al. Automated proof generation for rust code via self-evolution. In *International Conference on Learning Representations (ICLR)*, 2025.

Zaoyu Chen, Jianbo Dai, Boyu Zhu, Jingdong Wang, Huiming Wang, Xin Xu, Haoyang Yuan, Zhijiang Guo, and Xiao-Ming Wu. CodeSpecBench: Benchmarking LLMs for executable behavioral specification generation. *arXiv preprint arXiv:2604.12268*, 2026.

Daixuan Cheng, Shaohan Huang, and Furu Wei. Adapting large language models via reading comprehension. In *International Conference on Learning Representations (ICLR)*, 2024.

Xingyu Dang, Christina Baek, J. Zico Kolter, and Aditi Raghunathan. Assessing diversity collapse in reasoning. In *ICLR 2025 Workshop on Scaling Self-Improving Foundation Models without Human Supervision*, 2025.

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

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction (CADE)*, 2015.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Wenlong Deng, Yi Ren, Muchen Li, Danica J. Sutherland, Xiaoxiao Li, and Christos Thrampoulidis. On the effect of negative gradient in group relative deep reinforcement optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation. In *IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 72–79, 2025.

Quinn Dougherty, Max von Hippel, Simon Henniger, Hazel Shackleton, and Mike Dodds. Fvspec: Real-world property-based tests as lean challenges. *arXiv preprint arXiv:2606.01008*, 2026.

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. doi: 10.1145/3660791.

Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pîrlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. Certified program synthesis with a multi-modal verifier. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2026.

Jia Fu, Xinyu Yang, Hongzhi Zhang, Yahui Liu, Jingyuan Zhang, Qi Wang, Fuzheng Zhang, and Guorui Zhou. Klear-CodeTest: Scalable test case generation for code reinforcement learning. *arXiv preprint arXiv:2508.05710*, 2025.

Chang Gao, Chujie Zheng, Xiong-Hui Chen, Kai Dang, Shixuan Liu, Bowen Yu, An Yang, Shuai Bai, Jingren Zhou, and Junyang Lin. Soft adaptive policy optimization. *arXiv preprint arXiv:2511.20347*, 2025.

GasStationManager. SafeVerify: A Lean 4 script for robustly verifying submitted proofs of theorems and implementations of functions. https://github.com/GasStationManager/SafeVerify, 2024.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

Ruohao Guo, Afshin Oroojlooy, Roshan Sridhar, Miguel Ballesteros, Alan Ritter, and Dan Roth. Tree-based dialogue reinforced policy optimization for red-teaming attacks. In *International Conference on Learning Representations (ICLR)*, 2026.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

Jujie He, Jiacai Liu, Chris Yuhao Liu, Rui Yan, Chaojie Wang, Peng Cheng, Xiaoyu Zhang, Fuxiang Zhang, Jiacheng Xu, Wei Shen, et al. Skywork open reasoner 1 technical report. *arXiv preprint arXiv:2505.22312*, 2025.

Zhongmou He, Yee Man Choi, Kexun Zhang, Ivan Bercovich, Jiabao Ji, Junting Zhou, Dejia Xu, Aidan Zhang, Yixiao Zeng, and Lei Li. HardTestGen: A high-quality RL verifier generation pipeline for LLM algorithmic coding. In *International Conference on Learning Representations (ICLR)*, 2026.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10):576–580, 1969.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations (ICLR)*, 2020.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. REINFORCE++: Stabilizing critic-free policy optimization with global advantage normalization. *arXiv preprint arXiv:2501.03262*, 2025.

Zhechong Huang, Zhao Zhang, Zeyu Sun, Huifeng Sun, and Yingfei Xiong. Specrl: Reinforcement learning with test-based completeness rewards for formal specification synthesis. In *IEEE Transactions on Software Engineering*, 2026.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhengbao Jiang, Zhiqing Sun, Weijia Shi, Pedro Rodriguez, Chunting Zhou, Graham Neubig, Xi Victoria Lin, Wen tau Yih, and Srinivasan Iyer. Instruction-tuned language models are better knowledge learners. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2024.

Woogyeol Jin, Taywon Min, Yongjin Yang, Dennis Wei, Yi Zhou, Swanand Ravindra Kadhe, Nathalie Baracaldo, and Kimin Lee. Entropy-aware on-policy distillation of language models. In *International Conference on Machine Learning (ICML)*, 2026.

Kimi Team. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, JD Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M. Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Shuvendu K. Lahiri. Evaluating LLM-driven user-intent formalization for verification-aware languages. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2024.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Lean community. REPL: A read-eval-print loop for Lean 4. https://github.com/leanprover-community/repl, 2023.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. TACO: Topics in algorithmic COde generation dataset, 2023. URL https://arxiv.org/abs/2312.14852.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan ang Gao, Wenkai Yang, Zhiyuan Liu, et al. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *arXiv preprint arXiv:2604.13016*, 2026a.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

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)*, 2026b.

Wenze Lin, Jiale Zhao, Xitai Jiang, Songde Rao, Yining Li, Shenzhi Wang, Bingxiang He, and Gao Huang. On-policy distillation with verifiable reward. *arXiv preprint arXiv:2608.24696*, 2026.

Mingjie Liu, Shizhe Diao, Ximing Lu, Jian Hu, Xin Dong, Yejin Choi, Jan Kautz, and Yi Dong. ProRL: Prolonged reinforcement learning expands reasoning boundaries in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Kevin Lu and Thinking Machines Lab. On-policy distillation. https://thinkingmachines.ai/blog/on-policy-distillation/, 2025.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. In *International Conference on Learning Representations (ICLR)*, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Noam Razin, Sadhika Malladi, Adithya Bhaskar, Danqi Chen, Sanjeev Arora, and Boris Hanin. Unintentional unalignment: Likelihood displacement in direct preference optimization. In *International Conference on Learning Representations (ICLR)*, 2025a.

Noam Razin, Zixuan Wang, Hubert Strauss, Stanley Wei, Jason D. Lee, and Sanjeev Arora. What makes a reward model a good teacher? an optimization perspective. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Chi Ruan, Dongfu Jiang, Huaye Zeng, Ping Nie, and Wenhu Chen. EvolveCoder: Evolving test cases via adversarial verification for code reinforcement learning. *arXiv preprint arXiv:2603.12698*, 2026.

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina Lean server: A high-performance Lean server for large-scale verification. In *The 5th Workshop on Mathematical Reasoning and AI at the 39th Conference on Neural Information Processing Systems (NeurIPS)*, 2025.

John Schulman. Approximating KL divergence. http://joschu.net/blog/kl-approx.html, 2020.

Vedant Shah, Johan Obando-Ceron, Vineet Jain, Brian Bartoldson, Bhavya Kailkhura, Sarthak Mittal, Glen Berseth, Pablo Samuel Castro, Yoshua Bengio, Esmeralda S. Whitammer, et al. A comedy of estimators: On KL regularization in RL training of LLMs. In *Conference on Language Modeling (COLM)*, 2026.

Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, et al. rStar2-Agent: Agentic reasoning technical report. *arXiv preprint arXiv:2508.20722*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. CodeHacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 2352–2382, 2026.

Zhengxiang Shi and Aldo Lipani. Don't stop pretraining? make prompt-based fine-tuning powerful learner. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

The mathlib Community. The Lean mathematical library. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381, 2020.

Chen Wang, Zhaochun Li, Jionghao Bai, Yining Zhang, Hexuan Deng, Ge Lan, and Yue Wang. Distilled reinforcement learning for LLM post-training. *arXiv preprint arXiv:2607.17247*, 2026a.

Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, et al. LEGO-prover: Neural theorem proving with growing libraries. In *International Conference on Learning Representations (ICLR)*, 2024.

Rui Wang, Hongru Wang, Yi Chen, Boyang Xue, Tianqing Fang, Wenhao Yu, and Kam-Fai Wong. Demystifying on-policy distillation: Roles, pathologies, and regulations. *arXiv preprint arXiv:2607.13399*, 2026b.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. CodeContests+: High-quality test case generation for competitive programming. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5576–5600, 2025a.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

Sean Welleck, Ilia Kulikov, Stephen Roller, Emily Dinan, Kyunghyun Cho, and Jason Weston. Neural text generation with unlikelihood training. In *International Conference on Learning Representations (ICLR)*, 2020.

Yunhui Xia, Wei Shen, Yan Wang, Jason Klein Liu, Huifeng Sun, Siyue Wu, Jian Hu, and Xiaolong Xu. LeetCodeDataset: A temporal dataset for robust evaluation and efficient training of code LLMs. *arXiv preprint arXiv:2504.14655*, 2025.

Zichen Xie, Mrigank Pawagi, Yuxin Liu, Aaditi Rai, Lize Shao, John Berberian, Sicong Che, and Wenxi Wang. Vericontest: A competitive-programming benchmark for verifiable code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2026.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. In *International Conference on Learning Representations (ICLR)*, 2025.

Hongling Xu, Qi Zhu, Heyuan Deng, Jinpeng Li, Lu Hou, Yasheng Wang, Lifeng Shang, Ruifeng Xu, and Fei Mi. KDRL: Post-training reasoning LLMs via unified knowledge distillation and reinforcement learning. *arXiv preprint arXiv:2506.02208*, 2025.

Chuanhao Yan, Fengdi Che, Xuhan Huang, Xu Xu, Xin Li, Yizhi Li, Xingwei Qu, Jingzhe Shi, Chenghua Lin, Yaodong Yang, et al. Re:form – reducing human priors in scalable formal software verification with RL in LLMs: A preliminary study on Dafny. *Transactions on Machine Learning Research*, 2026.

Fanpeng Yang, Xing Li, Shuling Wang, Jie An, Zeyu Sun, Shenghua Feng, Wenhan Wang, Weiyi Wang, Naijun Zhan, and Fanjiang Xu. How powerful are LLMs in generating formal program specifications? In *International Conference on Machine Learning (ICML)*, 2026.

Zitong Yang, Neil Band, Shuangping Li, Emmanuel Candès, and Tatsunori Hashimoto. Synthetic continued pretraining. In *International Conference on Learning Representations (ICLR)*, 2025.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *International Conference on Learning Representations (ICLR)*, 2026a.

Zhe Ye, Aidan Z. H. Yang, Huangyuan Su, Zhenyu Liao, Samuel Tenka, Zhizhen Qin, Udaya Ghai, Dawn Song, and Soonho Kong. Intent-aligned formal specification synthesis via traceable refinement. *arXiv preprint arXiv:2604.10392*, 2026b.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Zichao Yu, Chengzhi Yu, Shengze Xu, Yujin Han, Bingqing Jiang, Xu Wang, and Difan Zou. Mismatch matters: On-policy distillation beyond token agreement. *arXiv preprint arXiv:2608.09836*, 2026.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

Z.ai. Glm-5.2: Built for long-horizon tasks. https://z.ai/blog/glm-5.2, 2026.

Lingfei Zeng, Fengdi Che, Xuhan Huang, Fei Ye, Xu Xu, Binhang Yuan, and Jie Fu. Veriequivbench: An equivalence score for ground-truth-free evaluation of formally verifiable code. In *International Conference on Learning Representations (ICLR)*, 2025.

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. In *International Conference on Machine Learning (ICML)*, 2026.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Yichi Zhou, Jianqiu Zhao, Yongxin Zhang, Bohan Wang, Siran Wang, Luoxin Chen, Jiahui Wang, Haowei Chen, Allan Jie, Xinbo Zhang, et al. Solving formal math problems by decomposition and iterative reflection. *arXiv preprint arXiv:2507.15225*, 2025.

A FILE FORMAT

A.1 DECLARATIONS

Table 6 lists the declarations that implement the terms of the paper.

Table 6. Declarations per AUXSPEC problem.

Term	Declaration	Description
Precondition	`isValidInput`	Executable predicate encoding the input constraints
Postcondition	`<algo>_spec`	`Prop` relating input and output
Executable postcondition	`<algo>_check`	`Bool` function of input and output
Equivalence theorem	`<algo>_check_iff_spec`	`_check` decides `_spec` on valid inputs
Implementation	`<algo>`	Total function computing the output
Correctness theorem	`<algo>_correct`	`<algo>` satisfies `_spec` on valid inputs

A.2 FORMAT

Each file includes precondition, postcondition and executable postcondition with equivalence theorem, implementation with correctness theorem, and test cases. Helper definitions follow a topological order of dependency. The namespace, the type abbreviations, the printed expressions of the frozen test cases, and both theorem statements are assembled. Table 7 lists the test layers.

Table 7. Test layers with the component under test and the error detected.

Layer	Component Under Test	Test Case	Assertion	Error Detected
L1[a]	Precondition	Valid input	Accepts	Overly restrictive
L1[b]		Invalid input	Rejects	Overly permissive
L2[a]	Executable postcondition	Valid input, correct output	Accepts	Incomplete
L2[b]		Valid input, incorrect output	Rejects	Unsound
L3	Implementation	Valid input, expected output	Matches	Incorrect
L4	Executable postcondition on implementation output	Valid input	Accepts	Implementation output rejected

B CONFIGURATION

B.1 TRAINING

Both model sizes are fine-tuned with the settings in Table 8. In a round of expert iteration, the model is fine-tuned for 3 epochs on the training samples of that round. Checkpoints are evaluated after each epoch, and the models reported in the main text are the third-epoch checkpoints. Loss is taken on assistant tokens only.

Table 8. Training settings of supervised fine-tuning (left) and reinforcement learning (right).

Supervised fine-tuning	9B	27B
Base model	Qwen3.5-9B	Qwen3.8-27B
Hardware	4 GH200	8 GH200
Batch (sequences)	64	
Sequence length (tokens)	16,384	
Learning rate	1e-5, cosine	
Warmup	5%	
Epochs	3	

Reinforcement learning	9B	27B
Start model	AUXSPEC-9B-SFT	AUXSPEC-27B-SFT
Prompts per step	32	64
Rollouts per prompt	16	
Prompts per mini-batch	8	
Learning rate	1e-6, constant	
Clip range ϵ_{low} / ϵ_{high}	0.2 / 0.28	
KL coefficient β	0.04	
Temperature	0.7	
Max new tokens	16,384	
Training steps (1 epoch)	120	60

Reinforcement learning uses the settings on the right of Table 8. Runs from AUXSPEC-9B-SFT train on 3,794 prompts over 2,672 problems, and the run from AUXSPEC-27B-SFT on 3,863 prompts over the same problems, adding 69 CodeGen prompts whose given specifications passed all acceptance criteria in earlier runs. A SpecGen rollout needs 12 tests and a CodeGen rollout 3, fewer for problems with fewer test rows. A rollout fails compilation when a declaration does not elaborate, a test cannot be evaluated, compilation exceeds the time limit, the compiler reports an error

without a position, or the rollout uses `sorry` outside the equivalence theorem, `admit`, `axiom` or `native_decide`. A rollout receives 0.4 when a judge call fails, which happens for 20 of the 87,680 rollouts of the AUXSPEC-27B run. A rollout is repetitive when a 40-character n-gram, taken at a stride of 4 characters, occurs at least 20 times and fewer than 60% of the n-grams are distinct, a rule that flags 5 of 17,502 rollouts that compile. We define collapse as the first step at which more than 15% of rollouts are repetitive or malformed, the threshold at which the run monitor stops a run.

B.2 INFERENCE

Expert iteration, the evaluation of our models and the analyses of Appendix D sample with the settings of Table 9. Baselines use the sampling settings recommended for each model with the same maximum output length, except the long-reasoning models DeepSeek-V4-Flash, DeepSeek-V4-Pro and Kimi-K3, which run without a length limit.

Table 9. Inference settings.

Inference engine	vLLM
Temperature	0.3 (evaluation), 0.7 (expert iteration and analyses)
Top-p	0.95
Max new tokens	16,384
Samples per problem, compile@k and pass@k	4

B.3 METRICS

Compile@k and Pass@k. We follow the standard definition of pass@k (Chen et al., 2021) and define compile@k by analogy. For a problem with k samples, compile@k is 1 when at least one sample compiles, and pass@k is 1 when at least one sample meets the acceptance criteria of Section 2. The reported value is the mean over problems, which equals the unbiased estimator of pass@k (Chen et al., 2021) since exactly k samples are drawn per problem. Test case alignment compares the test lines of a sample with the frozen test cases regardless of layer headings, and Lean code blocks are extracted regardless of the capitalization of the language tag.

C TRAINING DATA

C.1 SOURCE SELECTION

The LeetCode problems combine 2,869 problems of LeetCodeDataset (Xia et al., 2025) with 746 free problems collected from the site after the dataset snapshot. Of the 746 problems, 263 pass a buildability filter that requires at least 3 examples and a Python function signature and excludes database, shell, concurrency and design problems. The seeds are 3,109 problems, 2,864 from LeetCodeDataset and 245 from the site. We take the Codeforces and AtCoder problems from the train split of TACO (Li et al., 2023). Problems are admitted with at least 3 tests, a reference solution, and a single accepted answer, since one reference output cannot label a problem with several accepted answers. Stored tests whose inputs violate the constraints of the problem description are removed before use, which drops 1,028 of 90,885 test rows on Codeforces and 2,119 of 22,115 on AtCoder. Difficulty follows the labels of LeetCode and TACO, where the TACO labels MEDIUM_HARD and unknown count as medium and VERY_HARD counts as hard.

C.2 PROBLEM FORMAT

Each Codeforces and AtCoder problem, which reads standard input, is rephrased into a typed function interface and admitted only when the reference solution still passes the stored tests through that interface. Conversion admits 3,490 of the 4,314 Codeforces problems (80.9%). The rejected problems have stored tests that no reference solution reproduces (361), inputs that change under parsing and unparsing (259), several accepted answers (184), or other harness errors (20). Freezing the test cases removes 205 more problems, 201 of which lack 3 invalid inputs for L1[b], and leaves 3,285. Of the 1,440 AtCoder problems, conversion admits 892, an audit for descriptions that drop a stated condition and for examples that violate the stated constraints leaves 677, and freezing leaves 623. Most AtCoder rejections at conversion are inputs that change under parsing and unparsing (352 of 548), which trace to faulty stored tests such as leading zeros or counts that do not match the input.

C.3 DATA BREAKDOWN

Table 10 lists the number of samples of each task for supervised fine-tuning, reinforcement learning, and AUXSPEC-Test. A reinforcement learning sample is a generation prompt, and repair prompts are built during training from failed rollouts (Section 4.3).

Table 10. Number of samples of each task in each stage.

Task	SFT	RL	AUXSPEC-Test
SpecGen	6,528	2,672	300
SpecGen-Repair	3,807	0	131
CodeGen	6,645	1,122	300
CodeGen-Repair	2,512	0	103
Total	19,492	3,794	834

The 19,492 supervised fine-tuning samples are drawn from the synthesis pipeline of Section 3 and expert iteration (Table 11). An expert-iteration repair chain accepted after more than 1 round becomes 1 generation sample with a reasoning trace over the first attempt, feedback and repair.

Table 11. Supervised fine-tuning samples by source.

Source	SpecGen	SpecGen-Repair	CodeGen	CodeGen-Repair	Total
Synthesis pipeline	4,786	2,479	4,475	1,100	12,840
Expert iteration	373	1,328	739	1,412	3,852
Expert iteration, rewritten	1,369	0	1,431	0	2,800
Total	6,528	3,807	6,645	2,512	19,492

The corpus builder also removes samples longer than 16,384 tokens (42), duplicate samples (57), samples beyond 3 per problem and task (11), and 1 sample that is not a whole file.

C.4 EXPERT ITERATION DETAILS

Each round samples 1 refinement chain per problem and task from a fine-tuned Qwen3.8-27B. A chain generates an output, compiles the output, and repairs the output for up to 8 rounds, or 16 rounds in the last round of expert iteration. A chain is accepted under the criteria of Section 2, and the reasons of each judge rejection enter the next repair round as review text. Table 12 lists the rounds. The first 2 rounds sample from the model fine-tuned on the LeetCode data, and the last round from the model fine-tuned on the data of the first 2 rounds and the Codeforces pipeline data. Accepted chains that pass at the first round become generation samples, and later rounds become repair samples or are rewritten into generation samples (Appendix C.3). A scan over all refinement records adds 384 samples from accepted chains that no round collected. The chains on reinforcement learning problems in the last round add no supervised fine-tuning samples.

Table 12. Rounds of expert iteration. A chain is 1 refinement run on 1 problem and task.

Round	Maximum repair rounds	Chains	Accepted chains	SFT samples
LeetCode	8	5,239	2,268	3,610
Codeforces	8	3,778	1,248	2,137
Last round	16	3,208	1,444	521
Scan over all records	–	–	–	384
Total		12,225	4,960	6,652

Expert iteration adds repairs more than first attempts. Of the 2,268 accepted chains of the LeetCode round, 685 pass at the first round, and of the 298 accepted SpecGen chains on reinforcement learning problems in the last round, 290 need at least 1 repair round.

C.5 DECONTAMINATION

The training problems pass filters that remove the 300 AUXSPEC-Test problems, the problems that Verina links to, and the reinforcement learning problems from supervised fine-tuning. Verina problems are matched through the LeetCode links in the Verina descriptions, where 43 of the 189 problems link to 36 LeetCode problems. The filters remove 21 samples of AUXSPEC-Test problems and 166 samples of 32 LeetCode problems that Verina links to, and a recount of the final training data finds no problem of these sets.

D TRAINING METHOD ANALYSIS

D.1 EXPERT ITERATION

SpecGen pass@4 rises at each stage of the SFT data (Table 13), from 30.7% on the LeetCode data to 39.3% after adding the first round of expert iteration and 47.3% after further adding the Codeforces problems and 2 more rounds.

Table 13. Pass@4 on the in-domain test set after each stage of the SFT data, sampled with the analysis settings of Table 9. Each stage adds data to the previous stage and is fine-tuned from Qwen3.8-27B for 3 epochs.

Training data	Problems	Samples	SpecGen	CodeGen
LeetCode	2,687	7,993	30.7	38.0
+ expert iteration round 1	2,733	11,811	39.3	52.0
+ Codeforces, expert iteration rounds 2 and 3 (AUXSPEC-27B-SFT)	3,993	19,492	47.3	48.7

D.2 CONTINUED PRETRAINING

Motivation. An early supervised fine-tuning run of Qwen3.5-9B compiles 5.05% of the 1,208 SpecGen samples and 3.8% of all 3,596 samples on the held-out problems. Classifying the first compiler error of the failed samples shows that 40.8% misuse Mathlib declarations, through application type mismatches (17.3%), invalid fields or projections (14.7%) and type mismatches (8.7%), and 65.2% once unknown identifiers (11.6%) and missing instances (12.8%) are counted. With retrieved declaration names in 58% of the answers, the failures lie in the use of Mathlib.

Corpus. Synthesis augments prompts with Mathlib declarations retrieved for the problem (Appendix E.1.2), and across the 3,109 LeetCode problems the retrievals cover 1,381 distinct declarations, of which 1,298 are found in the pinned Mathlib source. These declarations lie in 411 files, and we take all declarations of those files rather than the retrieved ones alone, which gives 26,936 declaration heads. The heads are rewritten with DeepSeek-V4-Flash (DeepSeek-AI, 2026) into English lessons that state the content, the application, the occasion and the contrast of a declaration, which follows the rewriting of raw corpora into reading-comprehension text (Cheng et al., 2024) and textbook-style synthetic corpora (Gunasekar et al., 2023). Together with 8,312 rows of Mathlib source, 2,336 lessons on decidability routes and 3,790 pairs of compiler error and repair, the corpus $\mathcal{M}$ holds 41,374 rows and about 43M tokens.

Training and result. Adding problems to supervised fine-tuning raises pass@4 more than continued pretraining does (Table 14). Continued pretraining minimizes the next-token loss on the corpus,

$$\mathcal{L}_{\text{CPT}}(\theta) = -\,\mathbb{E}_{u\sim\mathcal{M}} \sum_{t} \log \pi_\theta(u_t \mid u_{<t}), \tag{3}$$

for 2 epochs, 92 steps, on Qwen3.5-9B base with learning rate 5e-6 and sequence length 16,384, and supervised fine-tuning follows on 4,826 rows from 1,631 problems. Domain corpora of this kind help on downstream tasks (Gururangan et al., 2020) and formal provers are pretrained on Lean and mathematics before fine-tuning (Azerbayev et al., 2024; Xin et al., 2025), yet the benefit of continued pretraining is not universal (Shi & Lipani, 2023). Here continued pretraining raises SpecGen pass@4 from 8.2 to 9.8 and pass@4 over the 4 tasks from 26.4 to 27.4, whereas adding 1,056 problems, 3,167 rows, to supervised fine-tuning raises the same 2 numbers to 13.7 and 30.2. The corpus loss falls from 1.03 to 0.69 with held-out perplexity 2.03. The corpus is therefore fit while the downstream gain stays at 1.0 point, the distance between document perplexity and downstream use reported for knowledge injection through continued pretraining (Jiang et al., 2024). Content memorized from a corpus becomes usable after many representations of the same content (Allen-Zhu & Li, 2024; Yang et al., 2025), and each of the 26,936 declarations appears in 1 lesson here. Lessons state the meaning of declarations, whereas a compiling specification requires declarations composed into terms that the elaborator accepts, and added problems carry test cases and judges that label samples of that composition. Later rounds therefore put the effort into data.

Table 14. Continued pretraining against more fine-tuning data for Qwen3.5-9B, pass@k with 4 samples on an earlier version of the training data (4,826 rows from 1,631 problems). SFT, more data adds 3,167 rows from 1,056 problems. Parentheses give the change from SFT.

Model	AUXSPEC-Test						Verina SpecGen	
	SpecGen		CodeGen		All 4 tasks		Basic + Advanced	
	p@1	p@4	p@1	p@4	p@1	p@4	p@1	p@4
Qwen3.5-9B	0.0	0.0	0.0	0.0	2.4	7.3	2.0	7.0
SFT	2.0	8.2	4.3	15.6	13.6	26.4	16.0	35.4
CPT + SFT	2.3 (+0.3)	9.8 (+1.6)	5.9 (+1.6)	16.0 (+0.4)	14.1 (+0.5)	27.4 (+1.0)	16.0 (0.0)	37.6 (+2.2)
SFT, more data	7.0 (+5.0)	13.7 (+5.5)	7.0 (+2.7)	18.8 (+3.2)	18.3 (+4.7)	30.2 (+3.8)	18.3 (+2.3)	43.9 (+8.5)

D.3 REINFORCEMENT LEARNING

The final recipe for AUXSPEC-9B-RL and AUXSPEC-27B-RL combines the progressive reward of Eq. (1), the self-repair buffer, and the loss mask and the KL penalty on the clipped surrogate of DAPO with batch-normalized advantages (Section 4.3), and adds the token-level masks for the 27B model. The runs on the 9B model develop the recipe, and both models train for about 1 epoch, 120 steps for the 9B model and 60 steps for the 27B model, with the settings of Table 8. The reported 9B model is the checkpoint at step 80, the evaluated checkpoint of the run with the KL penalty with the highest generation pass@4 (Figure 2b), and the reported 27B model is the checkpoint at step 60. The subsections below analyze the reward stages, the recipe ablations and the causes of collapse.

D.3.1 REWARD STAGES

The test execution stage separates from compilation failures the next largest class of failed rollouts (Table 15). Test execution fails only when each compiler error comes from a test that ran and returned false, and any other error, including a test that cannot be elaborated or does not finish, counts as a compilation failure. Without the stage, rollouts that compile but fail a test receive the value of compilation failures, and rescoring the rollouts of an earlier 9B run with Eq. (1) moves 9.8% of SpecGen rollouts from 0.1 to 0.2. The fractions of Section 5.4 are measured on the 3,434 SpecGen groups of the first 33 steps of that run with 8 rollouts per group.

Table 15. Fraction of rollouts ending at each stage of Eq. (1) over all 4 tasks, in the runs of Figure 2c and 2d, through step 120 for the 9B model and step 60 for the 27B model.

Stage reached	Reward	27B (%)	9B (%)
Malformed rollout or unchanged repair	0	2.26	4.43
Fewer tests than required	0.1	0.16	0.54
Fails compilation	0.1	52.36	61.31
Fails test execution	0.2	21.76	17.71
Fails test case alignment	0.3	4.09	2.78
Judge call fails	0.4	0.02	0.01
Fails non-tautology	0.6	0.48	0.30
Fails overall quality	0.8	2.97	1.60
Passes all stages	1.0	15.90	11.31
Rollouts		87,680	88,992

A reward proportional to the fraction of passed tests would invite hacking and add little signal. The policy generates the tests in a rollout, and proportional credit would reward edits of expected outputs, while adding little variance over Eq. (1) (51.2% to 51.4% of SpecGen groups with nonzero variance, 82.5% to 82.9% of CodeGen groups). Eq. (1) therefore gives test execution a flat value, whereas prior work grants proportional credit on test suites fixed before training (Huang et al., 2026).

Each rule against reward hacking closes a shortcut that the stages would otherwise reward. A test that calls no declaration of the output, such as `#guard true`, passes without testing the output, and counting only tests that call a declaration keeps the test stages meaningful. An escape hatch, the `sorry` and `admit` tactics, an `axiom` declaration or the `native_decide` tactic, lets an output compile without the checks that later stages assume. A test that cannot run would otherwise earn the test execution credit of a test that ran and failed. An unchanged repair would keep the credit of the stages the failed input already passes. The non-tautology stage precedes the overall quality stage, the order in which evaluation runs the judges.

D.3.2 COLLAPSE ABLATIONS

RL from the fine-tuned models collapses into repetitive, truncated or malformed rollouts (Table 16). Repetitive rollouts loop inside the thinking segment until the length limit, truncated rollouts reach the limit without looping, and malformed rollouts end before closing the thinking segment or the code block. The final recipe results from the changes below, each targeting one form, and the resumes of the masked 9B run from step 70 without the KL penalty collapse at steps 80 to 83, which bounds the variation between runs from one checkpoint.

Repetition. Repetitive rollouts dominate the collapse of the 9B model. To push the policy away from them, we first give these rollouts a reward of -0.2, which advances collapse by 4 to 5 steps, with or without n-grams of the prompt in the repetition rule. Zeroing the gradient of these rollouts with the loss mask instead delays collapse from step 68 to step 82 and stays in the final recipe.

Truncation. Truncated rollouts rise first before collapse (Appendix D.3.3). To remove the gradient of rollouts that reach the length limit, we mask truncated rollouts, the overlong filtering of DAPO (Yu et al., 2025). The mask delays collapse to step 80, but truncated rollouts rise from 18.9% to 33.3% and the collapse turns into length inflation, and the final recipe leaves truncated rollouts unmasked. Penalizing truncated rollouts instead has been reported to help in other settings (He et al., 2025; Shang et al., 2025).

Policy update. To stabilize the policy update, we replace the clipped surrogate with SAPO, which uses a soft gate and is reported to reduce early collapse (Gao et al., 2025), and separately add a KL penalty against the fine-tuned model (Liu et al., 2025). SAPO advances collapse whether applied from the fine-tuned model (step 82 to 52) or resumed from step 70 (step 80 to 76), and the SAPO run with the KL penalty collapses at step 109. The KL penalty with $\beta = 0.04$ (Shao et al., 2024) keeps repetitive rollouts below 10% through step 120, whereas $\beta = 0.001$ leaves collapse at step 83. The final recipe therefore keeps the clipped surrogate of DAPO and adds the KL penalty.

Malformed rollouts. For the 27B model, the final recipe adds the token-level masks of Section 5.4 against malformed rollouts, and Appendix D.3.3 analyzes the cause of these rollouts. The mean reward and response length of the RL runs of Section 5.4 are in Figure 3.

Table 16. Collapse steps of RL runs, single seed. Start is the step of the checkpoint a run resumes from, 0 for the fine-tuned model. Resumed runs start from a checkpoint of the first run in the same block, taken before collapse. Collapse is the first step at which more than 15% of rollouts are repetitive or malformed, and – marks a run without collapse through the last step (120 for the 9B model, 60 for the 27B model).

Model	Start	Recipe	Collapse form	Collapse
9B	0	no loss mask, no KL penalty	repetitive	68
9B	60	+ reward -0.2 for repetitive rollouts	repetitive	64
9B	60	+ reward -0.2, n-grams of the prompt excluded	repetitive	63
9B	60	+ mask on truncated rollouts	repetitive, truncated	80
9B	0	loss mask	repetitive	82
9B	70	loss mask, resumed	repetitive	80
9B	70	+ KL penalty, $\beta = 0.001$	repetitive	83
9B	70	+ SAPO	repetitive	76
9B	70	+ KL penalty, $\beta = 0.04$ (final recipe)	none	–
9B	0	loss mask + SAPO	repetitive	52
9B	40	+ KL penalty, $\beta = 0.04$	repetitive	109
27B	0	loss mask + KL penalty	none	–
27B	40	+ token-level masks (final recipe)	none	–

D.3.3 COLLAPSE ATTRIBUTION

In the 9B run with the loss mask, truncated rollouts rise first, and entropy and gradient norm rise before the reward falls (Table 17), consistent with the signals that precede collapse in multi-turn agent RL (Wang et al., 2025b). Passing rollouts keep a median length between 2,131 and 2,817 tokens over steps 61 to 95, while failed rollouts reach the 16,384-token limit. Generation and repair rollouts collapse at the same steps, and the problems with repetitive rollouts grow from 46% to 97% of the problems of each 5-step window, most of which had no repetitive rollout before. Of the 3,509 rollouts without a closed thinking segment during collapse, 96% repeat a span until the length limit.

In the 27B run, malformed rollouts grow although none of the 660 malformed rollouts with a gradient over the 60 steps receives a positive advantage, and repair rollouts are malformed less often than

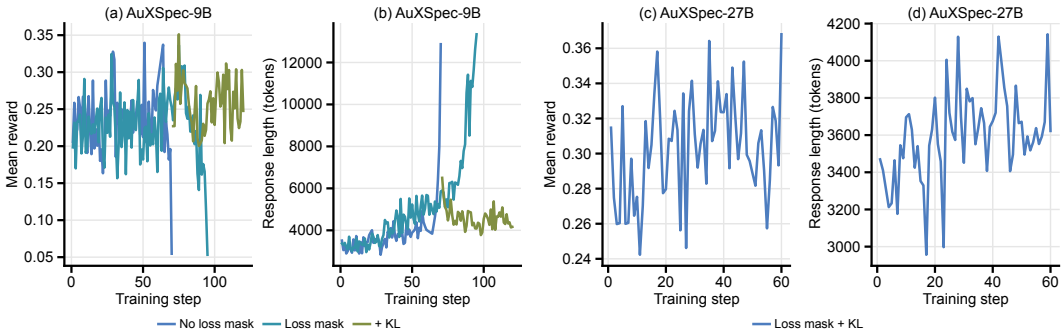


Figure 3. Mean reward and mean response length per step of the RL runs of Section 5.4.

Table 17. Training statistics of the 9B run with the loss mask around collapse, in 5-step windows.

Steps	Gradient norm	Entropy	Truncated (%)	Mean reward
71–75	0.49	0.23	10	0.276
76–80	0.66	0.26	14	0.283
81–85	0.68	0.28	20	0.235
86–90	1.04	0.30	40	0.199
91–95	1.28	0.38	62	0.134

generation rollouts (0.57% against 1.90%). At the positions where malformed rollouts of steps 51 to 60 end, the median probability of the end-of-sequence token is 5.9e-9 under the fine-tuned model, 1.2e-8 at step 40 and 7.1e-3 at step 60 (Table 18). The presence penalty of 1.0 used in training raises the mean probability of sampling the end-of-sequence token at these positions from 2.3% to 6.0% at step 60 and leaves the probability at 0 at step 40. At step 60, the end-of-sequence token ranks second at 147 of 200 such positions, behind a content token, and the median per-token gap between the log-probabilities of training and inference is 2.3e-4 nats. A negative advantage raises the logits of unsampled tokens in proportion to the probabilities of these tokens (Gao et al., 2025), which alone cannot lift a token from 1e-8 to second rank, and negative gradients also move probability through shared representations to tokens outside the trained responses (Deng et al., 2025; Razin et al., 2025a). We hypothesize that the rise of the end-of-sequence token comes from this displacement by negative gradients on failed rollouts. The token-level masks of Section 5.4 act on this hypothesis. For malformed rollouts, the loss keeps only the end-of-sequence token, which receives the negative gradient. For other failed rollouts, structural tokens (code fence tokens, the closing thinking tag and end-of-sequence tokens) receive no negative gradient (Guo et al., 2026). Over steps 41 to 60, malformed rollouts are 1.6% of the rollouts of the masked run against 3.2% of the run without the masks.

Table 18. Probability of the end-of-sequence token at 200 positions where malformed rollouts of steps 51 to 60 of the 27B run end. Sampling uses temperature 0.7, top-p 0.95 and top-k 20, with presence penalty (pp) 0 or 1, and the last column counts positions where the sampling probability under pp 1 exceeds 1%.

Weights	Median probability	Mean sampling, pp 0 (%)	Mean sampling, pp 1 (%)	Positions above 1%
AuXSpec-27B-SFT	5.9e-9	0.0	0.0	0 / 200
RL step 40	1.2e-8	0.0	0.0	0 / 200
RL step 60	7.1e-3	2.3	6.0	51 / 200

Appendix D.4 reports RL with a distillation term from AuXSpec-27B-SFT.

D.4 On-policy distillation

To transfer the capability of AuXSpec-27B-SFT, the strongest of our models on AuXSpec-Test, to the smaller AuXSpec-9B-SFT, we apply on-policy distillation (OPD) and OPD within reinforcement learning (Agarwal et al., 2024; Xu et al., 2025), but neither improves the student model. OPD, with the k3 estimator in the advantage, suffers from mode collapse, where the student model concentrates on the decidable form of the executable postcondition, leading to a compile rate of 2.0% without compiler feedback. SpecGen pass@4 drops from 24.3 to 5.3 while CodeGen changes by at most 3 points. Distillation within reinforcement learning suffers from teacher redundancy, where a

teacher model fine-tuned on the same data carries 4% to 7% of the advantage with little information about which samples succeed. We next describe the setup, the failure patterns and the attribution.

Setup. On-policy distillation trains student models on self-generated responses with token-level supervision from teacher models (Agarwal et al., 2024). We select AUXSPEC-27B-SFT as the teacher model π_T, the strongest of our models on AUXSPEC-Test (Table 4), and AUXSPEC-9B-SFT as the student model π_θ, which has the same vocabulary and chat template as the teacher model for a locally served teacher model to score student tokens. For a prompt x and a response $y \sim \pi_\theta(\cdot \mid x)$, the per-token log-ratio

$$\ell_t = \log \pi_\theta(y_t \mid x, y_{<t}) - \log \pi_T(y_t \mid x, y_{<t}) \tag{4}$$

is the k1 estimate of the reverse KL divergence $\mathrm{KL}(\pi_\theta \,\|\, \pi_T)$, and $k_3(t) = e^{-\ell_t} - 1 + \ell_t$ is a non-negative estimate of the same divergence (Schulman, 2020). Both settings enter the policy gradient through the per-token advantage A_t.

- **OPD**: the per-token advantage is $A_t = -k_3(t)$ with no task reward, as in on-policy distillation (Agarwal et al., 2024; Lu & Thinking Machines Lab, 2025), over 200 steps on 12,791 prompts (1 epoch), with student responses and teacher log-probabilities at temperature 1.0.
- **OPD within RL**: the objective adds a distillation term with per-token advantage $-\ell_t$, weighted by $\beta = 0.1$, to the reinforcement learning objective of Appendix D.3, as in joint distillation and reinforcement learning (Xu et al., 2025) with the k1 estimator in place of k2. Variants keep the distillation term only where the sign agrees with the task advantage (Lin et al., 2026) or drop the term and reweight the task advantage by clipped teacher ratios (Wang et al., 2026a). Student responses are sampled at temperature 0.7 and teacher log-probabilities at 1.0.

Diagnostics. Besides pass@k, we track token entropy, the fraction of generated text inside repetition loops, the fraction of SpecGen samples in the decidable form of Section 3.1, and teacher informativeness, the mean of $-\ell_t$ on sequences with positive task advantage minus the same mean on sequences with negative advantage (Wang et al., 2026b).

Results. OPD degrades SpecGen and changes CodeGen by at most 3 points (Table 19). SpecGen pass@4 drops from 24.3 to 5.3 on AUXSPEC-Test and from 42.0 to 18.5 on Verina advanced problems, and CodeGen pass@4 from 30.7 to 27.7. In OPD within RL, teacher informativeness stays at +0.004 or below and the teacher term carries 4% to 7% of the total advantage over the first steps. The 3 runs ended at steps 4 to 9 when the judge service ran out of budget, and the measured informativeness led us to stop this line rather than resume.

Table 19. On-policy distillation of AUXSPEC-9B-SFT from AUXSPEC-27B-SFT, pass@k with 4 samples. Parentheses give the change from AUXSPEC-9B-SFT, and the teacher row is a reference. *An earlier checkpoint of AUXSPEC-27B-SFT, used as the teacher.

Model	AUXSPEC-Test				Verina SpecGen			
	SpecGen		CodeGen		All		Advanced	
	p@1	p@4	p@1	p@4	p@1	p@4	p@1	p@4
AUXSPEC-9B-SFT	9.6	24.3	13.5	30.7	30.3	61.4	17.6	42.0
+ OPD	2.3 (−7.3)	5.3 (−19.0)	15.2 (+1.7)	27.7 (−3.0)	29.1 (−1.2)	43.9 (−17.5)	8.3 (−9.3)	18.5 (−23.5)
AUXSPEC-27B-SFT* (teacher)	20.4	47.0	23.1	44.3	49.1	82.5	41.4	79.0

Phenomena. After OPD, 1,147 of 1,200 SpecGen samples (95.6%) fail at compilation, against 1,032 (86.0%) before OPD and 875 (72.9%) for the teacher (Figure 4), and OPD also narrows the sample distribution and lets repetition loops grow unpenalized.

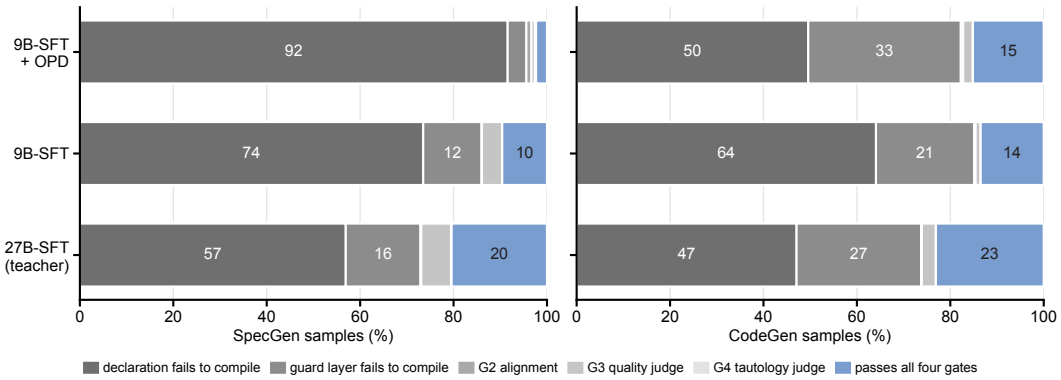


Figure 4. Distribution of SpecGen and CodeGen samples by failure stage, 1,200 samples per model and task. After OPD, 92% of SpecGen samples stop at a declaration that does not compile, while CodeGen samples shift from declaration errors to guard-layer errors with an unchanged pass rate.

Template concentration. The number of SpecGen samples stating the executable postcondition in the decidable form increases from 150 (12.5%) before OPD to 706 of 1,200 (58.8%) after OPD, against 114 (9.5%) for the teacher (Figure 5). Within this form, the compile rate falls from 10.7% to 2.0%, against 21.9% for the teacher, and failed samples report a missing decidability instance.

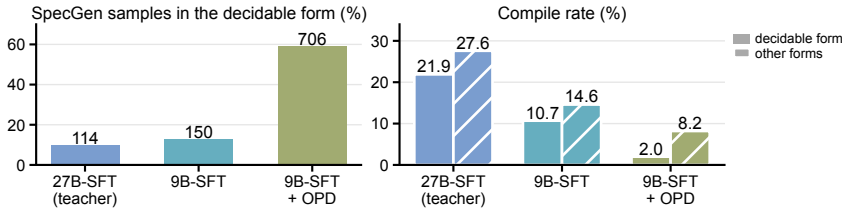


Figure 5. Use of the decidable form in SpecGen samples, with counts out of 1,200 above the bars (left), and compile rates of the decidable form against other forms (right).

Diversity collapse. Token entropy falls from 0.451 to 0.012 (−97.3%) within the first 100 steps (Figure 7), the entropy collapse also reported for reinforcement learning (Yu et al., 2025; He et al., 2025). Near-duplicate sample pairs occur on 40.7% of SpecGen problems, against 0.3% before OPD (Figure 6). On Verina basic problems, pass@1 rises from 39.8 to 44.7 while pass@4 falls from 75.9 to 63.0, the pattern of diversity collapse in which gains at small k are paired with losses at larger k (Dang et al., 2025; Yue et al., 2025).

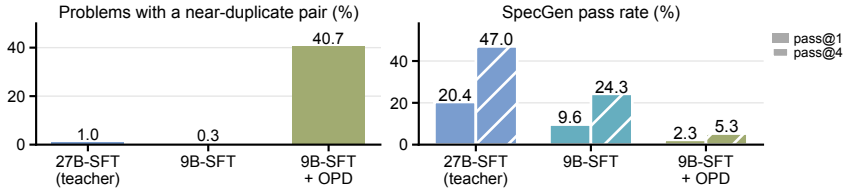


Figure 6. Near-duplicate SpecGen samples per problem (left) and SpecGen pass@1 against pass@4 (right).

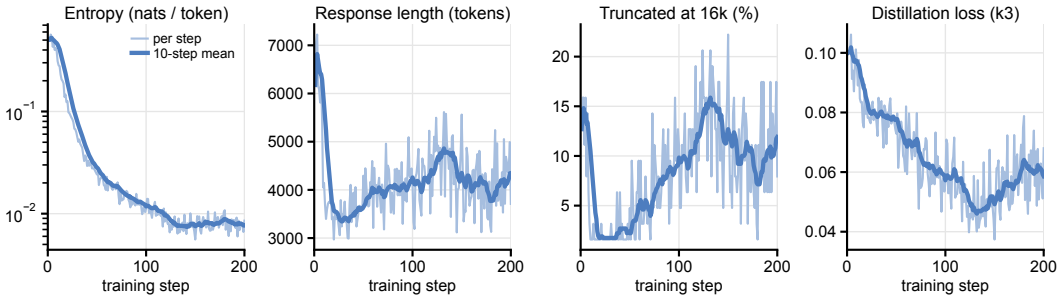


Figure 7. Training dynamics of OPD: token entropy, mean response length, fraction of responses truncated at 16,384 tokens, and distillation loss. Light lines give single steps and dark lines a moving average over 10 steps.

Degenerate agreement. Repetition loops (Holtzman et al., 2020) grow during training, and from step 51 on, looping responses hold 43% to 51% of the generated text. Inside a loop, the teacher ranks the next student token first on 98.9% of tokens, against 90.7% before the loop, and the k3 loss per token therefore drops from 0.136 to 0.019 (−86.0%, Figure 8). The same degenerate agreement is reported for on-policy distillation (Yu et al., 2026). Loops account for 1 of the 60 SpecGen problems lost to OPD, where all 4 samples are truncated or loop.

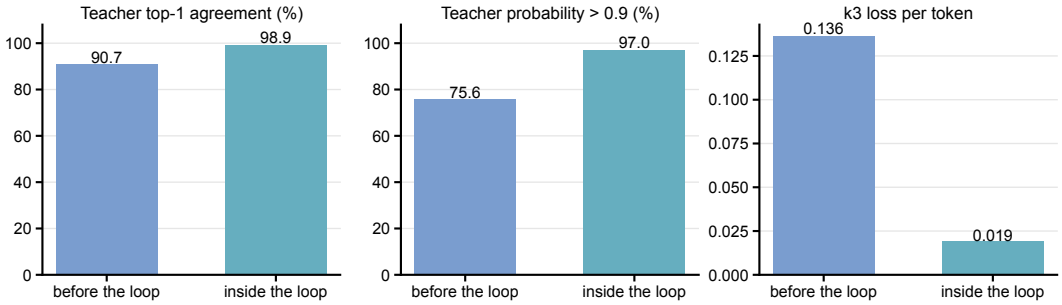


Figure 8. Degenerate agreement inside repetition loops: teacher top-1 agreement, fraction of tokens with teacher probability above 0.9, and k3 loss per token, on 26 looping SpecGen outputs.

Attribution. We attribute the SpecGen drop to estimator bias, which lowers entropy throughout training, to mode collapse, which moves probability onto a form that often fails to compile, and to teacher redundancy, which leaves the teacher term uninformative about sample success.

1. *Estimator bias.* With k_3 inside the advantage, the policy gradient is a biased estimate of the reverse KL gradient that also raises the forward KL, and this configuration is reported to collapse training (Shah et al., 2026). The entropy decline of Figure 7 matches this bias.
2. *Mode collapse.* The pipeline sets the decidable form as the first executable postcondition and replaces the form through instance synthesis or a separate definition when compilation fails (Section 3.1), and the form covers 35.5% of synthesized problems as a result. Evaluation scores single samples without compiler feedback. We conjecture that the mode-seeking reverse KL (Agarwal et al., 2024; Jin et al., 2026) moves probability onto this frequent form, including on problems where the form does not compile. The student uses the form in 54.6% of samples on problems where the teacher never uses the form. OPD therefore amplifies the form beyond teacher usage.
3. *Teacher redundancy.* Teacher models with higher scores help only when offering capabilities beyond the experience of the student (Li et al., 2026a). Teacher and student are fine-tuned on the same data, and although OPD within RL adds the fine-grained reward of reinforcement learning, the teacher term stays uninformative about which samples succeed.

Identical tokenizers and chat templates rule out a format mismatch, and on problems the teacher fails, the student stops at the failing stage of the teacher no more often after OPD than before.

E PIPELINE DETAILS

Each stage below ends with an analysis of the stage on the 3,037 synthesized problems.

E.1 PREPROCESSING

E.1.1 TEST CASES

For each of 3,109 problems (719 easy, 1,622 medium, 768 hard), we screen the stored examples and propose negative test cases, invalid inputs and incorrect outputs beside the positive test cases. Invalid inputs are proposed by category (a value just outside a stated bound, a size outside the stated range, a container of the wrong shape, a broken structural invariant, parameters invalid only together, a value outside the stated domain), and incorrect outputs by category (a near miss, a suboptimal feasible answer, an unattainable answer, the answer under a relaxed constraint, an output of the wrong shape, a degenerate answer). Benchmarks store negative test cases beside positive ones (Ye et al., 2026a;b), since a specification that accepts correct outputs on the given examples can still accept incorrect outputs (Endres et al., 2024).

A proposed incorrect output is admitted when the reference solution, run on the proposed valid input, returns an output that is not a reordering of the proposed output. Invalid inputs are proposed together with the violated constraint and are dropped on coinciding with a valid example of the problem. Cases too large for compiled evaluation and cases whose stored answer is null are refused before any model call, with the measured thresholds in Appendix E.5.

The 27,981 test cases, 9 per problem, are assembled into 18 `#guard` commands per problem, one per test case and layer of Table 7, with each positive test case reused at $L1^a$, $L2^a$, L3, and L4. A `#guard` command runs compiled code, and a passed test case relies on the compiler and runtime as trusted base, the same as `native_decide`. For the test cases, we store the provenance (public example or synthesized) and the admission round. A test case judged wrong is replaced by a held-back test case at most once per problem.

Analysis. The frozen test cases, negative test cases included, are shown to the generation model, and the negative test cases act as part of the problem statement rather than as a hidden filter. Of 3,276 specification candidates tested at $L1^a$ and $L1^b$, 8 (0.24%) fail only at $L1^b$, and of 3,195 executable postconditions tested at $L2^a$ and $L2^b$, 4 (0.13%) fail only at $L2^b$. The 4 failures at $L2^b$ have one cause, a lossy output decoder in the specification, such as a tree decoder that drops children below null nodes. The 8 failures at $L1^b$ fall on 5 problems, and in 4 of the 5 problems the frozen invalid input is in fact valid. Repair narrowed the precondition to reject that input in all 5 problems. Frozen test values keep test cases from adapting to a wrong specification, and these 5 preconditions are the only adaptation in the opposite direction.

Screening. Before any model call, test cases are filtered by size and by null answers. A test case containing a list or string longer than 128 items is refused, since a case is frozen as a literal that compiled evaluation builds item by item. We cap compilation at 600 seconds on the Lean server and observe that a test case exceeding the cap terminates the serving REPL. The threshold was measured over the stored test cases, and any threshold between 102 and 185 items refuses the same test cases. A case whose stored answer is null is refused in the layers that read an answer, since the source dataset stores null wherever the answer is a mutated argument, and a null true answer leaves each negative built from the null answer meaningless.

E.1.2 LIBRARY RETRIEVAL

Models reference library declarations absent from the pinned library version (Chakraborty et al., 2025). Typed queries against Mathlib, for a definition, a theorem, or an instance, are generated together with one sentence stating the use the specification has for the answer, and relevance is judged against that sentence. Search candidates are deduplicated, elaborated with `#check` against the pinned toolchain, and judged for relevance by an LLM given the problem description. Specification synthesis is augmented with the declarations admitted by the relevance judge, 4.2 per problem, each with a checked signature and a one-sentence reason from the LLM. Appendix E.5 reports the counts at each step.

Analysis. Elaboration keeps declarations that exist only in newer Mathlib versions out of specification synthesis (Table 20). 23,608 queries (14,888 definition, 5,052 theorem, 3,668 instance) produced 92,104 candidates. Of the elaborated declarations, 5.4% were absent from the pinned Mathlib, and review dropped 10.0% of the reviewed candidates. Of the 13,191 stored declarations, specification synthesis of the 3,037 problems received 12,876, 4.2 per problem, with 131 problems receiving none. The absent declarations originate in a search index built on the latest Mathlib, while synthesis compiles against the pinned version. Retrieved declarations leave the first compilation unchanged. Specifications that use a retrieved declaration by the full name compile at the first attempt in 83.2% of 1,871 problems, and the other specifications in 83.9% of 1,032.

Table 20. Library retrieval by step over 3,109 problems and 3,715 rounds. The first 5 rows count work across all rounds, and the last row counts the declarations passed to specification synthesis, so a problem retrieved more than once counts only the last round there.

Step	Count	Note
Queries generated	23,608	Definition, theorem, and instance queries
Candidates returned	92,104	Search hits
Declarations elaborated	83,679	Deduplicated, #check against the pinned toolchain
Declarations present	79,126	5.4% of elaborated declarations were absent
Reviewed	17,372	1,729 dropped as irrelevant
Stored	13,191	Last record per problem, 4.2 per problem

E.2 SPECIFICATIONS

Planning. A type plan fixes the declaration names and one type per parameter, with a conversion function wherever the stored value has a different type. The postcondition is then planned regardless of decidability, which keeps computation out of the postcondition. A frozen test value is printed once as the expression used by all test layers to prevent models from altering the value.

Formalization. Each component is generated separately and compiled with the accepted components before the full file is assembled. The `#guard` commands of the layers are generated from the expressions the plan fixed, and each command is accepted at the first passing check in the order shape, verbatim expression, token-level expression, and judge review of equivalence. The shape check admits one command per test case, attached to the declaration of that layer and with the polarity of the test case. The expression check compares the tokens of the printed expression, which prevents 2 values from being collapsed by whitespace.

Executable postcondition. The executable postcondition is first set to the decidable form of the postcondition, and the file is compiled. When no decidability instance is derived, a model supplies instances from the compiler diagnostics for up to 3 rounds. When the supplied instances still fail to elaborate, a separate definition is generated. Only a separate definition can disagree with the postcondition, which the L2^{a} and L2^{b} test cases detect on the tested outputs and the equivalence theorem excludes on all valid inputs. Appendix E.5 reports the counts by form.

E.2.1 EXAMPLE PROBLEMS

Separate executable postcondition. We illustrate the format with problem *1955 Count Number of Special Subsequences* from AUXSPEC-Test (Figures 9 and 10). In this problem, we are given an array over $\{0, 1, 2\}$ and need to count, modulo $10^9 + 7$, the subsequences of the form $0^a1^b2^c$ with $a, b, c \geq 1$. The postcondition states the result as the size of the set of index sets whose selected elements take this form. The executable postcondition enumerates all subsets of the indices, a brute-force computation that finishes on the short test inputs. The implementation counts in 1 pass with 3 counters, for subsequences of 0s, of 0s followed by 1s, and of 0s, 1s and 2s, since an element either extends or skips the subsequences counted so far. Since neither specification mentions the counters, the correctness theorem relates the recurrence to the set of index sets.

For this problem, a natural postcondition characterizes the valid index sets S as exactly those that select a subsequence of the form $0^+1^+2^+$ from the input. This characterization is not directly executable, since it ranges over arbitrary finite sets of natural numbers. However, every relevant index set is contained in the input indices. The executable postcondition therefore enumerates $2^{\{0,\ldots,\text{input.length}-1\}}$ and counts the subsets satisfying the same property. The two formulations describe the same relation but are not definitionally identical in Lean, so the equivalence theorem con-

nects the logical characterization with the bounded enumeration. Once this equivalence is proved, we can test the natural postcondition by executing the executable counterpart.

```
import Mathlib

/-!
# Count Number of Special Subsequences

A sequence is special if it consists of a positive number of 0s, followed by a positive number
of 1s, then a positive number of 2s.

Given an array nums (consisting of only integers 0, 1, and 2), return the number of different
subsequences that are special. Since the answer may be very large, return it modulo 10^9 + 7.
-/

abbrev Input := List Int
abbrev Output := Nat

-- Specification helpers
def IsSpecialSubsequence (nums : List Int) (S : Finset Nat) : Prop :=
  (∀ i ∈ S, i < nums.length) ∧
    ∃ a b c : List Int, a ≠ [] ∧ b ≠ [] ∧ c ≠ [] ∧
      (∀ x ∈ a, x = 0) ∧ (∀ x ∈ b, x = 1) ∧ (∀ x ∈ c, x = 2) ∧
      ((nums.zipIdx.filter fun p => p.2 ∈ S).map Prod.fst) = a ++ b ++ c

-- Precondition
def isValidInput (input : Input) : Bool :=
  1 ≤ input.length && input.length ≤ 10^5 &&
    input.all (fun x => x = 0 || x = 1 || x = 2)

-- Postcondition
def numberOfSpecialSubsequences_spec (input : Input) (result : Output) : Prop :=
  ∃ C : Finset (Finset Nat),
    result = C.card % (10^9 + 7) ∧
      ∀ S : Finset Nat, S ∈ C ↔ IsSpecialSubsequence input S

-- Executable postcondition helpers
def specialSplits (l : List Int) : Bool :=
  decide
    (∃ i j : Fin (l.length + 1),
      i ≤ j ∧
        l.take i.val ≠ [] ∧ (l.drop i.val).take (j.val - i.val) ≠ [] ∧ l.drop j.val ≠ [] ∧
          (∀ x ∈ l.take i.val, x = 0) ∧ (∀ x ∈ (l.drop i.val).take (j.val - i.val), x = 1) ∧
            (∀ x ∈ l.drop j.val, x = 2))

def isSpecialSet (nums : List Int) (S : Finset Nat) : Bool :=
  decide (∀ i ∈ S, i < nums.length) &&
    specialSplits ((nums.zipIdx.filter fun p => p.2 ∈ S).map Prod.fst)

-- Executable postcondition
def numberOfSpecialSubsequences_check (input : Input) (result : Output) : Bool :=
  result ==
    ((Finset.range input.length).powerset.filter
    (fun S => isSpecialSet input S = true)).card % (10 ^ 9 + 7)

-- Equivalence theorem
theorem numberOfSpecialSubsequences_check_iff_spec (input : Input) (result : Output)
    (h : isValidInput input) :
    numberOfSpecialSubsequences_check input result
    ↔ numberOfSpecialSubsequences_spec input result := by
  sorry

-- Implementation helpers
structure DPState where
  zeros : Nat
  ones : Nat
  twos : Nat

def dpStep (s : DPState) (x : Int) : DPState :=
  if x = 0 then { s with zeros := (2 * s.zeros + 1) % (10 ^ 9 + 7) }
  else if x = 1 then { s with ones := (2 * s.ones + s.zeros) % (10 ^ 9 + 7) }
  else { s with twos := (2 * s.twos + s.ones) % (10 ^ 9 + 7) }

-- Implementation
def countSpecialSubsequences (input : Input) : Output :=
  (input.foldl dpStep ⟨0, 0, 0⟩).twos

-- Correctness theorem
theorem countSpecialSubsequences_correct (input : Input) (h : isValidInput input) :
    numberOfSpecialSubsequences_spec input (countSpecialSubsequences input) := by
  sorry
```

Figure 9. Declarations of the example problem, without namespace and docstrings (tests in Figure 10).

```
/-! ## Test cases -/

-- L1a
#guard isValidInput [0, 1, 2, 2]
#guard isValidInput [2, 2, 0, 0]
#guard isValidInput [0, 1, 2, 0, 1, 2]

-- L1b
#guard !(isValidInput [])
#guard !(isValidInput [3])
#guard !(isValidInput [(-1)])

-- L2a
#guard numberOfSpecialSubsequences_check [0, 1, 2, 2] 3
#guard numberOfSpecialSubsequences_check [2, 2, 0, 0] 0
#guard numberOfSpecialSubsequences_check [0, 1, 2, 0, 1, 2] 7

-- L2b
#guard !(numberOfSpecialSubsequences_check [0, 1, 2, 2] 2)
#guard !(numberOfSpecialSubsequences_check [2, 2, 0, 0] 1)
#guard !(numberOfSpecialSubsequences_check [0, 1, 2, 0, 1, 2] 0)

-- L3
#guard countSpecialSubsequences [0, 1, 2, 2] == 3
#guard countSpecialSubsequences [2, 2, 0, 0] == 0
#guard countSpecialSubsequences [0, 1, 2, 0, 1, 2] == 7

-- L4
#guard numberOfSpecialSubsequences_check [0, 1, 2, 2] (countSpecialSubsequences [0, 1, 2, 2])
#guard numberOfSpecialSubsequences_check [2, 2, 0, 0] (countSpecialSubsequences [2, 2, 0, 0])
#guard numberOfSpecialSubsequences_check [0, 1, 2, 0, 1, 2] (countSpecialSubsequences [0, 1, 2,
    0, 1, 2])
```

Figure 10. Test cases of the example problem, grouped by test layer.

Executable postcondition through type class resolution. We illustrate the decidable form with problem *523 Continuous Subarray Sum* from AUXSPEC-Test (Figure 11). In this problem, we are given an array of non-negative integers and a positive integer k and need to decide whether some contiguous subarray of length at least 2 sums to a multiple of k. The postcondition states that the result is true exactly when there exist 2 indices and an integer multiplier n such that the subarray between the indices sums to $n \cdot k$.

The executable postcondition applies `decide` to the postcondition, which compiles only when type class resolution finds a `Decidable` instance for the unfolded proposition. The existential over natural-number indices and an integer multiplier has no such instance as stated, since the quantifiers range over unbounded domains. The pipeline therefore synthesizes a type class instance that bounds both indices by the length of the array, replaces the multiplier by integer divisibility, and proves the bounded proposition equivalent to the original with `decidable_of_iff`. Type class resolution then assembles the executable postcondition from this instance and the library instances for bounded quantifiers and divisibility, so the executable postcondition agrees with the postcondition by construction (Appendix F.2). The implementation, omitted from the figure, scans prefix sums modulo k with a map from each remainder to the earliest prefix index at which the remainder occurs, and the correctness theorem relates this scan to the existential of the postcondition.

```
import Mathlib

/-!
# Continuous Subarray Sum

Given an integer array nums and an integer k, return true if nums has a good subarray or false
otherwise.
A good subarray is a subarray where:

its length is at least two, and
the sum of the elements of the subarray is a multiple of k.
-/

abbrev Input := (List Int) × (Nat)
abbrev Output := Bool

-- Specification helpers
def IsGoodSubarray (nums : List Int) (k : Nat) (startIdx endIdx : Nat) (n : Int) : Prop :=
  startIdx + 2 ≤ endIdx ∧ endIdx ≤ nums.length ∧
    ((nums.drop startIdx).take (endIdx - startIdx)).sum = n * (k : Int)

-- Precondition
def isValidInput (input : Input) : Bool :=
  1 ≤ input.1.length && input.1.length ≤ 10^5 &&
    input.1.all (fun x => 0 ≤ x && x ≤ 10^9) &&
    input.1.sum ≤ 2^31 - 1 && 1 ≤ input.2 && input.2 ≤ 2^31 - 1

-- Postcondition
def hasGoodSubarray_spec (input : Input) (result : Output) : Prop :=
  result = true ↔ ∃ (startIdx endIdx : Nat) (n : Int),
    IsGoodSubarray input.1 input.2 startIdx endIdx n

-- Decidability instance
instance decidableGoodSubarrayExists (nums : List Int) (k : Nat) :
    Decidable (∃ startIdx endIdx n, IsGoodSubarray nums k startIdx endIdx n) :=
  decidable_of_iff
    (∃ startIdx < nums.length + 1,
      ∃ endIdx < nums.length + 1,
        startIdx + 2 ≤ endIdx ∧
          (k : Int) | ((nums.drop startIdx).take (endIdx - startIdx)).sum)
    (by
      constructor
      · rintro ⟨startIdx, _, endIdx, hend, hge, hdvd⟩
        rcases hdvd with ⟨n, hn⟩
        refine ⟨startIdx, endIdx, n, hge, ?_, ?_⟩
        · omega
        · rw [mul_comm]
          exact hn
      · rintro ⟨startIdx, endIdx, n, hge, hend, hsum⟩
        refine ⟨startIdx, ?_, endIdx, ?_, hge, ⟨n, ?_⟩⟩
        · omega
        · omega
        · rw [mul_comm]
          exact hsum)

-- Executable postcondition
def hasGoodSubarray_check (input : Input) (result : Output) : Bool :=
  @decide (hasGoodSubarray_spec input result)
    (by unfold hasGoodSubarray_spec; infer_instance)

-- Equivalence theorem
theorem hasGoodSubarray_check_iff_spec (input : Input) (result : Output)
    (h : isValidInput input) :
    hasGoodSubarray_check input result ↔ hasGoodSubarray_spec input result := by
  sorry
```

Figure 11. Declarations of the second example problem, whose executable postcondition is `decide` of the postcondition, with the `Decidable` instance found by type class resolution from a synthesized instance. Namespace, docstrings, the implementation with the correctness theorem, and the test cases are omitted.

Judges. The postcondition judge is given the problem description, the constraints, and the postcondition source. A rejection quotes a sentence of the description and names an output that the postcondition admits and the problem excludes. The executable postcondition judge is given the constraints, the accepted precondition and postcondition, and the executable postcondition, and a rejection names a valid input with a witness or counterexample that the executable postcondition misses or wrongly admits. The tautology judge rejects a postcondition that computes the output. Problems whose natural statement is the transformation itself are labeled in advance by a majority vote of 3 classifier calls, and the label exempts the problem from the tautology judge. The rejected component is repaired with the named error, or replanned for an error in the plan.

E.2.2 Analysis

The specification judges rarely reject, and the errors that pass all checks lie in the meaning of the specification. Each verdict in Table 21 reads a distinct component that compiles and passes the test layers of the component. The specification judges reject 1.7% to 3.8% of the verdicts, and all rejected problems are accepted after repair. The errors that proving or human review later found had all passed the test layers and the judge of the erroneous component. In 2 problems a correct rejection led to a repair that introduced the error found by review, and in 1 problem the judge accepted on a second reading the defect that the judge had named.

Table 21. Verdicts of the judges on components that compile and pass the test layers of the component. A problem counts as rejected when at least 1 verdict on the problem is a rejection.

Judge	Verdicts	Rejections	Problems rejected	Accepted after rejection
Faithfulness	3,237	123 (3.8%)	112	112
Non-tautology	3,143	69 (2.2%)	61	61
Completeness	3,159	55 (1.7%)	42	42
Implementation	3,667	717 (19.6%)	414	281

The decidable form is reliable where the postcondition quantifies over bounded types (Figure 12). The decidable form serves 1,079 of the 3,037 problems (35.5%). Of the other 1,958 problems, 1,343 have postconditions that quantify over unbounded types, 319 were synthesized before instance synthesis was added, 287 fail instance synthesis for 3 rounds, 6 remain unexplained, and 3 are routed to the decidable form but keep a separate definition in the final file. Where the decidable form applies, the executable postcondition passes all checks at the first attempt in 772 of 787 problems (98.1%), against 1,178 of 2,250 (52.4%) for separately generated executable postconditions. Missing instances arise from the decidable form. The pipeline first sets the executable postcondition to `decide` applied to the postcondition, which compiles only when Lean finds a decidability instance for the postcondition. A postcondition that quantifies over an unbounded type (e.g., over all natural numbers or all lists) has no computable instance as stated, and instance synthesis fails, for example on add-digits, where no decidability instance can be synthesized for a postcondition that quantifies over all digit roots d.

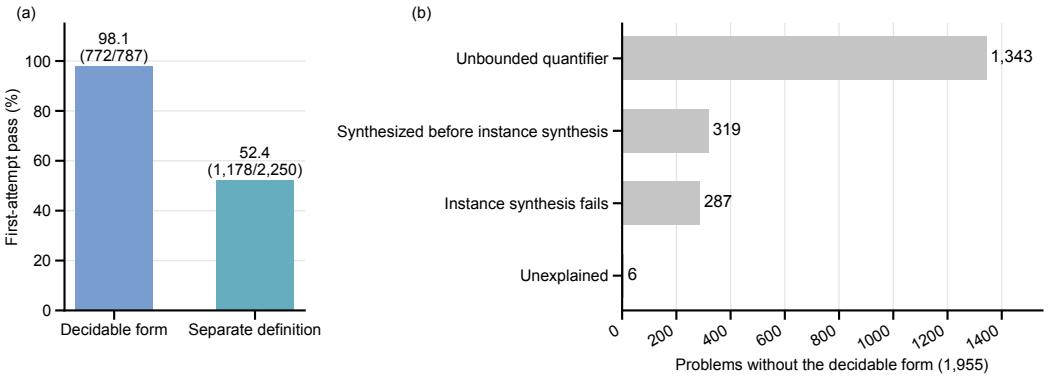


Figure 12. (a) Executable postconditions that pass all checks at the first attempt, by route. (b) Reasons that the decidable form is not used.

E.3 IMPLEMENTATION

Implementation planning is given the accepted specification as read-only text and selects an algorithm with cost argument, listing specification definitions to call and definitions to replace with reason. The plan is checked before generation for a termination argument based on a decreasing quantity of the input, a stated cost consistent with the approach and the input bounds, a replacement justified by cost or by a different question, and an algorithm that computes the answer, without search over candidates. Beyond the properties in Section 3.1, the implementation judge flags computations already available in the specification.

E.3.1 ANALYSIS

Test cases on small inputs admit false positives (Li et al., 2022), and the implementation judge rejects 717 of the 3,667 implementations (19.6%) that pass all test cases (Figure 13).

- **Fuel parameters (270).** The implementation recurses on a fuel argument, which hides nontermination on inputs beyond the test cases.
- **Cost beyond the input bounds (234).** The implementation is correct but exceeds the cost that the stated input bounds allow, known as a slow positive (Li et al., 2022).
- **Restated executable postconditions (198).** The implementation repeats a computation of the executable postcondition that already runs within the stated input bounds.
- **Replayed test cases (15).** The implementation returns stored outputs for the tested inputs.

The implementation judge is waived after 2 retries, and 133 problems keep an implementation that the judge rejected. Of these, 89 are rejected for a fuel argument that may hide nontermination, 27 for cost beyond the input bounds, 12 for restating the executable postcondition and 5 for replaying test cases. 12 of the 133 are in AUXSPEC-Test, and for the 9 among them rejected for fuel, the correctness proofs show that the fuel suffices on all valid inputs.

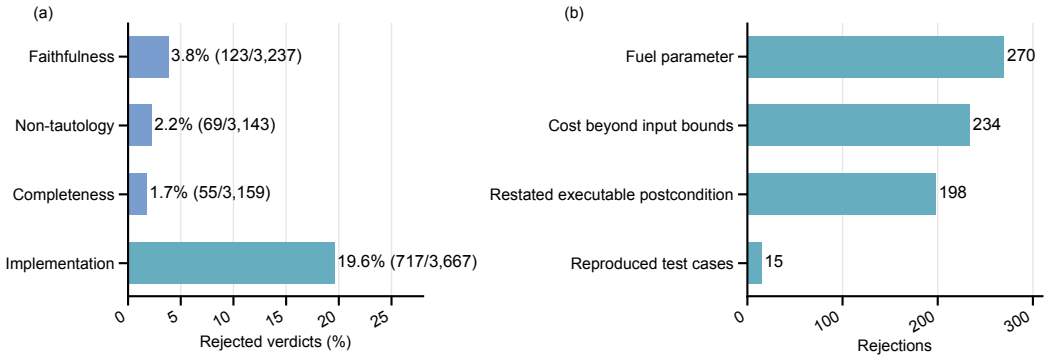


Figure 13. (a) Rejections by each judge on compiled components that pass the test layers. (b) Implementation judge rejections by reason.

E.4 REPAIR

Compilation runs through the Kimina Lean server (Santos et al., 2025), a REPL-based server (Lean community, 2023). Repair revises rejected components separately for at most 18 rounds per run of the repair loop, which a problem re-enters after replanning. From the second round on, the repair input includes a digest of earlier attempts, and a root-cause analysis runs for errors that line numbers cannot attribute. Repair stops early when a revision repeats an earlier diff or leaves the component unchanged. Where a repair attributes the error to a test case, the attribution is reviewed by a judge, and a confirmed attribution replaces the test case with a held-back test case at most once per problem.

E.4.1 ANALYSIS

Repair recovers most first failures within 2 rounds (Figures 14 and 15). The fraction of the 3,037 problems accepted within k repair rounds rises from 78.1% at $k = 0$ to 93.9% at $k = 1$ and 97.5% at $k = 2$ for the postcondition, from 67.7% to 91.9% and 97.7% for the executable postcondition, and

from 64.3% to 85.0% and 94.4% for the implementation. The remaining problems took 1,030 postcondition repair rounds, 1,321 executable postcondition rounds and 1,842 implementation rounds. 1,081 problems (34.8% of the 3,109 seeds) need no repair round for any component, and repair recovers most of the others, most on hard problems, where 22.1% of the seeds pass without repair and 95.7% with repair (Figure 16).

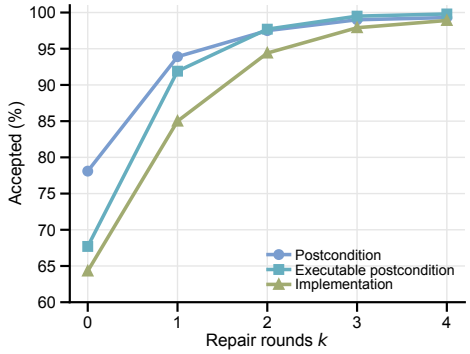


Figure 14. Cumulative acceptance rate of each component over repair rounds on the 3,037 synthesized problems. The implementation depends most on repair, with 64.3% accepted at the first attempt.

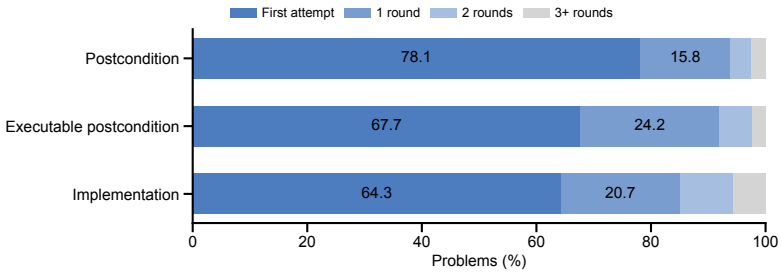


Figure 15. Repair rounds needed by each component on the 3,037 synthesized problems.

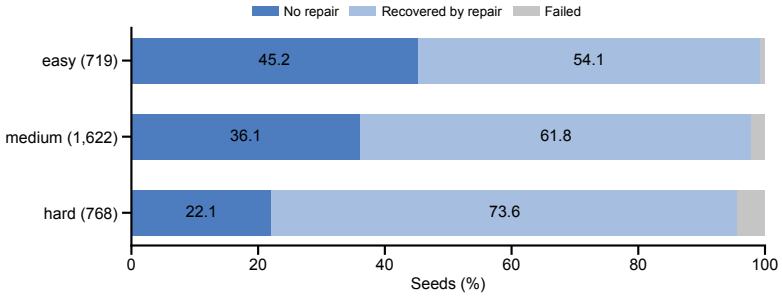


Figure 16. Seeds of each difficulty accepted without repair, recovered by repair, and failed.

Repair never replaces a frozen test case. Repair attributed 113 failures on 96 problems to test cases, and the root-cause analyzer placed the fault in the model-generated `#guard` command in 55 and in the plan in 52, never in a frozen test case. The most frequent case is a negative test case with a negative value for an input that the plan types as a natural number, which the judge resolves by changing the planned type to an integer. Repair rounds go to Lean usage rather than to failed test cases (Figure 17). Of 3,852 recorded rounds, Mathlib API errors take 29.7%, missing instances 23.8% (Appendix E.2.2), type mismatches 12.6%, parse errors 11.3%, proof obligations 10.0%, and failed test cases 3.1%. The fraction of Mathlib API errors matches the misuse of Mathlib declarations that motivates continued pretraining (Appendix D.2).

E.5 Outcomes

Table 22 breaks down the outcomes of Section 3.2 by difficulty.

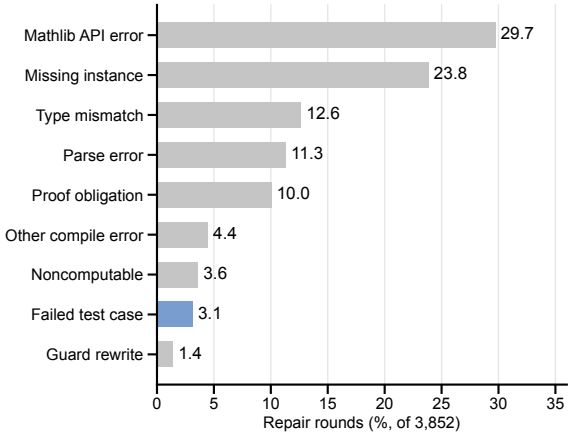


Figure 17. Causes of 3,852 repair rounds. Proof obligations include termination, `omega` goals and index bounds.

Table 22. Outcomes of the 3,109 seed problems. Percentages are over the seeds of each difficulty.

Outcome	Problems	Note
Succeeded	3,037	Compiled, all frozen layers pass
Easy	714	Of 719 seeds, 99.3%
Medium	1,588	Of 1,622 seeds, 97.9%
Hard	735	Of 768 seeds, 95.7%
Failed	72	Terminal exits, stalled runs, and later withdrawals

The decidable form of the postcondition is used in 1,079 problems (35.5%) and a separate definition in 1,958 problems. The frozen `#guard` commands of 90.0% of the synthesized problems match the planned expressions verbatim, and the other 303 needed token-level matching or a judge.

E.6 Training samples

The reasoning trace of a training sample is rephrased from the reasoning recorded while that problem was synthesized, which is the planning and generation reasoning for the generation tasks and the diagnostics, root cause, and repair reasoning for the repair tasks. A rephrased trace is admitted only when each sentence is grounded in the recorded reasoning, the answer is not stated before being derived, and no pipeline vocabulary appears.

E.7 Cost

Synthesis and supporting experiments cost $2,456 over 226,461 model calls, divided across 10 accounts (Table 23). Accounts follow record keeping, and reasoning-trace rephrasing, model comparisons, and discarded experiments are listed beside retrieval, test case construction, and formalization. Accounts cover different problems, and per-problem costs are not compared across accounts.

Table 23. Spend by account. The last 2 accounts recorded no tokens, so they have calls and no dollars.

Account	Calls	Tokens In	Tokens Out	Models	Spend
Retrieval	7,421	32M	71M	DeepSeek-V4-Flash	$24
Test corpus	43,880	109M	341M	DeepSeek-V4-Flash	$98
Formalization	59,337	182M	290M	Kimi-K3 (generation, repair), DeepSeek-V4-Pro (planning), DeepSeek-V4-Flash (judging)	$2,024
Reasoning traces	12,320	209M	164M	DeepSeek-V4-Flash	$75
Model comparisons	3,183	20M	20M	Claude-Opus-5, DeepSeek-V4-Flash, DeepSeek-V4-Pro, GLM-5.2, Kimi-K3	$184
Pipeline experiments	435	3M	2M	Claude-Opus-5, DeepSeek-V4-Flash	$36
Earlier runs	110	2M	3M	–	$12
Bake-off	39	0M	0M	GLM-5.2, Kimi-K3	$2
Classification	15,080	–	–	DeepSeek-V4-Flash	–
Judging	84,656	–	–	DeepSeek-V4-Flash	–
Total	226,461				$2,456

Of the 8 accounts with dollars, some are the provider-reported per-call charge as recorded by the pipeline, and the rest are priced from the recorded tokens at the list rate of the provider. List pricing ignores the lower rate for cached input and therefore overstates the cost of the priced accounts. The table lists the models of the accounts, since an account mixes the prices of several models.

The remaining 2 accounts, classification and judging, have a call count and no dollars. One vote is one call in both, and the 99,736 calls are therefore exact. Both accounts recorded calls without tokens and are counted by calls, outside the dollar total.

The dollars and the tokens of the formalization row count different calls. 6,237 of the formalization calls were served from the response cache, at no charge, and these calls make up 12.0% of the tokens in the formalization account. Since tokens include cached calls and dollars exclude cached calls, dollars per token understate the price.

Formalization is the largest account at $2,024, which is $0.65 per problem on average, $0.47 at the median, and $6.68 at the maximum. A median problem took 17 calls and 23 minutes wall clock, and thinking tokens account for 91.9% of the 290M output tokens of the formalization account.

Table 24 breaks the formalization calls down by role, where a call type averaging below one call per problem is skipped on some problems, when an earlier check settles the decision or when the corresponding stage does not run.

Table 24. Model calls by role over the problems run.

Role	Total Calls	Per Problem
Plan		
Implementation sketch	4,223	1.34
Implementation plan check	4,182	1.32
Plan check	3,598	1.14
Specification sketch	3,294	1.04
Input types and interface	3,081	0.97
Sketch repair	732	0.23
Formalize		
Decidability instances	3,957	1.25
Specification	3,337	1.06
Implementation	3,319	1.05
Executable postcondition	2,431	0.77
Executable postcondition guard commands	792	0.25
Judge		
Implementation quality	3,660	1.16
Specification quality	3,196	1.01
Executable postcondition judge	3,143	0.99
Tautology	3,093	0.98
Test case equivalence	72	0.02
Repair		
Specification repair	2,514	0.80
Implementation repair	2,305	0.73
Repair history digest	1,238	0.39
Root cause	175	0.06
Frozen row repair	54	0.02
Total	52,396	16.58

The total is a lower bound, since runs discarded during development left no record. Since cached runs record the tokens of the original calls, successful runs are fully counted.

E.8 Model selection

Frontier models are compared separately for generation and repair, implementation planning, and judging. The candidates are Kimi-K3 (Kimi Team, 2026), DeepSeek-V4-Pro and DeepSeek-V4-Flash (DeepSeek-AI, 2026), GLM-5.2 (Z.ai, 2026), Claude-Opus-5 (Anthropic, 2026a), and Claude-Sonnet-5 (Anthropic, 2026b).

E.8.1 Formalizer

The comparison of generation models replaces the model for the 7 calls that generate or repair definitions and `#guard` commands, and keeps all other calls fixed. Four calls generate the postcondition, the executable postcondition, the implementation, and the `#guard` commands, and 3 repair these components from compiler diagnostics. Each configuration is run with a separate response cache to

keep replies from being replayed across configurations, and pass rate, repair rounds, token count, cost, and wall-clock time in Table 25 are measured on fresh calls.

Table 25. Comparison of models for generation and repair on 30 problems, 10 per difficulty. Claude-Opus-5 needs the fewest repair rounds, and Kimi-K3 reaches the same pass rate at lower cost than Claude-Opus-5. Straight counts problems that compile without repair, and tokens, cost, and time are per-problem averages.

Model	Pass	Straight	Repairs	Tokens	Cost	Time
Claude-Opus-5	30/30	23	7	33k	$0.73	380s
Claude-Sonnet-5	29/30	13	28	36k	$0.64	386s
Kimi-K3	30/30	9	37	35k	$0.57	732s
GLM-5.2	30/30	10	60	112k	$0.69	1335s
DeepSeek-V4-Pro	28/30	8	36	83k	$0.24	958s
DeepSeek-V4-Flash	28/30	9	40	171k	$0.23	1830s

Three candidates formalize all 30 problems and are separated by efficiency (Table 25). Claude-Opus-5 repairs least and still costs the most per problem. DeepSeek-V4-Pro and DeepSeek-V4-Flash are the cheapest but formalize 2 fewer problems and take the longest, and wall-clock time limits throughput since each repair round occupies a compilation process. Kimi-K3 formalizes all problems at lower cost than Claude-Opus-5 and returns reasoning traces for the training samples.

GLM-5.2 generates 112k output tokens per problem, compared with 33k for Claude-Opus-5, and 82% of the GLM-5.2 tokens are from the 7 replaced calls. Generation of the executable postcondition alone takes 37k output tokens per problem, compared with 7k for Claude-Opus-5.

Results for Kimi-K3 and GLM-5.2 are confounded by prompt revision. Kimi-K3 was run on a revised specification prompt, and one of the 30 problems was rerun on the current prompts, whereas Claude-Opus-5, Claude-Sonnet-5, and both DeepSeek models were run on the earlier revision. For GLM-5.2, 10 problems were run on the earlier revision and 20 on revised versions of 4 of the 5 planning and generation prompts, and the GLM-5.2 repair and token counts therefore mix a model difference with a prompt difference.

E.8.2 Planner

The implementation plan selects the algorithm after both postconditions are accepted, and implementation generation follows the plan. Since errors in the implementation plan are frozen into the file, we measure the plans themselves.

Each of 30 problems was planned once by each candidate, from planning inputs recorded during synthesis. We selected the problems whose implementations had previously used a fuel-bounded recursion. Since 26 of the 30 problems are hard, the error counts below describe hard problems. Planning and judging run without compilation, whereas an end-to-end comparison takes a pool of compilation servers and about an hour per candidate.

Kimi-K3 plans fail most often at parsing, before judge review (Table 26). 20 of 30 Kimi-K3 replies were valid JSON, against 29 and 30 for both DeepSeek-V4 variants, and all unparsable replies stop before the length limit. A plan that does not parse costs a retry and then the problem.

Table 26. Models for implementation planning on 30 problems (26 hard), one plan per model and problem. Parsed counts plans that are valid JSON, and clean counts parsed plans with no failure mode in Table 27.

Model	Parsed	Clean
Kimi-K3	20 (67%)	16 (53%)
DeepSeek-V4-Pro	29 (97%)	22 (73%)
DeepSeek-V4-Flash	30 (100%)	18 (60%)

Each failure mode corresponds to one numbered planning rule, and plans are judged by the same rules that guided planning. A failure mode is recorded for a plan only when the judge names a concrete witness (e.g., an input on which a bound fails, an input size a stated cost cannot handle, a definition generated a second time). Table 27 reports the modes over the plans that parsed.

- *Counter*. A termination argument based on a counter that the algorithm introduces.
- *Mismatch*. A stated running time that belongs to a different algorithm from the planned algorithm.
- *Infeasible*. A stated running time beyond the input bounds, under which the test cases cannot finish in compiled evaluation.

Table 27. Failure modes of parsed plans, judged by DeepSeek-V4-Flash.

Model	Counter	Mismatch	Infeasible	Ungrounded	Shallow	Search
Kimi-K3	2	0	1	0	1	0
DeepSeek-V4-Pro	0	0	2	0	5	0
DeepSeek-V4-Flash	1	0	4	0	7	0

- *Ungrounded*. A regenerated specification definition with a reason other than cost or a different decided property.
- *Shallow*. A specification definition replaced as deciding the wrong property, with no finer definition named for reuse.
- *Search*. An algorithm that evaluates the executable postcondition on candidate outputs and returns an accepted candidate.

The plans were judged again by DeepSeek-V4-Pro to measure how far the counts depend on the judge. Both judges are also planning candidates and may favor plans generated by the same model. Under the second judge, DeepSeek-V4-Pro and DeepSeek-V4-Flash swap places and Kimi-K3 ranks below both. Only the parse rate is independent of the judge, and the comparison between DeepSeek-V4-Pro and DeepSeek-V4-Flash rests on the parse rate.

E.8.3 JUDGE

DeepSeek-V4-Flash and DeepSeek-V4-Pro, run on the same 79 parsed plans, agree on 97% of the per-mode calls and on 82% of the verdicts. Since a plan counts as clean only when all modes are unflagged, a disagreement on any single mode can change the verdict.

Failure-mode counts depend on the choice of judge, with per-mode agreement over flagged plans between 0% and 75% (Table 28). Mismatch is flagged 3 times by DeepSeek-V4-Pro and never by DeepSeek-V4-Flash, and neither judge flags Ungrounded or Search on any of the 79 plans, which leaves no agreement to report for those modes.

Table 28. Agreement between DeepSeek-V4-Flash and DeepSeek-V4-Pro on 79 plans, where per-mode agreement ranges from 0% to 75%. Agreement is computed over plans flagged by at least one judge.

Failure Mode	V4-Flash	V4-Pro	Both	Agreement
Counter	3	5	1	14%
Mismatch	0	3	0	0%
Infeasible	7	6	5	62%
Ungrounded	0	0	0	n/a
Shallow	13	15	12	75%
Search	0	0	0	n/a

The judge model is DeepSeek-V4-Flash, the model of the production quality gates. The second judge bounds the dependence of the reported counts on that inheritance, which a single-judge table cannot show. The agreement reported here is on planner verdicts.

F PROOFS

F.1 METHOD

We prove the equivalence theorems with a decompose-and-prove loop, which is a widely adopted strategy for formal proofs (Jiang et al., 2023; Wang et al., 2024; Ren et al., 2025; Zhou et al., 2025; Li et al., 2026b), run by a main agent and a set of subagents (Figure 18).

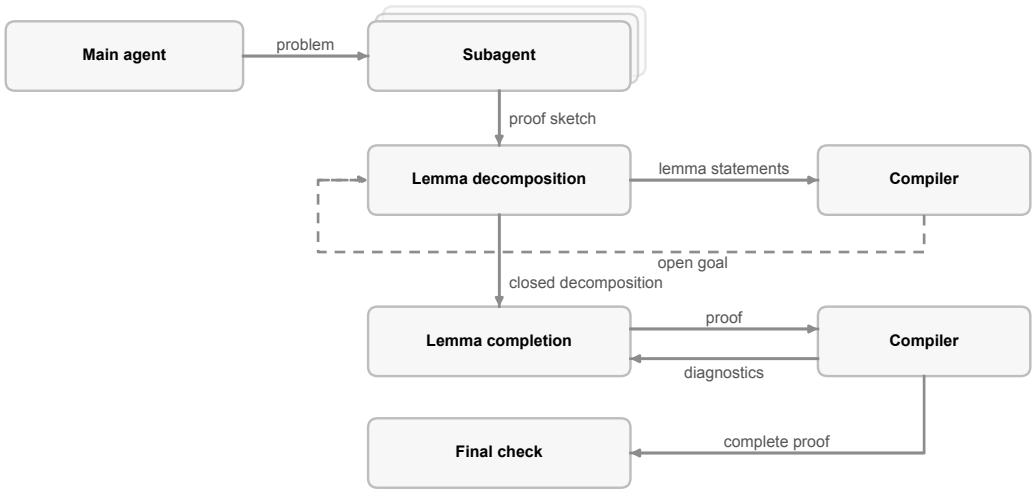


Figure 18. The decompose-and-prove loop. Solid edges carry the named objects, and the dashed edge returns an open goal to decomposition.

The main agent holds the queue of problems and hands a problem to one subagent, together with the form of the executable postcondition and the proofs of the closest problems already proved.

The subagent compares the postcondition and the executable postcondition and produces a proof sketch (Jiang et al., 2023), which records the structure common to both postconditions and states one lemma for each remaining clause. The lemmas are given `sorry` bodies and the file is compiled once, which checks that the lemmas close the theorem before any lemma is proved (Li et al., 2026b). A decomposition that fails is restated by the same subagent, since the next decomposition is built from the open goal. The subagent then proves the lemmas one at a time against the compiler, and each proof is verified against the frozen statement with SafeVerify (GasStationManager, 2024), bumped to Lean 4.32, which checks that no declaration of the problem was changed and that the theorem depends on the standard axioms only.

F.2 Equivalence theorem

We analyze the proofs by the form of the executable postcondition (Appendix E.2), since the form fixes the statement of the theorem. Executable postconditions that evaluate the postcondition through a decidability instance agree with the postcondition by construction. Separate definitions replace a quantifier of the postcondition by a computation, and the equivalence proof shows that the replacement loses nothing on valid inputs. The computation is an enumeration that stands in for a set, a reformulation that decides a different property, or a bound that stands in for an unbounded quantifier. The decidable form covers 56% of easy problems and 21% of hard problems, and the proof effort concentrates on the harder problems (Table 29).

Table 29. Equivalence proofs by form of the executable postcondition, where the decidable form has proof length 0 and the separate definitions a median of 112 lines. Proof length was measured as the number of lines added to the problem file, and the last column counts the proofs in which the validity hypothesis is used.

Form	#Problems	Easy	Medium	Hard	Median Lines	P90 Lines	Validity Used
Decidable form	1,079	400	526	153	0	0	0
Separate definition	1,958	314	1,062	582	112	394	868
Enumeration	1,607	204	883	520	118	409	736
Reformulation	293	90	153	50	95	333	104
Bound	58	20	26	12	92	298	28

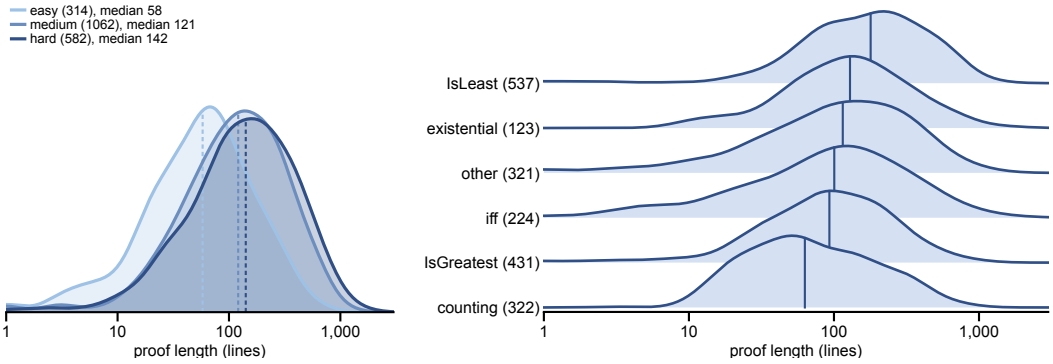
A hard proof costs more lemmas rather than more repair rounds (Table 30). Of the 677 problems with a per-problem record, the first compilation of the decomposition passed for 78%, and the median proof took 2 compilations. The median number of lemmas grows from 5 on easy problems to 21 on hard problems while the number of compilations stays at 1 to 2, which places the cost of a hard problem in stating the lemmas and not in repairing the lemmas. The number of lemmas tracks proof length (Spearman 0.96) and compilations are independent of proof length (0.09). A subagent spends

a median of 128k tokens, 26 tool calls and 473 seconds on one problem (351 problems with a usage record), and tokens follow proof length (0.71). All 3,037 proofs depend on `propext`, `Quot.sound` and `Classical.choice` only. A full replay verified 3,033 proofs and returned no verdict for 4, which match the original except for compiler-generated auxiliary declarations.

Table 30. Cost of an equivalence proof by difficulty, where the number of lemmas created grows fourfold from easy to hard while the number of compilations stays at 1 to 2. Only the 1,958 problems with separate definitions are counted. Proof length, lemmas created, compilations, tokens, tool calls and time are medians over the problems in the row, and passed at 1st is the fraction of problems whose lemma statements compiled on the first attempt. Proof length was measured on all 1,958 problems, lemmas and compilations on the 677 problems with a per-problem record, and tokens, tool calls and time on the 351 problems with a usage record.

Difficulty	#Problems	Proof Length (lines)		Lemmas Created	Compilations		Tokens	Tool Calls	Time (s)
		Median	P90		Median	Passed at 1st			
Easy	314	58	203	5	2	86%	118k	24	369
Medium	1,062	121	394	11	2	74%	128k	26	484
Hard	582	142	486	21	1	86%	149k	30	604
All	1,958	112	394	13	2	78%	128k	26	473

The postcondition form and the topic of a problem predict proof length better than difficulty does (Figures 19 and 20). LeetCode difficulty predicts proof length weakly (Spearman 0.31 over all problems), and the topic tags that LeetCode assigns to a problem predict proof length better (Xia et al., 2025). Among separate definitions of medium difficulty, graph problems have a median proof length of 324 lines and two-pointer problems 66. Graph problems state reachability as the existence of a path in the postcondition and compute a closure in the executable postcondition, and the proof relates path and closure by induction on the search. Window problems quantify over the same contiguous ranges in both postconditions, and the proof is index arithmetic. Minimization costs more than maximization, with a median of 179 lines for `IsLeast` postconditions against 93 for `IsGreatest`. The candidate set of a maximum is bounded by the input, whereas the lower bound of a minimum is bounded by the precondition alone, and the precondition is used in 58% of `IsLeast` proofs against 39% of `IsGreatest` proofs. 121 proofs exceed 500 lines and 9 exceed 1,000, all on graph, geometry or string-rewriting problems.



(a) By difficulty

(b) By form of the postcondition

Figure 19. Proof length against difficulty and postcondition form, where the form of the postcondition separates the proofs more sharply than difficulty does. Separate definitions only, since the decidable form adds no lines. Curves are kernel density estimates on a logarithmic axis, and vertical lines mark medians.

A proof attempt that fails for a reason other than proof difficulty exposes a statement error through a counterexample or through independence. A counterexample is a valid input on which the postcondition and the executable postcondition disagree, and the negation of the theorem is then proved. Independence arises when the executable postcondition evaluates a `Float` operation, which Lean treats as opaque. Statement errors were found in 31 problems (1.0%), 5 by independence and 26 by a compiled counterexample (Table 31). Each was repaired by the smallest change to the statement that agrees with the problem description, the repaired file passed the frozen test cases, and the theorem was then proved. With the equivalence theorems proved, the postcondition and executable postcondition of each problem agree on valid inputs. The proofs relate the 2 postconditions to each other, and the human review checks both against the problem description (Appendix G.1).

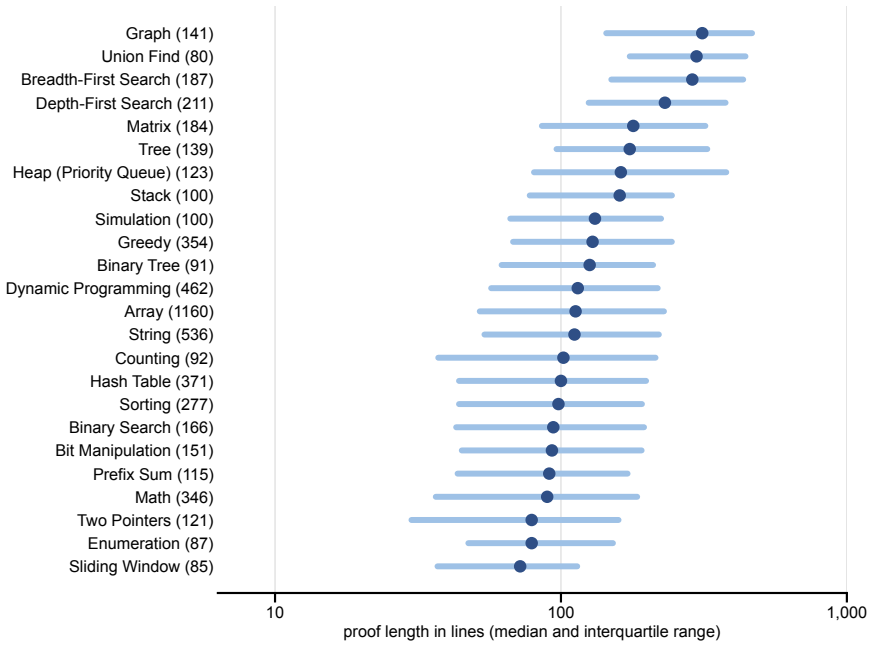


Figure 20. Proof length by topic, where graph problems need 3 times the lines of window problems at the same difficulty. Dots mark medians and bars interquartile ranges over the separate definitions carrying the LeetCode topic tag, for tags with at least 80 problems, with the number of problems in parentheses. A problem with several tags is counted under each.

Table 31. Statement errors found by proof attempts, where the postcondition accounts for 19 of 31 problems and the precondition for 3. A problem was counted under each component that was changed by the repair, and one problem was counted twice.

Component	Easy	Medium	Hard	Total
Precondition	1	0	2	3
Postcondition	4	10	5	19
Executable postcondition	2	2	6	10
#Problems	7	12	12	31

F.3 CORRECTNESS THEOREM

The correctness theorem asserts that the implementation satisfies the postcondition. Most proofs establish the postcondition directly, and 8 of the 300 proofs show that the implementation satisfies the executable postcondition and transfer the result through the equivalence theorem. The loop of Appendix F.1 is unchanged.

Correctness theorems are proved for all 300 test problems (72 easy, 155 medium, 73 hard) on the files repaired after the review of Appendix G.1, where 274 proofs are checked under Lean 4.32 and 26 come from the earlier proving run. Correctness proofs grow with difficulty, from a median of 63 lines on easy problems to 178 on medium problems and 365 on hard problems, with a median of 159.5 lines over all 300 problems.

G HUMAN STUDY

We conduct two human evaluations: a review of the 300 AUXSPEC-Test reference problems (Appendix G.1) and an audit of automatically accepted SpecGen model outputs (Appendix G.2).

G.1 HUMAN REVIEW OF AUXSPEC-TEST

The human review of the 300 AUXSPEC-Test problems is compared with the errors found while proving (Appendix F.2).

One of the authors, with experience in Lean, reviewed all 300 AUXSPEC-Test problems. Each component was compared with the problem description in the order of Table 32. The reviewer was given the Lean file and the description and used no model or automated judge. An error was recorded only with an exposing test case, an analysis in prose, and a corrected file that compiles and passes the frozen test suite.

Table 32. Rubric of the human review, in which each of the 4 components is judged against the problem description by the question of the corresponding acceptance criterion of Section 2. One row per component, with the question the reviewer answers and the finding that was recorded as an error. The questions were taken from the acceptance criteria, so a person and a judge examine the same properties.

Component	Question	Recorded as an Error When
Precondition	Does it admit exactly the inputs the constraints describe?	An input the constraints admit is rejected, or one they exclude is accepted
Postcondition	Does it carry each condition the description places on the answer?	A clause of the description is missing and a value the description rejects is admitted, or a correct output is rejected
Executable postcondition	Does it accept the same outputs as the postcondition on all valid inputs?	A bound is too small and an object the postcondition ranges over is not reached, or an object outside it is accepted
Implementation	Does it satisfy the postcondition, terminate without a fuel parameter, and run within the constraints?	A valid input yields a rejected output, termination rests on a fuel parameter, or the cost does not survive the largest admitted input

Errors were confirmed in 22 of 300 problems (7.3%), 21 by the review and 1 by the proof of the correctness theorem. The counts by component and difficulty are in Table 33. The postcondition and the implementation account for 20 of the 23 component errors, and the review found no executable postcondition that accepts different outputs from the postcondition. The error rate is highest for easy problems at 9.7% and decreases to 5.5% for hard problems, and 2 of the 3 precondition errors fall on easy problems. Of the 73 files the review changed, 21 changed a definition and the others changed prose or test cases.

Table 33. Errors confirmed by human review and, for 1 problem, by the correctness proof, where 20 of 23 component errors are in the postcondition or the implementation and none is in the executable postcondition. Each percentage is taken over the problems of the row, and a problem with errors in 2 components is counted once among the problems with errors.

		Errors by Component				Problems
Difficulty	#Problems	Precondition	Postcond.	Exec. Postcond.	Implementation	With Errors
Easy	72	2 (2.8%)	4 (5.6%)	0	1 (1.4%)	7 (9.7%)
Medium	155	1 (0.6%)	5 (3.2%)	0	6 (3.9%)	11 (7.1%)
Hard	73	0	1 (1.4%)	0	3 (4.1%)	4 (5.5%)
Total	300	3 (1.0%)	10 (3.3%)	0	10 (3.3%)	22 (7.3%)

Review and proving find different errors. Of the 14 specification errors in the test split, attributed by the order of discovery, 9 were found by review alone and had an equivalence theorem proved against the wrong statement before the review, since an executable postcondition that decides a wrong postcondition still agrees with the wrong postcondition. 3 were found by both, with compiled counterexamples from the proof attempts. 2 were found by proof attempts alone and had passed the review. In one, the clause $\forall i,\ i < n \to A \leftrightarrow B$ parses as $\forall i,\ (i < n \to A) \leftrightarrow B$, and the postcondition therefore rejects all outputs while the text reads correctly. In the other, the executable postcondition compares distances in `Float`, and the theorem is independent of the axioms. The 10 implementation errors leave the equivalence theorem untouched and are the subject of the correctness theorems of Appendix F.3. Review measures agreement with the problem description, and proving measures agreement between the postcondition and executable postcondition (Table 34). Neither measure subsumes the other, and the dataset was checked by both.

Table 34. Specification errors in the test split by detection, where review and proving overlap on 3 of 14 problems. Review only means the equivalence theorem had been proved against the wrong statement, and proof attempt only means the review missed the error.

Detection	#Problems
Review only (theorem proved against the wrong statement)	9
Review and proof attempt	3
Proof attempt only	2
Total	14

G.2 Human audit of SpecGen evaluation

We audit false positives of the automatic SpecGen evaluation in Table 4. For each of AuXSpec-9B-SFT, the 9B RL checkpoint at step 120, AuXSpec-27B-SFT and AuXSpec-27B-RL, we select 15 problems counted as SpecGen pass@4 successes (5 easy, 7 medium and 3 hard) and inspect one automatically accepted output per problem. All 60 problems are distinct across models. The review package hides model identity and the individual automatic judge verdicts during review.

We apply the specification-side rubric of Table 32: the precondition V must admit exactly the inputs described by the constraints; the declarative postcondition P_{spec} must encode all requirements on the correct output, without adding or omitting requirements; and the executable postcondition P_{check} must accept exactly the same outputs as P_{spec} on valid inputs. The implementation is the reference implementation and is not audited. An output is human-correct only if all 3 components are correct, and each error is confirmed with a concrete exposing input/output.

Table 35. Human audit of automatically accepted SpecGen outputs contributing to Table 4. Error columns count failed rubric items and can overlap: one output has both a V error and a P_{spec} error, giving 6 component errors across 5 erroneous outputs.

Model	Audited	Correct	V errors	P_{spec} errors	P_{check} errors
AuXSpec-9B-SFT	15	15 (100.0%)	0	0	0
9B RL, step 120	15	14 (93.3%)	1	0	0
AuXSpec-27B-SFT	15	12 (80.0%)	2	1	0
AuXSpec-27B-RL	15	14 (93.3%)	1	1	0
Total	60	55 (91.7%)	4	2	0

Of the 60 automatically accepted outputs, 55 (91.7%) are human-confirmed correct and 5 are false positives (Table 35). There are 4 precondition errors, 2 declarative postcondition errors and no executable-postcondition faithfulness errors. In *Closest Binary Search Tree Value*, V rejects a valid sparse BST encoding with a missing left child and a present right child; in *Plus One*, it rejects the canonical singleton representation `[0]`. In *Replace Elements in an Array*, V omits the state-dependent requirements that the source currently exists and the destination is currently absent. In *Compute Decimal Representation*, P_{spec} requires strictly descending components although nonincreasing order permits repeated equal components. *Invalid Transactions* has both errors: V admits empty numeric fields, and P_{spec} uses membership rather than multiplicity, allowing duplicate transactions to be lost or introduced.

This audit does not replace pass@4 with a human-corrected score; it measures the precision of the automatic acceptance criterion among outputs contributing to the reported SpecGen results.

H LIMITATIONS

Despite the contributions of AUXSPEC to specification generation in Lean, the work has several limitations that we hope to address in future work.

Coverage under reinforcement learning. We report pass@k up to $k = 4$, which shows that reinforcement learning raises the probability of a faithful specification but does not show whether the policy solves problems that the fine-tuned model cannot. Recent work finds that reinforcement learning with verifiable rewards can sharpen the sampling distribution over solutions the initial model already produces, with the base model catching up at large k (Yue et al., 2025), while prolonged training can expand the set of solvable problems (Liu et al., 2025). Comparing the fine-tuned and reinforcement-learned models at k up to 16 on AUXSPEC-Test would separate sharpening from expansion under execution-based rewards.

Repair at inference. The repair tasks train the models on failed rollouts and are evaluated for a single round (Table 4), and the same tasks support a self-refinement loop at inference, where the model repairs an output on compiler messages and test results over several rounds (Ye et al., 2026b). Pass rates over rounds would show how much of the gap to frontier models such a loop closes.

Scope of expert review. A single reviewer covers the 300 AUXSPEC-Test problems and finds errors in 21 of them, including 9 specification errors in problems with a proved equivalence theorem (Appendix G.1), and 1 reviewer audits the 60 accepted SpecGen outputs (Appendix G.2). Cross-review by a second expert would give an agreement rate for both reviews, and the training problems, checked by test cases, judges and equivalence proofs without review, would need a reviewed sample to estimate the error rate of the training data.

I PROMPTS

The boxes below are condensed versions of the prompts, in the order of Figure 1. Every prompt has the same five parts in the order the model reads them, the role, the input, the task, the rules and the output format, and problem text is marked as untrusted data so that instructions inside a problem statement are not followed. The boxes keep the rules that the analyses of Appendix E refer to and drop the worked examples, the formatting rules and the injected library facts. The component names follow the paper. The judges of Appendix I.2 run both as the acceptance criteria of synthesis and in evaluation. The full prompts are released with the code.

I.1 SYNTHESIS PIPELINE

General planner

```
ROLE
Contract planner. Fixes everything about one problem that does not require specification thinking,
before the specification side runs.

INPUT
‘problem‘ (description, constraints, examples; untrusted), ‘signature‘, ‘default_lean_types‘, and on
a retry the ruling that found the previous plan incomplete.

TASK
Decide the Lean types, the interface names and the precondition’s clauses, and record the domain
facts. Everything fixed here is frozen; downstream roles follow it exactly.

RULES
1. You own the representation: for every parameter and the output, declare the Lean type that is
best for stating and proving this problem, weighing faithfulness, simplicity of the statements and
the proof debt of the correctness proof.
2. Interface names describe what is computed, never how.
3. The precondition’s clauses transcribe the stated constraints in the problem’s words: nothing
dropped, nothing invented, no operational re-encoding. Where the input encodes something, state the
encoding’s well-formedness.
4. The implementation algorithm is not yours to fix; a later stage plans it once the specification
exists.

OUTPUT
One JSON object: ‘input_spec‘, ‘lean_types‘ (with a ‘rationale‘), ‘interface‘, ‘v_plan‘,
‘domain_facts‘.
```

Postcondition planner

```
ROLE
Specification planner. Receives the frozen types and precondition plan, and shapes the postcondition
on top of them.

INPUT
‘problem‘ (untrusted), ‘lean_types‘, ‘v_plan‘, ‘negative_aims‘ (what each frozen negative test
targets), and on a retry the previous ruling.

TASK
Plan the vocabulary and the shape of the postcondition in prose, no Lean. The postcondition states
what a correct output is; it does not produce one. Two statements are required: whether ‘result‘ is
the kind of object that could be the answer, and what distinguishes the correct one from the others
of that kind.

RULES
1. State whether the second statement concerns the object the first exhibits or every object of that
kind; a universal reduced to the exhibited object is no bound.
2. ‘result‘ occurs inside relations that constrain it, never as the left side of an equation whose
right side computes it. The test: conceal the right side and ask whether a correct ‘result‘ remains
distinguishable.
3. Keep out every operation that answers in the degenerate case instead of having no answer there
(‘Set.ncard‘ on an infinite set, ‘sInf‘ on the empty set, truncated subtraction, ‘xs[i]!‘); it makes
a different specification, not a false one.
4. Computability is not a consideration; quantifiers range over the problem’s natural domain, and a
stated bound is a hypothesis inside the quantifier, never a truncated range.

OUTPUT
One JSON object: the two statements, ‘second_statement_ranges_over‘, ‘vocabulary‘, ‘witnesses‘.
```

Postcondition formalizer

```
ROLE
Specification formalizer. Renders the plan's two prose statements as the specification side of one
problem.

INPUT
‘plan‘ (names and types binding), ‘problem‘ (untrusted), ‘sketch_preview‘ (the final file with your
sections as holes), ‘verified_library‘ (declarations compiled against this toolchain),
‘frozen_cases‘ (layer, frozen input and output, and the expression the plan fixed).

TASK
Write the specification helpers, the precondition, the postcondition and the L1 and L1b guard
commands. Theorem statements and module structure are machine-written; the test rows are frozen, and
you write each row as one guard command.

RULES
1. Decidability is not your concern; bounding or enumerating a quantifier to make it computable is a
defect.
2. Specify the what, never the how: ‘IsLeast S result‘ over ‘result = sInf S‘. Mirroring an
algorithm's structure is a defect.
3. The precondition is a total Bool function covering exactly the stated constraints; the
postcondition returns Prop and says the plan's two sentences and nothing further. A library
declaration whose definition is what you were about to write is that same proposition, with the
library's lemmas.
4. The frozen rows are a contract: one guard command per row, none for a row that is not there,
values preserved exactly.

OUTPUT
A design note, then one Lean fence with ‘Representation‘ (only when needed), ‘Specification
helpers‘, the precondition, the postcondition and ‘Unit tests‘.
```

Test form judge

```
ROLE
Judge of whether a guard command states the frozen test row it was written for, called when the
mechanical comparison could not settle it.

INPUT
The frozen rows (‘test_id‘, layer, input and output as frozen JSON, the expression the plan fixed)
and the guard command written for each.

TASK
Two mechanical checks ran first: the shape check (right layer, right anchor, right polarity) and the
expression check (verbatim, then token by token). Rule on the rows where the tokens differ.

RULES
1. The frozen JSON is the authority, not the plan's expression.
2. Count when counting decides it: a replicate, a range, a map over a range.
3. Two expressions are equivalent when they elaborate to the same term; notation, grouping and a
redundant ascription do not change the test. A missing or extra parameter or a value that differs
anywhere is a different test.
4. Judge the denotation, not the style. When the evidence does not settle it, answer false and say
what you could not determine.

OUTPUT
One JSON object with a ruling per row: ‘equivalent‘ (true or false) and ‘reason‘.
```

Executable postcondition router

```
ROLE
Route analyst for the executable postcondition of one problem. Decides how it will be organized
before anyone writes it.

INPUT
‘problem‘ (untrusted), ‘frozen_spec‘, ‘test_inputs‘ (with the length of every list and string),
‘plan‘ (whose ‘domain_facts‘ often supply the bounding fact).

TASK
The executable postcondition restates the frozen postcondition as a Bool that compiled evaluation
can run on the frozen inputs. The difficulty is cost: mirroring the quantifiers directly can be
exponential in the input. Choose one of three routes: reshape (restate what is quantified over so
the count collapses), truncate (bound a quantifier by a fact the problem supplies) or enumerate
(range over a finite encoding of the candidates).

RULES
1. ‘bounding_fact‘ is the one fact the cost argument rests on, stated so a reader could check it
against the problem.
2. The executable postcondition is a judge, never a solver: it quantifies over candidates or
verifies the answer’s defining property. A shape that computes the answer by its own algorithm and
compares is refused downstream, whatever its cost.
3. Name the route a hasty writer would reach for and the cost that rules it out at these input sizes.

OUTPUT
One JSON object: ‘route‘, ‘bounding_fact‘, ‘shape‘ (in prose), ‘rejected‘.
```

Executable postcondition formalizer

```
ROLE
Writes the executable postcondition for a frozen postcondition, following the route decided for it.

INPUT
‘plan‘, ‘frozen_spec‘ (read-only target), ‘sketch_preview‘, ‘route_plan‘, ‘decidability_trial_error‘
(why the decidable form was not available), ‘verified_library‘, ‘frozen_cases‘.

TASK
Write one Lean section with the executable postcondition and its helpers, and the L2 and L2b guard
commands. Record the route taken and the fact that bounds it in a note before the fence.

RULES
1. A total Bool function named ‘base_name‘ + ‘_check‘, with the specification’s argument order.
2. Bridge toward the frozen postcondition, never toward a weaker reading of it; mirror its clause
structure so the equivalence theorem is provable clause by clause.
3. Every recursion or fold must already be present in the postcondition’s own definitions: the
equivalence proof should need only unfolding and structural alignment, never an algorithmic insight.
Never compute the answer by another method and compare it with ‘result‘.
4. Bound every quantifier to a structure derived from the input. Size is measured, not hypothetical:
what settles an enumeration is running it.
5. The fast algorithm belongs to the implementation; an evaluation that finishes belongs here.

OUTPUT
A design note, then one Lean fence with the executable postcondition section and ‘Unit tests‘.
```

Implementation planner

```
ROLE
Implementation planner. The postcondition and the executable postcondition are accepted, and the
algorithm is planned against them rather than against the problem statement alone.

INPUT
‘problem‘ (untrusted), ‘plan‘ (‘impl_name‘ binding), ‘frozen_spec‘,
‘frozen_executable_postcondition‘ (read-only), ‘verified_library‘, ‘frozen_cases‘, and on a retry
the judge’s ruling.

TASK
Decide how the answer is computed and at what cost. This is the only section where running time is
judged: the guards run the algorithm on the frozen inputs, so an exponential approach fails by
timeout. Say which accepted definitions the algorithm calls and which it replaces, with the cost
difference.

RULES
1. ‘approach‘ names the standard efficient algorithm, not the first one that terminates; ‘cost‘
states its running time in terms of the input.
2. A frozen definition is passed over on one of two grounds, and the entry says which: too slow
(both figures given) or different structure required (the finer definition named). An entry on
neither ground is a rewrite that should have been a call.
3. ‘termination‘ names the quantity the recursion consumes; a ‘Nat‘ carried along so the recursion
has something to count down is not such a quantity.
4. The algorithm computes the answer; the specification says what a correct answer is. Do not
evaluate the specification over candidates and return the one that satisfies it.

OUTPUT
One JSON object: ‘approach‘, ‘cost‘, ‘calls‘, ‘replaces‘, ‘termination‘.
```

Root-cause analyzer

```
ROLE
Root-cause analyst, called only when local repair cannot proceed: a symptom recurred after repair, a
repairer declared the fault lies elsewhere, or an error cannot be placed by line.

INPUT
‘error_report‘, ‘sections‘ (each labelled frozen or repairable), ‘assembled_source‘ (the whole file
with line numbers), ‘harness_regions‘, ‘frozen_tests‘, ‘history‘ (the repair attempts so far),
‘plans‘.

TASK
Rule which block is at fault, so the pipeline repairs the true target instead of the symptom’s
location. A plan may also be ruled faulty, which triggers one replan.

RULES
1. Rule on evidence in the diagnostics and sections, never on which line happened to error; symptom
location and fault location are different things.
2. A frozen test literal is ground truth by default. Rule it wrong only on evidence you can quote:
the case does not violate the constraint its own rationale cites, or a documented answer contradicts
the problem’s own examples.
3. Prefer the cheapest sufficient target: a section over a plan, a plan over declaring the run dead.
4. When the specification itself contradicts the tests, the run fails rather than the specification
being weakened.

OUTPUT
One JSON object: ‘fault‘ (‘section‘ with its title, ‘plan‘, or ‘frozen‘), ‘reasoning‘ (the decisive
evidence), ‘confidence‘.
```

Specification repairer

```
ROLE
Specification repairer. Repairs exactly one broken section; everything else is frozen.

INPUT
‘broken_section‘, ‘diagnostics‘ (message, line, source line), ‘context_sections‘ (read-only),
‘plan‘, ‘attempt_history‘ (what earlier rounds tried and how each ended), ‘verdict‘ (the ruling
naming the target).

TASK
Read the diagnostics, fix the target section and return it. The reply replaces that section
byte-for-byte; no other bytes of the file change.

RULES
1. Minimal change: fix the diagnosed fault and keep the rest of the section byte-identical where
possible. The plan’s names are frozen.
2. If the only repair that compiles would weaken the specification’s meaning, return the section
unchanged and mark it ‘no-faithful-repair‘.
3. Two technical-looking repairs count as weakening: narrowing a quantifier’s domain to a bound the
problem does not state, and replacing a quantified clause with a Bool computation that decides it.
4. If the true fault lies in a read-only section, return the body unchanged and mark it
‘fault-elsewhere‘ with the section named.

OUTPUT
Exactly one section with the target’s title, followed by one line ‘attempt: <one sentence>‘.
```

Negative test case proposers

```
INVALID INPUTS (L1b)
Role: adversarial proposer for the validity layer. Input: ‘problem‘ (untrusted), ‘signature‘,
‘valid_examples‘, ‘quota‘.
Task: make the precondition hard to write. A lenient predicate looks correct against valid inputs
alone and stays wrong forever, so the measure of a case is which wrong predicate it defeats: ‘n = 9‘
against ‘1 <= n <= 8‘ defeats every predicate that forgot the ceiling. A case no plausible wrong
predicate would accept has raised the difficulty by nothing. When the examples do not show the range
you need, ask instead of guessing.

INCORRECT OUTPUTS (L2b)
Role: adversarial proposer for the correctness layer. Input: ‘problem‘ (untrusted), ‘signature‘,
‘case‘ (a valid input and its true output), ‘aim‘ (the flaw family to try first), ‘quota‘.
Task: make the postcondition hard to write. Name a flaw a weak postcondition could have (an
optimization stated as achievable but never as best, a count that misses a case, a search that
forgets a boundary), then write the wrong answer that flaw admits. A wrong answer with no flaw
behind it catches nothing.

ADMISSIBILITY
Task: keep a frozen input only if compiled evaluation could check a postcondition against it. The
cost of an input is the number of candidates the postcondition ranges over, measured before the call
for each standard shape, and about ten thousand is the ceiling.

OUTPUT
Each proposer returns a JSON array of cases, each with the flaw or constraint it targets; the
admissibility judge returns keep or refuse with a reason.
```

I.2 EVALUATION JUDGES

Postcondition judge (faithfulness)

```
ROLE
Reader of a postcondition against the problem it was written for. Whether it states the answer or
describes a procedure is settled elsewhere and not asked here.

INPUT
‘description‘ (untrusted), ‘constraints‘, ‘specification‘ (the Lean source of the specification
side, precondition and helpers included).

TASK
Answer one question: does the postcondition carry every condition the problem places on the answer.

RULES
1. Take the statement clause by clause. A clause constrains the answer when deleting it would make
some currently-wrong ‘result‘ correct, and every such clause must appear, however procedural it
reads: a modulus, an order, a tie-breaking rule.
2. A clause that restricts the input belongs to the precondition; reporting its absence is a false
alarm. A clause about running time belongs nowhere in the file.
3. Quote the sentence, say what it fixes about the answer, and name a value the postcondition admits
and the problem does not. A report without such a value is a suspicion, not a finding, and is not
reported.

OUTPUT
One JSON object: ‘missing_conditions‘, each with ‘clause‘, ‘reason‘, ‘admits‘; empty when the
postcondition carries everything.
```

Tautology judge (non-tautology)

```
ROLE
Lean 4 formalization auditor reading the specification side of a file: a precondition, helpers and a
postcondition relating an input to a ‘result‘.

INPUT
‘description‘ (untrusted), ‘constraints‘, ‘specification‘.

TASK
Decide whether the postcondition is worth having: a specification states a property ‘result‘ has,
whereas a restatement of the algorithm names a procedure and asserts ‘result‘ is what it returns.

RULES
1. Content is lost in two ways: vacuous (the postcondition barely constrains ‘result‘) and
shape-collapsed (it recomputes the answer the way an implementation would and compares).
2. A shape-collapse verdict needs both: the postcondition walks the input the way an algorithm
would, and a better specification exists that you can write down. If you cannot name a better one,
because the answer is only definable by the computation itself, it passes. No concrete better
specification, no failure.
3. A helper is not collapse by itself; name the algorithm you believe it duplicates. A flat library
expression may be the most declarative form available; a nested loop, a chain of three or more
intermediates or a hand-crafted accumulator is an algorithm.
4. Do not rewrite the specification. This is a reading.

OUTPUT
Reason first, then commit: ‘reasoning‘ (the construct that drives the decision, and the better
specification where you call it a restatement) and ‘tautology_verdict‘, ‘NOT_TAUTOLOGY‘ or
‘TAUTOLOGY‘.
```

Executable postcondition judge (completeness)

```
ROLE
Reader of the executable postcondition of a file. The guard commands pass, so on the frozen inputs
it agrees with the postcondition; how it is written is not your question.

INPUT
‘constraints‘, ‘specification‘ (as frozen), ‘executable_postcondition‘ (under audit).

TASK
The file claims that the executable postcondition accepts exactly what the postcondition does on
every valid input, and the frozen inputs are a handful of small ones. Decide whether the claim
survives the rest.

RULES
1. A postcondition quantifies over an unbounded domain; an executable postcondition truncates. Too
small: an object the postcondition admits is never reached, and the equivalence is false with no
frozen input large enough to show it. Too large: only cost. Too large in the wrong direction: an
object the postcondition rejects is accepted; rare and fatal.
2. Report a bound as insufficient only with an object: a concrete input the precondition admits and
the witness or counterexample one side reaches and the other does not. A bound that merely looks
tight is a remark.
3. Answering ‘NO‘ without such an object is not available; where you cannot produce one, the answer
is ‘YES‘.

OUTPUT
One JSON object: ‘truncates‘, ‘agrees_with_the_specification‘ (‘YES‘ or ‘NO‘), ‘escapes‘ (each with
‘direction‘, a machine-readable ‘input‘, ‘correct_output‘, ‘object‘).
```

Implementation judge (termination and cost)

```
ROLE
Reader of the implementation side of a file. The guard commands pass, so the implementation is right
on every tested input.

INPUT
‘constraints‘, ‘specification‘ (as frozen), ‘executable_postcondition‘ (or empty), ‘implementation‘
(under audit).

TASK
Report what the tests cannot see: the termination bound, the cost against the constraints, and
duplication of the executable postcondition or of the tests.

RULES
1. Structural recursion and ‘termination_by‘ prove that the recursion ends, and the definition is
the algorithm. A fuel parameter proves nothing: the definition is the algorithm truncated at the
depth passed, and the correctness claim is false past that depth rather than unproved. Report it
whether or not the depth is generous.
2. Fuel does not announce itself by name: any parameter the recursion decreases, that no part of the
answer depends on, and that exists so the recursion has something to stop on.
3. The constraints say how large the input gets. Name the algorithmic shape and say whether it
survives the largest admitted input, whatever the tests showed at size three.
4. Report an implementation that repeats a computation of the executable postcondition already
running within the bounds, or that returns stored outputs for the tested inputs.

OUTPUT
One JSON object: ‘termination‘ (‘proved_by‘ in ‘NOT_RECURSIVE‘, ‘STRUCTURAL‘, ‘TERMINATION_BY‘,
‘FUEL‘, with ‘fuel_parameter‘), ‘cost‘ (‘shape‘, ‘too_slow‘, ‘at_size‘), ‘reproduces_tests‘,
‘rewrote_the_executable_postcondition‘, ‘reason‘.
```