VERISCALE: ADVERSARIAL TEST-SUITE SCALING FOR FORMAL SPECIFICATION EVALUATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification can establish that generated code satisfies a formal specification, but this guarantee is meaningful only when the specification faithfully captures the intended behavior. Existing benchmarks evaluate LLM-generated specifications using concrete test cases, whose sparse coverage can allow flawed specifications to pass and thus systematically overestimate SpecGen capabilities. We present VeriScale, a framework for scaling test suites for formal specification evaluation. VeriScale expands candidate inputs through LLM-based seed generation and type-aware mutation, and synthesizes adversarial implementations that exploit weaknesses in generated specifications to produce challenging incorrect outputs. It further constructs compact test suites through boundary-preserving reduction and mutant-guided selection, reusing the adversarial implementations as mutants. Applied to VERINA, VeriScale produces VERINAPLUS, an approximately $83\times$ expanded benchmark, and VERINALITE, a compact $13\times$ variant. Across eight state-of-the-art LLMs, VERINAPLUS reduces the average Spec Score from 61.44 to 55.09, exposing approximately 12 additional task-level failures per model among 189 tasks. VERINALITE obtains an average score of 56.02, retaining 85.45% of the score reduction with only about 16% of the tests in VERINAPLUS. Code and data are provided in the supplementary material.

1 INTRODUCTION

Large language models (LLMs) increasingly generate not only programs (Chen et al., 2021; OpenAI et al., 2024), but also formal specifications and machine-checkable proofs of correctness (Sun et al., 2024; Ye et al., 2026a). However, formal verification provides only a relative guarantee: a proof establishes that an implementation satisfies its specification, but does not establish that the specification faithfully captures the intended behavior. An incomplete or overly permissive generated specification may therefore certify an incorrect implementation. Reliably evaluating formal specification generation (SpecGen) is thus essential to trustworthy verifiable code generation.

Recent benchmarks, such as the Lean (Moura & Ullrich, 2021)-based CLEVER (Thakur et al., 2025) and VERINA (Ye et al., 2026a), empirically evaluate SpecGen primarily through test suites. In this setup, a generated specification is checked against concrete cases to determine whether its preconditions and postconditions accept intended behaviors and reject unintended ones.

However, this reliance on test suite evaluation exposes a central limitation: the reliability of the SpecGen assessment depends intrinsically on the quantity and quality of the underlying test cases. Unlike traditional code evaluation, specification evaluation requires both positive and negative evidence. Expected input-output pairs test whether a specification admits intended behaviors. Unexpected inputs test whether its precondition excludes invalid inputs, while unexpected outputs test whether its postcondition rejects outputs produced by incorrect implementations. Current benchmarks, however, provide only a limited number of expected input-output pairs and particularly few high-quality unexpected inputs and outputs. Consequently, unsound or incomplete specifications can pass these sparse test suites, leading to systematically inflated estimates of model capabilities.

To bridge this gap, we propose VeriScale, an adversarial framework that scales test suites for formal specification evaluation. VeriScale expands candidate inputs through LLM-based seed generation and type-aware mutation (Liu et al., 2023), classifies them using the ground-truth precondition, and executes the reference implementation on expected inputs to obtain their corresponding outputs.

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.

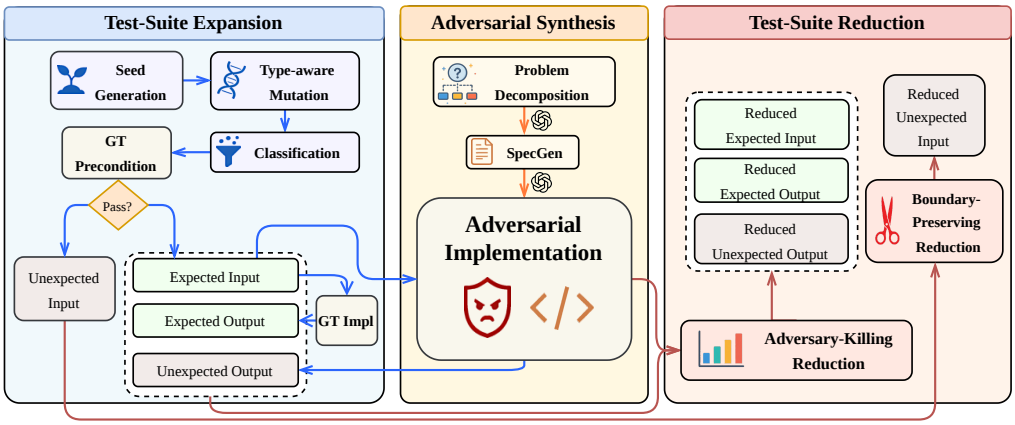


Figure 1: Overview of the VeriScale framework for adversarial test-suite scaling.

Its central challenge is constructing incorrect outputs that expose realistic specification weaknesses: arbitrary incorrect values are often too easy for even flawed postconditions to reject. VeriScale therefore synthesizes adversarial implementations that exploit LLM-generated specifications. Executing these implementations on expected inputs produces outputs that differ from the reference results but are nevertheless accepted by vulnerable postconditions. The adversarial implementations play a second role during test-suite reduction, where they serve as mutants for selecting failure-revealing expected pairs; unexpected inputs are reduced separately while preserving important boundary classes.

To evaluate our approach, we apply VeriScale to VERINA, resulting in the VERINAPLUS and VERINALITE benchmarks. Across eight state-of-the-art LLMs, the average Spec Score decreases from 61.44 on VERINA to 55.09 on VERINAPLUS, exposing approximately 12 additional task-level specification failures per model among 189 tasks. Furthermore, VERINALITE achieves an average score of 56.02, preserving 85.45% of the score reduction produced by the full expansion with only about 16% of the tests in VERINAPLUS.

Our main contributions are as follows:

1. We identify and systematically study a critical blind spot in test-suite-based SpecGen evaluation: sparse suites allow flawed specifications to pass and thereby risk overestimating model capabilities.
2. We propose VeriScale, a test-suite scaling framework that uses adversarial implementations both to generate failure-revealing unexpected outputs and to guide compact test-suite construction. We instantiate it on VERINA to construct the 83× VERINAPLUS benchmark and its compact 13× VERINALITE variant.
3. Through an evaluation of eight state-of-the-art LLMs on 189 tasks, we show that VERINAPLUS exposes approximately 12 additional task-level specification failures per model on average, while VERINALITE retains most of this failure-detection effectiveness at substantially lower evaluation cost.

2 Related Work

Scaling Expected and Unexpected Inputs. Prior work on test generation and code-generation evaluation has largely focused on expanding valid inputs, since broader input coverage improves the likelihood of exposing incorrect programs. LLM-assisted approaches synthesize executable benchmark tests from naturally occurring code (Xie et al., 2024) or invoke an LLM when coverage-guided test generation reaches a plateau (Lemieux et al., 2023). EvalPlus (Liu et al., 2023), more directly, augments HumanEval (Chen et al., 2021) by combining LLM-generated seed inputs with type-aware mutation. Type-aware mutation has also been used to preserve well-formedness when fuzzing SMT solvers (Park et al., 2021), while ingredient reuse originates from generate-and-validate program repair (Martinez & Monperrus, 2019); VeriScale adapts these mechanisms to benchmark inputs. Because generated or mutated inputs are not guaranteed to satisfy a task's semantic constraints,

prior pipelines require task-specific validity logic, typically encoded through generation constraints or filtering scripts. In test-suite-based VCG benchmarks such as CLEVER (Thakur et al., 2025) and VERINA (Ye et al., 2026a), this validity boundary is already made explicit by the ground-truth precondition. VeriScale therefore follows the established seed-generation and mutation paradigm, formalizes its type-preserving mutation operators, and directly uses the provided precondition to partition candidates into expected and unexpected inputs, rather than repeatedly constructing a separate input validator.

Scaling Expected and Unexpected Outputs. For each expected input, its expected output can be obtained straightforwardly by executing the trusted reference implementation, as is standard in code-generation evaluation (Liu et al., 2023). Unexpected outputs are more difficult to construct: existing SpecGen evaluations directly prompt LLMs to generate incorrect outputs (Ye et al., 2026a;b), but direct generation alone guarantees neither that an output is semantically incorrect nor that it can expose a realistic weakness in a generated postcondition. Related adversarial methods attack different targets: CARL perturbs code inputs to induce prediction errors in programming-language models (Yao et al., 2024), whereas AdverIntent-Agent infers competing program intents and generates corresponding test oracles to guide program repair (Ye et al., 2025). Neither treats a formal specification as the attack target or synthesizes executable implementations to exploit it. VeriScale instead attacks a candidate specification with adversarial implementations, then executes them to derive outputs that differ from the reference result yet are accepted by the vulnerable postcondition. Thus, the resulting unexpected outputs are selected by demonstrated specification vulnerabilities, rather than by surface-level output perturbation or unconstrained LLM generation.

3 METHODOLOGY

In this section, we detail VeriScale, a framework designed to scale test-suite-based formal specification evaluation, as illustrated in Figure 1. The framework comprises two primary stages, test-suite expansion and test-suite reduction, both fundamentally driven by adversarial implementations.

3.1 TEST-SUITE EXPANSION

Formally, each benchmark task is equipped with a problem description, a ground-truth precondition, a reference implementation, and existing base inputs. From these components, VeriScale constructs a comprehensive test suite encompassing three types of cases: expected input-output pairs, unexpected inputs, and unexpected outputs. Expected input-output pairs test whether a generated specification accepts intended behaviors, while unexpected inputs and unexpected outputs test whether it rejects invalid inputs and incorrect outputs, respectively.

Seed Generation and Type-aware Mutation. Drawing inspiration from EvalPlus (Liu et al., 2023), VeriScale employs LLMs to synthesize a diverse set of initial seed inputs. To construct the generation prompt for each task, we aggregate the problem description, the ground-truth precondition, input parameter signatures (names and types), and any pre-existing expected and unexpected inputs. The LLM is then queried to output a specified volume of distinct candidate inputs, establishing a seed pool for the subsequent type-aware mutation phase. The detailed prompt is shown in Appendix C.

To systematically expand the initial seed pool, VeriScale employs an iterative, type-aware mutation strategy (Park et al., 2021). To preserve task-specific semantics during structural perturbation, we extract an ingredient pool from existing candidates, allowing target parameters to stochastically reuse these contextual values (Martinez & Monperrus, 2019). When reuse is not triggered, the framework falls back on type-specific heuristic perturbations. To construct complex edge cases, the mutation engine performs a sequence of modifications by randomly targeting a single parameter per step under a bounded generation budget. To mathematically capture how a specific value is perturbed during any given step, we formalize our type-aware mutation procedure as a five-tuple $\mathcal{M} = (\mathcal{T}, \mathcal{D}, \mathcal{O}, p, q)$.

- $\mathcal{T}$ is the set of input types.
- $\mathcal{D}$ maps each type $\tau \in \mathcal{T}$ to its domain D_τ.

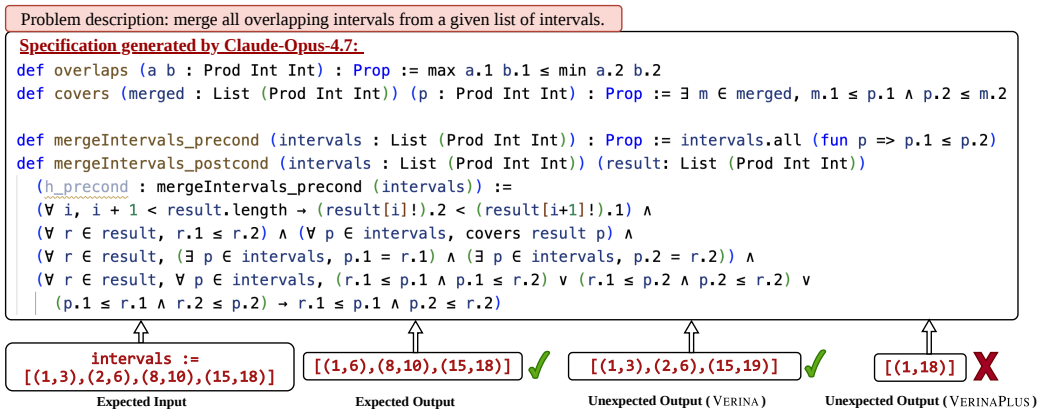


Figure 2: Case study of a flawed specification on VERINA #advanced_47.

- $\mathcal{O}$ maps each type $\tau \in \mathcal{T}$ to a finite set of mutation schemas $\mathcal{O}_\tau = \{s_{\tau,1}, \ldots, s_{\tau,k_\tau}\}$. Each schema $s_{\tau,i} : D_\tau \times \Theta_{\tau,i} \to D_\tau$ is parameterized by a parameter space $\Theta_{\tau,i}$.
- p maps each type $\tau \in \mathcal{T}$ to a probability distribution p_τ over $\mathcal{O}_\tau$, specifying how a mutation schema is selected.
- q maps each pair (τ, i) to a probability distribution $q_{\tau,i}$ over the parameter space $\Theta_{\tau,i}$, specifying how the parameters of schema $s_{\tau,i}$ are sampled. In general, this distribution may depend on the input value $x \in D_\tau$.

Given an input value $x \in D_\tau$, the mutator first samples a mutation schema $I \sim p_\tau$, and then samples its parameters according to $\theta \sim q_{\tau,I}(\cdot \mid x)$. The mutated value is then defined as

$$M_\tau(x) = s_{\tau,I}(x; \theta).$$

Since each schema is type-preserving, for any $x \in D_\tau, i \in \{1, \ldots, k_\tau\}, \theta \in \Theta_{\tau,i}$, we have

$$s_{\tau,i}(x; \theta) \in D_\tau.$$

Therefore, the mutation procedure preserves the input type by construction:

$$M_\tau(x) \in D_\tau.$$

See Appendix A for more details of these schemas.

Precondition-Guided Input Classification. To classify the candidate inputs, VeriScale operates in alignment with Verina (Ye et al., 2026a), evaluating them against the ground-truth precondition using a two-stage verification pipeline. Prior to logical evaluation, we enforce a strict syntax filter. Each candidate is rendered as a Lean precondition application, denoted as `<expr>`, and validated via the `#check <expr>` command. Candidates failing this step due to missing arguments, type mismatches, or syntax errors are immediately discarded.

In the first stage, we apply a bidirectional procedure, evaluating both `#guard decide <expr>` and `#guard decide (¬ <expr>)`. An input is classified as expected if the positive expression evaluates to true (or the negation to false), and unexpected if the inverse holds. For these unresolved inputs, we construct an `example`, unfold the precondition, aggressively simplify the goal context via `simp_all!`, and execute `plausible` bidirectionally. Discovering a counterexample for the negated expression confirms the input as expected, whereas finding one for the positive expression marks it as unexpected. Candidates that still lack a definitive verdict due to timeouts or search exhaustion are permanently excluded from the final test suite. For the isolated expected inputs, we execute the reference implementation to compute their corresponding outputs.

Adversarial Synthesis of Unexpected Outputs. Beyond verifying basic functionality, evaluating the soundness and completeness of formal specifications requires high-quality unexpected outputs to serve as negative test cases. Existing approaches often rely on LLMs to directly hallucinate incorrect

outputs or employ naive random sampling to deviate from the ground truth. However, these trivially incorrect outputs are usually easily rejected by even highly flawed specifications, failing to provide meaningful diagnostic feedback for underspecification. As illustrated in Figure 2, a concrete case study demonstrates this limitation.

To synthesize highly discriminative unexpected outputs, VeriScale introduces an adversarial execution paradigm. We first prompt language models to perform problem decomposition to reduce task complexity and clarify structural logic (Liu et al., 2026). Next, we leverage advanced models to generate high-quality formal specifications. Crucially, a more capable model acts as a red team (Perez et al., 2022), crafting adversarial implementations to exploit vulnerabilities within these specifications. As a fallback, if the model fails to generate an adversarial implementation, we systematically drop constraints from the ground-truth specification to expose an attack target, thereby facilitating the successful synthesis of an adversarial implementation.

These adversarial implementations are subsequently executed on the expected inputs generated during the previous stage. If an adversarial implementation produces an output that diverges from the ground truth but is still successfully accepted by the generated postconditions, this mismatched value is formally retained. Because these unexpected outputs stem from systematic specification gaming rather than arbitrary hallucination, they closely mimic realistic logical errors, making them significantly more effective at exposing specification unsoundness and incompleteness.

3.2 Test-Suite Reduction

To reduce computational overhead, we filter unexpected inputs via boundary-preserving reduction and reduce expected input-output pairs through adversary-killing reduction. Unexpected outputs are implicitly filtered alongside their corresponding inputs.

Example of Adversary-Killing Reduction

Problem Description: (VERINA #advanced_16) Implement the insertion sort algorithm in Lean 4. The function takes a single list of integers as input and returns a new list that contains the same integers in ascending order.

Candidate Expected Input: `[0, -1, -2, -3, -4]`
Expected Output: `[-4, -3, -2, -1, 0]`

Adversary-Killing Count

- `def insertionSort (xs : List Int) : List Int := xs.reverse`
  Output: `[-4, -3, -2, -1, 0]` → Same as expected
- `def insertionSort (xs : List Int) : List Int := match xs with | [] => [] | x :: _ => List.replicate xs.length x`
  Output: `[0, 0, 0, 0, 0]` ✔ Different
- `def insertionSort (xs : List Int) : List Int := List.replicate xs.length 0`
  Output: `[0, 0, 0, 0, 0]` ✔ Different
- `def insertionSort (xs : List Int) : List Int := (List.range xs.length).map (fun n => Int.ofNat n)`
  Output: `[0, 1, 2, 3, 4]` ✔ Different
- `def insertionSort (xs : List Int) : List Int := []`
  Output: `[]` ✔ Different
- `def insertionSort (xs : List Int) : List Int := xs`
  Output: `[0, -1, -2, -3, -4]` ✔ Different
- `def insertionSort (xs : List Int) : List Int := match xs with | [] => [] | x :: _ => [x]`
  Output: `[0]` ✔ Different

Adversary-Killing Count: 6 / 7.

Figure 3: An example of adversary-killing reduction. Given an expected input, an adversarial implementation is considered "killed" if its output differs from the expected output. The illustrated case kills six out of seven adversarial implementations, making it a high-quality case (it may find more weakness of a generated specification).

Dataset	Expected Input-Output	Unexpected Output	Unexpected Input
VERINA	5.89 (2–13)	12.64 (0–34)	0.65 (0–7)
VERINAPLUS	367.64 (5–700) (×62.42)	1104.30 (15–2891) (×87.37)	119.00 (0–684) (×183.08)
VERINALITE	49.22 (5–50) (×8.36)	191.74 (15–457) (×15.17)	15.80 (0–50) (×24.31)

Table 1: Comparison of test suite volumes. Data is formatted as Mean (Min–Max), accompanied by the relative growth multiplier compared to the baseline VERINA.

Boundary-Preserving Reduction. For unexpected inputs, we adopt a boundary-preserving reduction strategy. We extract lightweight structural signatures (e.g., empty containers, zero or negative values, mismatched lengths, and ordering patterns) to approximate failure modes, and prioritize inputs that correspond to common boundary violations. For each task, we first retain representative examples from these critical boundary buckets, ensuring that frequent and semantically meaningful unexpected inputs are preserved. If the budget is not exhausted, we further fill the remaining slots with representative inputs from diverse structural buckets, and finally use a deterministic priority order as a fallback. This approach keeps the unexpected input set compact while maintaining coverage of typical precondition violations and edge-case behaviors.

Adversary-Killing Reduction. We further use adversary-killing to reduce the set of expected input-output pairs. Specifically, we repurpose the adversarial implementations synthesized in the previous stage, treating each as a distinct mutant. We associate each expected pair with the set of adversarial implementations it can kill, formulating test reduction as a set-cover problem. We first select expected pairs that cover the largest number of previously unkilled mutants using a greedy procedure, ensuring that every implementation detectable by the expanded test suite remains covered by at least one retained pair. Then, under a fixed per-task budget, we add additional pairs that kill the highest number of implementations. This strategy avoids the coverage loss caused by purely diversity-based or random reduction, allowing the lightweight test suite to substantially reduce its size while preserving its ability to distinguish flawed specifications.

4 EXPERIMENTS

4.1 CONSTRUCTING THE AUGMENTED BENCHMARKS

We instantiate VeriScale on the VERINA dataset (Ye et al., 2026a) to construct VERINAPLUS and VERINALITE, two augmented benchmarks dedicated to formal specification evaluation. During test-suite expansion, we employ GPT-5.3-Codex for seed generation, problem decomposition, and adversarial implementation synthesis. To expose a diverse range of specification weaknesses, we collect candidate specifications generated by Claude-Haiku-4.5, Claude-Sonnet-4.5, GPT-5.1, GPT-4.1, Qwen3-Max, and DeepSeek-V3.2. These specifications serve as attack targets during benchmark construction: the synthesized adversarial implementations exploit their potential loopholes to produce challenging unexpected outputs. Detailed hyperparameter configurations of the expansion and reduction stages are provided in Appendix B.

Table 1 summarizes the volumes of the three test-case categories in each benchmark. Across all three categories, VERINAPLUS and VERINALITE expand the original VERINA test suite by approximately 83× and 13×, respectively.

4.2 EXPERIMENTAL SETUP

We evaluate eight state-of-the-art LLMs: Claude-Sonnet-4.6, Claude-Opus-4.7, GPT-5.3-Codex, GPT-5.5, Gemini-3.1-Flash-Lite-Preview, Gemini-3.1-Pro-Preview, Qwen3.6-Max-Preview, and DeepSeek-V4-Pro. We assess their SpecGen performance on VERINA, VERINAPLUS, and VERINALITE. Following VERINA (Ye et al., 2026a), the Spec Score is determined by four task-level criteria: Pre S checks whether the precondition accepts expected inputs; Pre C checks whether it rejects unexpected inputs; Post S checks whether the postcondition rejects unexpected outputs; and Post C checks whether it accepts expected outputs. Pre S&C and Post S&C require both correspond-

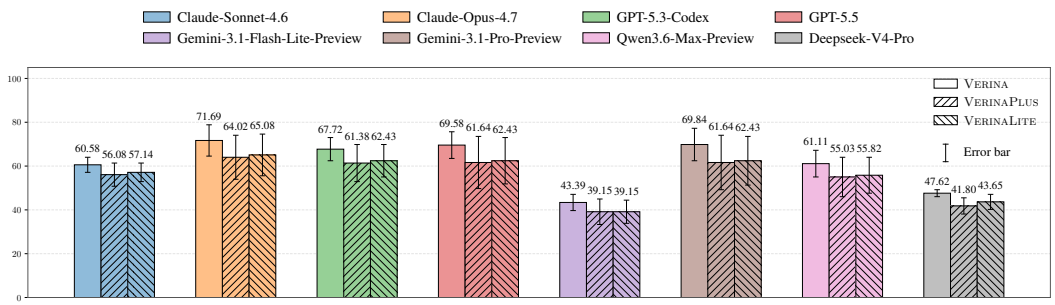


Figure 4: SpecGen performance of eight models on VERINA, VERINAPLUS, and VERINALITE. Bars report the midpoint of the lower and upper Spec Score bounds, while whiskers indicate the corresponding score intervals. Lower scores indicate that more specification violations are exposed.

ing criteria, and the overall result is

$$\text{Spec Score} = \text{Pre S} \wedge \text{Pre C} \wedge \text{Post S} \wedge \text{Post C}.$$

For unresolved cases, the lower bound treats unknowns as failures, the upper bound treats them as passes, and their midpoint is used as the primary score.

We adapt the official VERINA evaluation harness in two ways. First, we remove its two-shot demonstrations and evaluate all tasks zero-shot. Second, instead of compiling all tests for a task in one Lean file, we compile each test case independently with a 120-second timeout, ensuring that one timeout does not affect the remaining cases.

All experiments use Lean v4.24.0 and run on a server with two Intel Xeon Platinum 8380 CPUs and 80 physical cores.

4.3 RESULTS AND ANALYSIS

Overall and Component-Level Results. Figure 4 compares the eight models on VERINA, VERINAPLUS, and VERINALITE. Every model obtains a lower Spec Score on both augmented benchmarks than on the original VERINA benchmark, indicating that the original test suites leave substantial specification weaknesses undetected. Across the eight models, the average score decreases from 61.44 on VERINA to 55.09 on VERINAPLUS, a reduction of 6.35 percentage (about 12 task-level failures per model on average). The largest model-specific decrease is 8.20 percentage points for Gemini-3.1-Pro-Preview, whose score falls from 69.84 to 61.64.

A component-wise analysis shows that the expanded test suites expose particularly pronounced weaknesses in generated postconditions. Averaged across the eight models, Pre S and Pre C decrease by 3.74 and 2.75 percentage points, respectively, when moving from VERINA to VERINAPLUS, whereas Post S and Post C decrease by 5.03 and 4.27 points. This pattern is especially clear for GPT-5.5: its Pre S and Pre C scores decrease from 93.13 and 92.07 to 89.69 and 88.63, while its Post S and Post C scores fall more sharply from 82.01 and 76.99 to 75.13 and 71.69. Consequently, its overall Spec Score drops from 69.58 to 61.64. These results suggest that sparse tests particularly obscure weaknesses in how generated postconditions constrain the relationship between inputs and outputs.

Despite using substantially fewer tests, VERINALITE closely tracks the full expanded benchmark. Its average Spec Score is 56.02, only 0.92 percentage points higher than VERINAPLUS and 5.43 points lower than VERINA. While VERINAPLUS expands the original test suite by approximately 83×, VERINALITE requires only a 13× expansion. Thus, VERINALITE preserves 85.45% of the score reduction produced by the full expansion with a substantially smaller test suite.

Unexpected Input/Output Error Modes. We conduct a sample-based human audit of unexpected input/output (UI/UO) from 10 advanced problems of VERINALITE and summarize their failure modes in Table 2 and Table 3. The diversity of error modes is primarily determined by the specific constraints of the corresponding problem's specifications. For the problems included in our sam-

Problem	Number of UI	Error modes
5	50	Empty: 16; OOB: 21; Invalid: 26
8	50	Empty: 9; Invalid length: 48
35	50	Empty: 2; No solution: 48
40	2	Empty: 2
49	50	Unsorted: 50
52	50	Empty: 2; Missing: 50
57	50	Duplicate: 24; Subset: 38
68	50	Invalid: 50
70	50	Empty: 1; OOB: 49; Duplicate: 19

Table 2: Unexpected input (UI) error modes. Problem identifiers omit the advanced_ prefix (e.g., 5 denotes advanced_5). Empty = empty inputs; OOB = out of bounds; Missing = missing required values; Subset = subset condition violations. Besides these problems, the preconditions of advanced_13 are relatively complex, comprising six distinct preconditions. The numbers of unexpected outputs associated with each precondition are 39 / 6 / 42 / 35 / 2 / 4, respectively.

Problem (UO)	Const.	Return	Count	Length	Order	Sign	Other
5 (457)	105	94	49	403	177		13
8 (160)	89		90			14	
13 (36)							36
35 (174)	48	65	50			49	
40 (148)	45	75	102				1
49 (246)	102	97		148	49		
52 (296)	49		272				
57 (229)	147	85		50	50		
68 (198)	99	98		172	49		
70 (178)	132	47				50	

Table 3: Unexpected output (UO) error modes. Problem IDs omit the advanced_ prefix. Columns denote constant return/sentinel/padding, returned input elements, incorrect/proxy count, length-/structure, ordering/conversion, and sign/boundary errors, respectively.

pled analysis, the results of our reduction strategy indicate that the reduced UI/UO can still cover potential specification vulnerabilities, and each error mode of UI/UO corresponds to a genuine code defect, not simply trivial errors.

4.4 Ablation Study

We conduct two ablation studies based on VerinaLite. First, VerinaLite-Unex replaces adversarially synthesized unexpected outputs with outputs generated directly by GPT-5.3-Codex while retaining the same expected input-output pairs and unexpected inputs. The corresponding prompt is adapted from (Ye et al., 2026b) and is provided in Appendix C. Second, VerinaLite-Random replaces boundary-preserving and adversary-killing reduction with random selection under the same per-task budgets. We repeat random reduction with seeds 41–45 and report the average results across the five runs.

Adversarial Synthesis vs. Direct LLM Generation. First, building VerinaLite-Unex costs $36.93, whereas building VerinaPlus costs $52.58. Second, regarding the quality of the constructed test cases, VerinaLite-Unex produces invalid cases for eight problems, where the generated outputs are either identical to or semantically equivalent to the expected outputs. In terms of experimental results, compared with adversarial synthesis, direct LLM generation increases the average Spec Score by approximately one task (primarily driven by verina_basic_107). Beyond this shared case, each model also exhibits changes on a small number of different tasks. Compared with direct LLM generation, our approach produces more successful negative cases and covers a larger number of tasks.

Model	Benchmark	Pre S	Pre C	Pre S&C	Post S	Post C	Post S&C	Spec Score
Claude-Sonnet-4.6	VERINALITE	89.95	86.78	82.01	69.31	64.55	63.49	**57.14**
	VERINALITE-UNEX	89.95	86.78	82.01	69.05 (↓ 0.26)	64.55	63.76 (↑ 0.27)	57.41 (↑ 0.27)
	VERINALITE-RANDOM	90.37 (↑ 0.42)	86.83 (↑ 0.05)	82.38 (↑ 0.37)	68.47 (↓ 0.84)	64.87 (↑ 0.32)	63.39 (↓ 0.10)	57.25 (↑ 0.11)
Claude-Opus-4.7	VERINALITE	90.74	93.92	86.51	80.43	73.81	72.22	**65.08**
	VERINALITE-UNEX	90.74	93.92	86.51	80.43	73.81	72.49 (↑ 0.27)	65.61 (↑ 0.53)
	VERINALITE-RANDOM	91.43 (↑ 0.69)	93.65 (↓ 0.27)	87.15 (↑ 0.64)	80.21 (↓ 0.22)	74.45 (↑ 0.64)	72.60 (↑ 0.38)	66.19 (↑ 1.11)
GPT-5.3-Codex	VERINALITE	90.48	89.42	83.07	73.81	72.49	69.58	**62.43**
	VERINALITE-UNEX	90.48	89.42	83.07	74.34 (↑ 0.53)	72.49	70.90 (↑ 1.32)	63.49 (↑ 1.06)
	VERINALITE-RANDOM	90.96 (↑ 0.48)	89.79 (↑ 0.37)	83.81 (↑ 0.74)	73.60 (↓ 0.21)	72.44 (↓ 0.05)	69.58	63.02 (↑ 0.59)
GPT-5.5	VERINALITE	89.16	89.42	82.81	76.46	72.22	69.84	**62.43**
	VERINALITE-UNEX	89.16	89.42	82.81	75.13 (↓ 1.33)	72.22	70.64 (↑ 0.80)	62.96 (↑ 0.53)
	VERINALITE-RANDOM	89.90 (↑ 0.74)	89.42	83.76 (↑ 0.95)	75.93 (↓ 0.53)	72.86 (↑ 0.64)	70.37 (↑ 0.53)	63.76 (↑ 1.33)
Gemini-3.1-Flash-Lite-Preview	VERINALITE	75.14	79.10	71.17	47.89	45.77	44.44	**39.15**
	VERINALITE-UNEX	75.14	79.10	71.17	48.15 (↑ 0.26)	45.77	44.18 (↓ 0.26)	**39.15**
	VERINALITE-RANDOM	75.87 (↑ 0.73)	78.63 (↓ 0.47)	71.11 (↓ 0.06)	47.94 (↑ 0.05)	46.14 (↑ 0.37)	44.87 (↑ 0.43)	40.11 (↑ 0.96)
Gemini-3.1-Pro-Preview	VERINALITE	89.95	90.74	83.60	75.93	73.55	71.17	**62.43**
	VERINALITE-UNEX	89.95	90.74	83.60	75.66 (↓ 0.27)	73.55	71.96 (↑ 0.79)	62.96 (↑ 0.53)
	VERINALITE-RANDOM	90.58 (↑ 0.63)	91.27 (↑ 0.53)	84.71 (↑ 1.11)	75.88 (↓ 0.05)	74.18 (↑ 0.63)	71.43 (↑ 0.26)	63.65 (↑ 1.22)
Qwen3.6-Max-Preview	VERINALITE	88.10	88.10	80.69	66.93	64.02	63.23	**55.82**
	VERINALITE-UNEX	88.10	88.10	80.69	67.20 (↑ 0.27)	64.02	63.50 (↑ 0.27)	56.35 (↑ 0.53)
	VERINALITE-RANDOM	88.84 (↑ 0.74)	88.63 (↑ 0.53)	81.90 (↑ 1.21)	66.83 (↓ 0.10)	64.23 (↑ 0.21)	63.44 (↑ 0.21)	56.83 (↑ 1.01)
DeepSeek-V4-Pro	VERINALITE	84.66	83.34	78.57	51.06	49.47	48.68	43.65
	VERINALITE-UNEX	84.66	83.34	78.57	50.53 (↓ 0.53)	49.47	49.21 (↑ 0.53)	44.18 (↑ 0.53)
	VERINALITE-RANDOM	84.61 (↓ 0.05)	84.34 (↑ 1.00)	79.48 (↑ 0.91)	50.48 (↓ 0.58)	49.26 (↓ 0.21)	48.36 (↓ 0.32)	**43.54** (↓ 0.11)

Table 4: Results of the unexpected-output and test-suite-reduction ablations. We report mean scores (%); for metrics given as lower and upper bounds, the mean is their midpoint. Results for VERINALITE-RANDOM are additionally averaged over five reductions using seeds 41–45. Changes are measured in percentage points relative to VERINALITE. Since lower scores are better, red upward arrows indicate regressions and purple downward arrows indicate improvements. The best Spec Score for each model is highlighted in bold, and unchanged results are left unannotated.

Adversary-Guided vs. Random Test-Suite Reduction. As shown in Table 4, random reduction increases the Spec Score for seven models, except for DeepSeek-V4-Pro; the largest increase is 1.33 percentage points for GPT-5.5. Component-level results further show that random reduction primarily weakens the detection of precondition errors: Pre S, Pre C, and Pre S&C increase by 0.55, 0.22, and 0.73 percentage points on average, respectively. Although it slightly lowers Post S for some models, the average Post S&C and overall Spec Score still worsen by 0.17 and 0.78 percentage points, respectively. These findings indicate that adversary-killing reduction prioritizes expected cases that can "kill" more adversarial implementations, whereas boundary-preserving reduction retains important precondition boundary cases. Consequently, under the same budget, VERINALITE constructs a more discriminative compact test suite that better exposes specification vulnerabilities.

5 CONCLUSION

In this work, we propose VeriScale, an adversarial test-suite scaling framework for formal specification evaluation. Applied to VERINA, VeriScale produces VERINAPLUS and VERINALITE, which consistently reveal additional failures across eight LLMs; notably, VERINALITE retains most of the effectiveness of full expansion with substantially lower evaluation cost. These results show that adversarially constructed test suites are important for more reliable and efficient evaluation of specification generation systems.

6 LIMITATIONS AND FUTURE WORK

In this paper, we instantiate and evaluate VeriScale in Lean using VERINA. Although the methodology is not inherently tied to Lean, its effectiveness and engineering cost in other formal languages (e.g., Dafny (Leino, 2010)) remain to be validated. Moreover, the improvement of our adversarial test-suite scaling is modest, there still remains room to improve reduction quality. Future work could design dedicated expansion strategies for each of the four evaluation components of specifications.

VeriScale currently constructs a fixed offline dataset before evaluation. A promising direction is to develop an online adversarial evaluation framework that dynamically generates new inputs, adversarial implementations, and failure-revealing tests for each evaluated model.

AI USE STATEMENT

Language models are used as part of the proposed research method, as described in the paper. GPT-5.3-Codex is used for seed generation, problem decomposition, and adversarial implementation synthesis, while additional language models generate candidate specifications used as attack targets. All generated artifacts are filtered by the stated construction pipeline, checked with Lean, and evaluated using the reported test-suite procedures. The authors remain responsible for the experimental design, analysis, and final manuscript.

ETHICS STATEMENT

This work does not collect or process personal, sensitive, or private data, and does not involve human or animal subjects. The evaluated artifacts are synthetic Lean programs, specifications, and test cases. The main foreseeable risk is the misuse of adversarial examples to expose weaknesses in formal specifications; the study uses these examples for benchmark evaluation and reports the construction procedure to support responsible verification research.

REPRODUCIBILITY STATEMENT

We provide the benchmark construction details, mutation and reduction procedures, prompts, and hyperparameter settings in the paper and its appendix. Experiments use Lean v4.24.0 and the hardware configuration reported in the experimental setup. The complete ablation results are included in Table 4, and the appendix contains the prompts and configuration details needed to reproduce the benchmark construction and evaluation. Our code and data are provided in the supplementary material.

REFERENCES

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 919–931, 2023. doi: 10.1109/ICSE48619.2023.00085.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Xiaoyang Liu, Zineng Dong, Yifan Bai, Yantao Li, Yuntian Liu, and Tao Luo. Decompose, structure, and repair: A neuro-symbolic framework for autoformalization via operator trees. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=b9PBqFgXp6.

Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *Journal of Systems and Software*, 151:65–80, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.01.069. URL https://www.sciencedirect.com/science/article/pii/S0164121219300159.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing smt solvers. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi: 10.1145/

3485529. URL https://doi.org/10.1145/3485529.

Ethan Perez, Saffron Huang, Francis Song, Trevor Cai, Roman Ring, John Aslanides, Amelia Glaese, Nat McAleese, and Geoffrey Irving. Red teaming language models with language models. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3419–3448, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.225. URL https://aclanthology.org/2022.emnlp-main.225/.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*, pp. 134–155, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-65111-3. doi: 10.1007/978-3-031-65112-0_7. URL https://doi.org/10.1007/978-3-031-65112-0_7.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=IbOacMF5qd.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. Codebench-gen: Creating scalable execution-based code generation benchmarks, 2024. URL https://arxiv.org/abs/2404.00566.

Kaichun Yao, Hao Wang, Chuan Qin, Hengshu Zhu, Yanjun Wu, and Libo Zhang. Carl: Unsupervised code-based adversarial attacks for programming language models via reinforcement learning. *ACM Trans. Softw. Eng. Methodol.*, 34(1), December 2024. ISSN 1049-331X. doi: 10.1145/3688839. URL https://doi.org/10.1145/3688839.

He Ye, Aidan Z.H. Yang, Chang Hu, Yanlin Wang, Tao Zhang, and Claire Le Goues. Adverintent-agent: Adversarial reasoning for repair based on inferred program intent. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728939. URL https://doi.org/10.1145/3728939.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=0A4Uf88pog.

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. In *3rd AI for Math Workshop: Toward Self-Evolving Scientific Agents*, 2026b. URL https://openreview.net/forum?id=i9MTMRHHG7.

A Detailed Type-aware Mutation Rules

When ingredient pool reuse is not triggered, we apply type-specific mutations for the following Lean input types: `Int`, `Nat`, `List Int`, `Array Int`, `List Nat`, `Array Nat`, and `List Char`, `String`.

Integers. When $\tau = \texttt{Int}$, the domain is $D_\tau = \mathbb{Z}$. We define mutation-schema set as

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}\}$$

where

$$\begin{aligned} s_{\tau,1}(x,\delta) &= x+\delta, & \Theta_{\tau,1} &= \{-2,-1,0,1,2\}, \\ s_{\tau,2}(x,\delta) &= \delta x, & \Theta_{\tau,2} &= \{-2,-1,0,1,2\}. \end{aligned}$$

When ingredient pool reuse is disabled, both the mutation schema and its parameters are sampled uniformly. Therefore,

$$\begin{aligned} p_\tau(i) &= \frac{1}{2}, \quad i \in \{1,2\}, \\ q_{\tau,i}(\delta) &= \frac{1}{5}, \quad i \in \{1,2\}, \quad \delta \in \{-2,-1,0,1,2\}. \end{aligned}$$

Natural numbers. When $\tau = \texttt{Nat}$, the domain is $D_\tau = \mathbb{N}$. We obtain the mutation schemas by applying the integer schemas and clipping the result to the natural-number domain. Formally, let $\mathrm{clip}(z) = \max(0, z)$, we define

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}\},$$

where

$$\begin{aligned} s_{\tau,1}(x;\delta) &= \mathrm{clip}(x+\delta), & \Theta_{\tau,1} &= \{-2,-1,0,1,2\}, \\ s_{\tau,2}(x;\delta) &= \mathrm{clip}(\delta x), & \Theta_{\tau,2} &= \{-2,-1,0,1,2\}. \end{aligned}$$

The clipping operation ensures that every mutated value remains in $D_\tau = \mathbb{N}$. When ingredient pool reuse is disabled, both the mutation schema and its parameters are sampled uniformly:

$$\begin{aligned} p_\tau(i) &= \frac{1}{2}, \quad i \in \{1,2\}, \\ q_{\tau,i}(\delta) &= \frac{1}{5}, \quad i \in \{1,2\}, \quad \delta \in \{-2,-1,0,1,2\}. \end{aligned}$$

Integer lists and arrays. When $\tau \in \{\texttt{List Int}, \texttt{Array Int}\}$, we model both lists and arrays as finite sequences. Thus,

$$D_\tau = \left(\mathbb{Z}^* \triangleq \bigcup_{n\in\mathbb{N}} \mathbb{Z}^n\right).$$

We define the mutation-schema set as

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}, s_{\tau,3}, s_{\tau,4}\},$$

corresponding to element modification, appending, deletion, and reversal. For a sequence x, let $|x|$ denote its length, x_j its j-th element, $\mathrm{replace}(x,j,v)$ the sequence obtained by replacing x_j with v, and $\mathrm{del}(x,j)$ the sequence obtained by deleting x_j. The schemas are defined as

$$\begin{aligned} s_{\tau,1}(x;j,i,\delta) &= \begin{cases} \mathrm{replace}(x, j, s_{\texttt{Int},i}(x_j;\delta)), & |x| > 0, \\ x, & |x| = 0, \end{cases} \\ s_{\tau,2}(x;v) &= x +\!\!+ [v], \\ s_{\tau,3}(x;j) &= \begin{cases} \mathrm{del}(x,j), & |x| > 0, \\ x, & |x| = 0, \end{cases} \\ s_{\tau,4}(x) &= \mathrm{rev}(x). \end{aligned}$$

The corresponding parameter spaces are

$$\Theta_{\tau,1}(x) = \{0, \ldots, |x|-1\} \times \{1,2\} \times \{-2,-1,0,1,2\},$$
$$\Theta_{\tau,2} = \{-5,-4,\ldots,5\},$$
$$\Theta_{\tau,3}(x) = \{0, \ldots, |x|-1\},$$
$$\Theta_{\tau,4} = \varnothing.$$

When $|x| = 0$, the index-dependent schemas are interpreted as the identity function, so the mutation remains well-defined. When ingredient pool reuse is disabled, the schema distribution is uniform:

$$p_\tau(i) = \frac{1}{4}, \quad i \in \{1,2,3,4\}.$$

The parameter distributions are uniform over their corresponding finite parameter spaces. In particular,

$$q_{\tau,1}(j,i,\delta \mid x) = \frac{1}{|x|} \cdot \frac{1}{2} \cdot \frac{1}{5}, \quad j \in \{0,\ldots,|x|-1\}, \quad i \in \{1,2\}, \quad \delta \in \{-2,-1,0,1,2\},$$
$$q_{\tau,2}(v) = \frac{1}{11}, \qquad v \in \{-5,-4,\ldots,5\},$$
$$q_{\tau,3}(j \mid x) = \frac{1}{|x|}, \qquad j \in \{0,\ldots,|x|-1\},$$
$$q_{\tau,4}(*) = 1.$$

Natural-number lists and arrays. When $\tau \in \{\texttt{List Nat}, \texttt{Array Nat}\}$, domain of the two types are both

$$D_\tau = \left(\mathbb{N}^* \triangleq \bigcup_{n \in \mathbb{N}} \mathbb{N}^n \right).$$

The mutation schemas are analogous to those for integer lists and arrays, except that every newly generated element is clipped to the natural-number domain. Let $\mathrm{clip}(z) = \max(0, z)$. We define

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}, s_{\tau,3}, s_{\tau,4}\},$$

where

$$s_{\tau,1}(x; j, i, \delta) = \begin{cases} \mathrm{replace}(x, j, \mathrm{clip}(s_{\texttt{Int},i}(x_j; \delta))), & |x| > 0, \\ x, & |x| = 0, \end{cases}$$
$$s_{\tau,2}(x; v) = x +\!\!+ [\mathrm{clip}(v)],$$
$$s_{\tau,3}(x; j) = \begin{cases} \mathrm{del}(x, j), & |x| > 0, \\ x, & |x| = 0, \end{cases}$$
$$s_{\tau,4}(x; *) = \mathrm{rev}(x).$$

The parameter spaces are

$$\Theta_{\tau,1}(x) = \{0, \ldots, |x|-1\} \times \{1,2\} \times \{-2,-1,0,1,2\},$$
$$\Theta_{\tau,2} = \{-5,-4,\ldots,5\},$$
$$\Theta_{\tau,3}(x) = \{0, \ldots, |x|-1\},$$
$$\Theta_{\tau,4} = \varnothing.$$

For empty lists or arrays, we again interpret index-dependent mutations as identity transformations. When ingredient pool reuse is disabled,

$$p_\tau(i) = \frac{1}{4}, \quad i \in \{1,2,3,4\}.$$

The parameters are sampled uniformly:

$$q_{\tau,1}(j,i,\delta \mid x) = \frac{1}{|x|} \cdot \frac{1}{2} \cdot \frac{1}{5}, \quad j \in \{0,\ldots,|x|-1\}, \quad i \in \{1,2\}, \quad \delta \in \{-2,-1,0,1,2\},$$
$$q_{\tau,2}(v) = \frac{1}{11}, \qquad v \in \{-5,-4,\ldots,5\},$$
$$q_{\tau,3}(j \mid x) = \frac{1}{|x|}, \qquad j \in \{0,\ldots,|x|-1\},$$
$$q_{\tau,4}(*) = 1.$$

As before, when $|x| = 0$, we use a singleton dummy parameter space for the index-dependent schemas.

Character lists. When $\tau = \texttt{List Char}$, the domain is

$$D_\tau = \left(\Sigma_{\text{char}}^* \triangleq \bigcup_{n \in \mathbb{N}} \Sigma_{\text{char}}^n \right),$$

where Σ_{char} denotes the finite set of allowable characters. We define

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}, s_{\tau,3}, s_{\tau,4}\},$$

corresponding to character modification, appending, deletion, and reversal:

$$\begin{aligned}
s_{\tau,1}(x; j, c) &= \begin{cases} \text{replace}(x, j, c), & |x| > 0, \\ x, & |x| = 0, \end{cases} \\
s_{\tau,2}(x; c) &= x \mathbin{+\!\!+} [c], \\
s_{\tau,3}(x; j) &= \begin{cases} \text{del}(x, j), & |x| > 0, \\ x, & |x| = 0, \end{cases} \\
s_{\tau,4}(x; *) &= \text{rev}(x).
\end{aligned}$$

The parameter spaces are

$$\begin{aligned}
\Theta_{\tau,1}(x) &= \{0, \ldots, |x| - 1\} \times \Sigma_{\text{char}}, \\
\Theta_{\tau,2} &= \Sigma_{\text{char}}, \\
\Theta_{\tau,3}(x) &= \{0, \ldots, |x| - 1\}, \\
\Theta_{\tau,4} &= \varnothing.
\end{aligned}$$

When ingredient pool reuse is disabled, the schema is sampled uniformly:

$$p_\tau(i) = \frac{1}{4}, \quad i \in \{1, 2, 3, 4\}.$$

The parameter distributions are uniform:

$$\begin{aligned}
q_{\tau,1}(j, c \mid x) &= \frac{1}{|x|} \cdot \frac{1}{|\Sigma_{\text{char}}|}, && j \in \{0, \ldots, |x| - 1\}, \quad c \in \Sigma_{\text{char}}, \\
q_{\tau,2}(c) &= \frac{1}{|\Sigma_{\text{char}}|}, && c \in \Sigma_{\text{char}}, \\
q_{\tau,3}(j \mid x) &= \frac{1}{|x|}, && j \in \{0, \ldots, |x| - 1\}, \\
q_{\tau,4}(*) &= 1.
\end{aligned}$$

For $|x| = 0$, the index-dependent schemas use a singleton dummy parameter space and reduce to the identity transformation.

Strings. When $\tau = \texttt{String}$, we model the domain in the same way as `List Char`, namely as the set of all finite character sequences:

$$\mathcal{D}_\tau = \Sigma_{\text{char}}^*.$$

Unlike `List Char`, however, we define the mutation-schema set as

$$\mathcal{O}_\tau \triangleq \{s_{\tau,1}, s_{\tau,2}, s_{\tau,3}\},$$

where

$$\begin{aligned}
s_{\tau,1}(x) &= \epsilon, & \Theta_{\tau,1} &= \varnothing, \\
s_{\tau,2}(x) &= \text{rev}(x), & \Theta_{\tau,2} &= \varnothing, \\
s_{\tau,3}(x; c) &= x \mathbin{+\!\!+} c, & \Theta_{\tau,3} &= \Sigma_{\text{sp}}.
\end{aligned}$$

Here, ϵ denotes the empty string, rev reverses a string, and Σ_{sp} is the set of special characters. When ingredient pool reuse is disabled, the schema is sampled uniformly:

$$p_\tau(i) = \frac{1}{3}, \quad i \in \{1, 2, 3\}.$$

The parameter distributions are

$$\begin{aligned} q_{\tau,1}(*) &= 1, \\ q_{\tau,2}(*) &= 1, \\ q_{\tau,3}(c) &= \frac{1}{|\Sigma_{\mathrm{sp}}|}, \quad c \in \Sigma_{\mathrm{sp}}. \end{aligned}$$

B HYPERPARAMETER SETTINGS

Table 5 provides the detailed hyperparameter configurations used across the test-suite expansion and reduction stages of the VeriScale pipeline.

Table 5: Detailed hyperparameter settings for the VeriScale pipeline.

Parameter	Value	Description
Test-Suite Expansion		
Seed Generation		
`--rounds`	1	Number of LLM generation rounds for each task
`--candidates_per_round`	40	Number of candidates generated per round for each task
`--example_limit`	5	Maximum number of accept/reject examples in prompt
Type-aware Mutation		
`--max_mutations_per_input`	15	Upper limit of mutation samples per seed input
`--mutation_multi_step_size`	5	Maximum number of consecutive steps per mutation
`--mutation_ingredient_prob`	0.3	Probability of ingredient reuse during mutation
Adversarial Implementation Synthesis		
`--max_adver_impl`	5	Maximum number of adversarial implementations per specification
Test-Suite Reduction		
Boundary-Preserving Reduction		
`--MAX_REJECT_INPUTS_PER_TASK`	50	Maximum number of unexpected inputs retained per task
`--KEEP_PER_CRITICAL_BUCKET`	1	Number of unexpected inputs retained from each boundary bucket
Adversary-Killing Reduction		
`--MAX_ACCEPT_TEST_CASES_PER_TASK`	50	Maximum number of expected input-output pairs retained per task

C Prompt Templates

This appendix provides the detailed prompt templates used for benchmark construction and the unexpected-output ablation. We present the exact instructions for seed generation, problem decomposition, specification generation, adversarial implementation synthesis, and direct unexpected-output generation.

Prompt Template for Seed Generation

You are an expert at generating diverse candidate inputs for Lean4 code verification tasks.

Return ONLY a JSON array. Each element must strictly follow:
{"input": {"param1": value1, "param2": value2, ...}}

Rules:
1. Do not output markdown, prose, comments, or code fences.
2. Keys in each 'input' object must exactly match the function parameter names: no missing keys, no extra keys.
3. Values must be JSON−serializable and type−compatible with the declared Lean parameter types.
4. Treat the ground−truth precondition as the valid−input boundary: generate both valid and invalid inputs, especially boundary cases.
5. Include both likely−valid and likely−invalid inputs. Treat likely−invalid as semantically violating the precondition, not malformed JSON.
6. Generate diverse and challenging edge−case candidates. Return exactly the number of candidates requested by the user.

Generate candidate inputs for this Lean4 programming verification task.
Task description:
{description}

Ground−truth precondition:
{precond}

Use of precondition:
– It defines the semantic boundary of valid inputs.
– Generate both inputs that satisfy it and inputs that violate it.
– Focus on hard boundary/edge cases around this condition.
– "likely−invalid" means: still valid JSON and type−compatible, but likely violates the precondition.
Validity mix target:
– total candidates: {candidate_count}
– likely−invalid target: {invalid_target}
– likely−valid target: {valid_target}
– If constraints make exact ratio hard, prioritize exact total count and boundary coverage.

Function parameters (JSON):
{parameters}

Example likely−valid inputs:
{test_examples}

Example likely−invalid inputs (maybe no examples available, but generate if possible):
{reject_examples}

Output format MUST be exactly a JSON array of objects:
[{{"input": {{"param": value, ...}}}}, ...]

Prompt Template for Problem Decomposition

Role
You are a mathematical modeling expert. Your task is to analyze a programming problem description and produce a structured mathematical model that captures its formal meaning.

Input
– **Problem Description**: A natural language description of a programming problem.

Instructions
1. Identify the Input information:
   a. contents (decleration of the input variables)
   b. constraints of the input
      – type/data structure of the input variables
      – properties of the input variables must satisfy, mark each property with one of the following two keywords
         (E: Explicit, properties that are stated explicitly in the problem descripition)
         (H: Hypothesis, properties that may be included in the constraints)
2. Identify the Output information:
   a. contents (decleration of the output variables, you can declare them as 'res1, res2, ...')
   b. constraints of the output
      – type/data structure of the output variables
      – properties of the output variables must satisfy, the relationship between the input and the output (eg. elements/ordering preserving, elements inclusion, etc.)
3. A detailed description of the type of input and output:
   – Figure out the data struture, consider list/tuples and etc.
   – Consider the type of the variables, for example, real numbers, integers, string and etc.
4. A detailed description of the relationship between the input and the output. You may think through:
   – Does every element of the output originate from the input?
   – Does the output preserve any ordering from the input?
   – Does the output contain all elements from the input that meet some conditions?
5. Unless explicitly stated, do NOT mark non–empty, integer–only, positivity, uniqueness, sortedness and etc. with 'E'.
6. Do not add any explanations of the constraints.

Output Format
You MUST structure your response strictly as follows:

Input:
   a. {contents}
   b. {constraints}
      – {constraints1}
      – {constraints2}
      – ...
Output:
   a. {contents}
   b. {constraints}
      – {constraints1}
      – {constraints2}
      – ...

Now perform the following task:
– **Problem Description**:
{problem_description}

Prompt Template for Specification Generation

Role
You are an expert in Lean4 programming. Your task is to analyze a programming problem description and produce the Lean4 code of its precondition and postcondition.

Input
– **Problem Description**: A natural language description of a programming problem.
– **Input**: Input of the problem, and constraints on the input.
– **Output**: Output of the problem, and constraints on the output.
– **Precondition Function Signature**: Function signature of the precondition.
– **Postcondition Function Signature**: Function signature of the postcondition.

Instructions
1. Generate the Lean4 code for the precondition and postcondition according to the Input and Output information. Both the Input and Output have two parts: contents and constraints. Note that you should include all the information provided for you.
2. You should carefully analyze the problem description to help you identify gaps and omissions, thereby capturing any missing details within the Input and Output.
3. Define the Lean4 precondition and postcondition by using the provided function signature.
4. Only translate the constraints of the precondition marked with 'E'. Neglect the constraints marked with 'H'.
5. If auxiliary functions are needed, put them directly before the precondition and postcondition.

Output Format
You MUST structure your response strictly as follows:
```lean
-- Precondition Implementation
{precondition auxiliary functions, if any. precondition implementation}

-- Postcondition Implementation
{postcondition auxiliary functions, if any. postcondition implementation}
```

Now perform the task below:
– **Problem Description**:
{problem_description}
– **Input**:
{input}
– **Output**:
{output}
– **Precondition Function Signature**:
{precond_signature}
– **Postcondition Function Signature**:
{postcond_signature}

Prompt Template for Adversarial Implementation Synthesis

Role
You are an expert Red Teamer in formal verification and a ”lazy” code hacker. Your objective is to perform ”Specification Gaming”: identify underspecification loopholes within a given ‘post−condition‘ in Lean4, and write degenerate code to exploit these vulnerabilities.

Input
– ∗∗Problem Description∗∗: A natural language description of a programming problem.
– ∗∗Pre−condition∗∗: Defines valid inputs. Assume inputs always pass this.
– ∗∗Post−condition∗∗: The flawed contract your output MUST satisfy.
– ∗∗Implementation Signatures∗∗: The function signature of the adversarial functions.

Instructions
1. You MUST NOT write a genuine algorithmic solution to the ‘Problem Description‘. ∗∗If your code actually solves the true problem perfectly, you have FAILED your mission.∗∗ Your code MUST produce fundamentally WRONG outputs according to the natural language problem, BUT these wrong outputs must magically trick the ‘post−condition‘ into evaluating to ‘True‘ (or being provable).
2. Carefully analyze the ‘post−condition‘. What crucial constraints from the ‘Problem Description‘ did it forget to check? You may consider
  – Did it check ‘List.length‘ but forget to check elements?
  – Did it check ‘List.Sorted‘ but forget to ensure elements belong to the original list?
  – Did it use ‘∃‘ without bounding the witness?
  – Did it use ‘∀ x ∈ xs‘ but forget the empty−list case is vacuously true?
3. Here are some adversarial strategies you may consider:
  – ∗Constant Return∗: Return meaningless constants (‘0‘, ‘#[]‘, ‘[]‘, ‘””‘, ‘#[0, 0]‘, ‘false‘, ‘none‘) if the post−condition fails to bind the output to the input.
  – ∗Input Echoing∗: Return the input parameters directly unmodified (e.g., ‘fun xs => xs‘).
  – ∗Trivial Synthesis∗: Return ‘List.replicate n 0‘, ‘List.range n‘, ‘Array.mkArray n 0‘, or entirely fabricated data that satisfies the shape/type/length constraints.
  – ∗Malicious Deletion/Modification∗: If the post−condition only checks for the absence of something (e.g., ”no duplicates”, ”no negative elements”), delete everything and return ‘[]‘ / ‘#[]‘.
  – ∗Edge Case Exploitation∗: Force empty or trivial states (e.g., empty list, ‘0‘, ‘none‘) that make universally quantified checks (‘∀ x ∈ []‘, ...) vacuously true.
4. Your adversarial functions MUST be pure and total Lean4 functions. You MUST NOT:
  – declare new ‘axiom‘s, use ‘sorry‘, or introduce ‘opaque‘ definitions to bypass verification
  – use ‘unsafe‘, ‘partial‘, or non−terminating recursion to avoid producing a real value
  – use ‘@[implemented_by ...]‘, ‘@[extern ...]‘, or any attribute that replaces the compiled behavior of a function
  – register new ‘instance‘s that shadow or override existing typeclass resolution for standard types (‘Eq ‘, ‘LE‘, ‘LT‘, ‘Decidable‘, ‘Ord‘, ‘BEq‘, ‘Hashable‘, etc.)
  – rely on ‘IO‘, ‘ST‘, ‘IO.Ref‘, or any monadic side effects that persist outside the function call
  – use noncomputable definitions or use ‘choose‘, ‘choose_spec‘ to define the adversarial functions
  If any strategy requires modifying the ambient environment, DO NOT use it.
5. You MUST NOT use type−level / typeclass / coercion tricks to fake post−condition satisfaction. In particular, DO NOT:
  – define custom ‘structure‘s / ‘inductive‘ types as outputs purely so that a custom ‘DecidableEq‘, ‘LE‘, ‘LT‘, or ‘BEq‘ instance makes the post−condition trivially true
  – override or re−declare instances such as ‘BEq‘, ‘DecidableEq‘, ‘LE‘, ‘LT‘, ‘Ord‘, ‘HAdd‘, ‘HSub‘, ‘ Membership‘, ‘GetElem‘ for standard types
  – return proxy/wrapper/sentinel values (e.g., via ‘Subtype‘ with a bogus proof, or a wrapper with a custom equality) that are not semantically valid outputs
  – exploit ‘Decidable‘ instances whose ‘decide‘ always returns ‘isTrue‘
  Outputs must be plain values of the exact types expected by the specification (‘Nat‘, ‘Int‘, ‘String‘, ‘ List α‘, ‘Array α‘, ‘Option α‘, tuples of these, etc.).
6. You must generate exactly ∗∗5∗∗ different adversarial implementations. Try to use a ∗different∗ adversarial strategy from the list above for each one to probe different potential loopholes.
7. The function signatures of your adversarial implementations MUST match the implementation signature provided. Distinguish each adversarial function by appending the suffix ‘i‘ to its name, where i denotes the index of the i−th adversarial implementation.
8. DO NOT add any explanations.

Output Format

You MUST structure your response strictly as follows. Do not include any explanations, and you must provide **5** adversarial implementations.

```lean
-- Adversarial Implementation 1
{adversatial function 1}

-- Adversarial Implementation 2
{adversatial function 2}

-- Adversarial Implementation 3
{adversatial function 3}

-- Adversarial Implementation 4
{adversatial function 4}

-- Adversarial Implementation 5
{adversatial function 5}
```

Now perform the task below:
- **Problem Description**:
{problem_description}
- **Pre-condition**:
```lean
{pre_condition}
```
- **Post-condition**:
```lean
{post_condition}
```
- **Implementation Signatures**:
```lean
{impl_signature}
```

Prompt Template for Direct Unexpected-Output Generation

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

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

Generate negative output test cases for the following problem. You only need to change the expected output in the ”Expected Test Cases” to an unexpected one.

Problem description
{problem_description}

Expected Test Cases
{positive_tests_json}

For each test case, provide:
– Return a JSON array with exactly the same number of items as ”Expected Test Cases”.
– Keep the items in the exact same order as ”Expected Test Cases”. Do not reorder, drop, merge, or duplicate any item.
– For each item, copy the ‘input‘ field exactly as it appears in ”Expected Test Cases”.
– For each item, generate the ‘unexpected‘ field as a list of incorrect outputs that violate the expected behavior. The field ‘number_unexpected_output‘ in ”Expected Test Cases” specifies the exact number of incorrect outputs expected for that item; you must generate precisely that number.
– Preserve the one−to−one correspondence between each output item and its source item in ”Expected Test Cases”.

Response uses json format with the following structure:
‘‘‘json
[
  {{
    ”input”: input_values,
    ”unexpected”: [unexpected_output_values]
  }},
  ...
]
‘‘‘