# VeriScale: Adversarial Test-Suite Scaling for Verifiable Code Generation

**Yifan Bai** [1*] **Xiaoyang Liu** [1*] **Zihao Mou** [2] **Guihong Wang** [1] **Jian Yu** [3] **Shuhan Xie** [4] **Yantao Li** [5] **Yangyu Zhang** [6] **Jingwei Liang** [1 7] **Tao Luo** [1 7 8]

## Abstract

As large language models (LLMs) are increasingly deployed for software engineering, constructing high-quality benchmarks is crucial for evaluating not just the functional correctness, but also the formal verifiability of generated code. However, existing benchmarks are limited by the quantity and quality of positive and negative test cases, leading to an overestimation of model capabilities in generating specifications and implementations. To address this, we propose VeriScale, a novel framework driven by the adversarial implementations. It consists of two stages: test-suite expansion to construct diverse and challenging test cases, and test-suite reduction to distill them into compact yet discriminative suites. While VeriScale is general, we instantiate it on VERINA to construct VERINAPLUS, which expands the original test suites by over 83×, and VERINALITE, a lightweight 14× variant. Our experiments across eight state-of-the-art LLMs demonstrate that VERINAPLUS exposes substantial model weaknesses hidden by the original benchmark, evidenced by sharp score drops on both SpecGen and CodeGen tasks, whereas VERINALITE maintains this discriminative power at a fraction of the evaluation cost. The enhanced benchmarks and source code are publicly available at https://github.com/XiaoyangLiu-sjtu/VeriScale.

[*]Equal contribution [1]School of Mathematical Sciences, Shanghai Jiao Tong University [2]School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen [3]School of Mathematics, Jilin University [4]School of Mathematical Sciences, Tongji University [5]Zhiyuan College, Shanghai Jiao Tong University [6]School of Future Technology, South China University of Technology [7]Institute of Natural Sciences, Shanghai Jiao Tong University [8]MOE-LSC, CMA-Shanghai, Shanghai Jiao Tong University. Correspondence to: Jingwei Liang <jingwei.liang@sjtu.edu.cn>, Tao Luo <luotao41@sjtu.edu.cn>.

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

## 1. Introduction

The rapid advancement of large language models (LLMs) has reshaped software engineering, particularly in program synthesis and code generation (Chen et al., 2021; OpenAI et al., 2024). However, LLM-generated programs remain prone to subtle logical errors (Pearce et al., 2025), limiting their deployment in high-stakes settings. To address this reliability challenge, verifiable code generation has emerged as a principled paradigm in which models are required to jointly synthesize formal specifications, code implementations, and machine-checkable proofs to establish their consistency (Sun et al., 2024; Ye et al., 2026). By integrating machine-checkable correctness guarantees into the generation process, this paradigm offers a promising path toward more trustworthy LLM-based software development.

To rigorously evaluate this emerging paradigm, recent benchmarks such as CLEVER (Thakur et al., 2026) and VERINA (Ye et al., 2026) have adopted Lean (Moura & Ullrich, 2021) as their core verification backend. Unlike verification frameworks such as Dafny (Leino, 2010) and Verus (Lattuada et al., 2023), which heavily leverage SMT solvers (de Moura & Bjørner, 2008) for automated theorem proving with less explicit proof steps, Lean operates as an interactive theorem prover (ITP). This distinction enables the construction of proofs with explicit intermediate steps, thereby facilitating the direct assessment of model capabilities in proof generation (ProofGen). For specification generation (SpecGen) and code generation (CodeGen), these benchmarks primarily rely on test cases as the empirical evaluation mechanism (Chen et al., 2021; Ye et al., 2026).

However, this reliance on test-suite evaluation exposes a central limitation: the reliability of the assessment depends heavily on the quantity and quality of the underlying test cases. In current verifiable code generation benchmarks, each problem is typically accompanied by only a limited number of **positive test cases**, i.e., **expected input-output pairs**. More importantly, high-quality **negative test cases** are even more limited, including **unexpected inputs and unexpected outputs** that are essential for testing whether generated specifications correctly reject invalid behaviors. As a result, models may satisfy the sparse test cases without

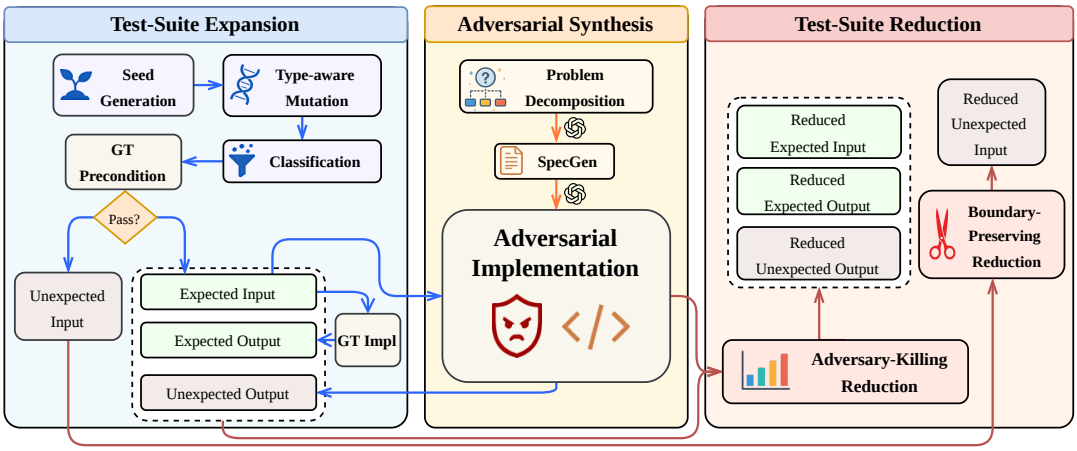


*Figure 1.* **Overview of the VeriScale framework for adversarial test-suite scaling.** Driven by the adversarial implementations, the framework scales verifiable code generation benchmarks through two core stages: test-suite expansion to ensure rigorous boundary coverage, and test-suite reduction to optimize evaluation efficiency without sacrificing discriminative power.

capturing the program intent, allowing incorrect implementations or unsound and incomplete specifications to achieve inflated pass rates and leading to an overestimate of model capabilities.

To bridge this gap, we propose VeriScale, an adversarial framework that systematically scales test-suite evaluation for verifiable code generation. VeriScale first combines LLM-based seed generation with type-aware mutation to construct a large pool of candidate inputs. These inputs are then classified as either expected or unexpected according to the ground-truth preconditions. For the former, VeriScale executes the reference implementation to derive the corresponding outputs, thereby forming robust expected input-output pairs.

**The core innovation of VeriScale lies in constructing adversarial implementations to drive both the construction of unexpected outputs and the reduction of expected test cases.** Instead of directly prompting LLMs to generate arbitrary unexpected outputs, VeriScale synthesizes adversarial implementations designed to exploit weaknesses in LLM-generated specifications, executing them on expected inputs. Outputs are retained only when they differ from the reference outputs while being falsely accepted by the generated postconditions, ensuring they are genuinely informative for exposing specification flaws. Finally, VeriScale repurposes these implementations in a reduction stage, distilling the expanded cases into compact yet discriminative test suites through boundary-preserving and adversary-killing reduction.

To evaluate our approach, we apply VeriScale to VERINA, resulting in the VERINAPLUS and VERINALITE benchmarks. The empirical results underscore the necessity of adversarial test scaling. Notably, GPT-5.5 experiences a severe performance degradation on VERINAPLUS, plummeting from 68.78% to 44.44% on SpecGen and from 96.83% to 86.24% on CodeGen. This stark contrast highlights how the original benchmark significantly overestimates true model capabilities. Furthermore, model performance on VERINALITE closely mirrors that on VERINAPLUS, confirming that our reduction strategy successfully eliminates redundant test cases while preserving discriminative rigor.

Our main contributions are as follows:

1. We propose VeriScale, a framework that leverages adversarial implementations to drive both the systematic expansion and the reduction of test suites for verifiable code generation.
2. We instantiate VeriScale on VERINA to construct VERINAPLUS, expanding the original test suites by over $83\times$, alongside VERINALITE, a lightweight $14\times$ variant.
3. We show that VERINAPLUS reveals substantial weaknesses overlooked by the original benchmark in both SpecGen and CodeGen, while VERINALITE retains the discriminative power at a fraction of the evaluation cost.

## 2. Related Work

**Benchmarks for Verifiable Code Generation.** The evaluation of LLM-generated programs has evolved from functional correctness toward formal verifiability. Early benchmarks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and LiveCodeBench (Jain et al., 2025) primarily assess code generation through unit tests, but do not evaluate formal specifications or proofs. Recent verification-oriented benchmarks, including Dafny-Synthesis (Misu et al., 2024),

Clover (Sun et al., 2024), DafnyBench (Loughridge et al., 2025), miniCodeProps (Lohn & Welleck, 2024), FVAPPS (Dougherty & Mehta, 2025), and VERINA (Ye et al., 2026), incorporate formal reasoning tasks in languages such as Dafny and Lean.

Among these benchmarks, VERINA provides a comprehensive benchmark for verifiable code generation, supporting the evaluation of SpecGen, CodeGen, and ProofGen. VeriScale is complementary to these efforts: instead of introducing new tasks, it strengthens existing benchmarks by scaling expected input-output pairs, unexpected inputs, and high-quality unexpected outputs for more discriminative test-suite evaluation.

**Scaling Test-Suite Evaluation.** Test-suite quality is crucial for reliable code generation evaluation, as limited unit tests may allow incorrect programs to pass by missing corner cases. EvalPlus (Liu et al., 2023) is a representative effort that augments HumanEval (Chen et al., 2021) with large-scale test cases by combining seed generation and type-aware mutation. Mutation-based generation can mitigate benchmark leakage, since newly generated tests are less likely to have been memorized during training (Xu et al., 2024; Satvaty et al., 2026). Adversarial tests have also proven effective in exposing weaknesses in code generation and program repair systems (Yao et al., 2024; Ye et al., 2025).

VeriScale is inspired by these methods but targets verifiable code generation, where evaluation requires richer signals than expected input-output pairs alone. In particular, SpecGen evaluation needs unexpected inputs and unexpected outputs to test whether generated specifications correctly reject invalid behaviors. Therefore, VeriScale extends test-suite scaling with precondition-guided classification and adversarial implementation synthesis, producing more informative negative cases to strictly evaluate the SpecGen task.

## 3. Methodology

In this section, we detail VeriScale, a framework designed to scale test-suite evaluation for verifiable code generation, 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 support both CodeGen and SpecGen evaluation by checking intended behaviors, while unexpected inputs and unexpected outputs test whether generated specifications reject 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 D.

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 $D_\tau$.
- $\mathcal{O}$ 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 $p_\tau$ over $\mathcal{O}_\tau$, specifying how a mutation schema is selected.
- $q$ maps each pair $(\tau, i)$ 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 B for more details of these schemas.

**Precondition-Guided Input Classification.** To classify the candidate inputs, VeriScale operates in alignment with Verina (Ye et al., 2026), 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.

Figure 2 illustrates this bidirectional pipeline in practice. For the isolated expected inputs, we execute the reference implementation to compute their corresponding outputs. This completes the expected input-output pairs, establishing a ground truth for evaluating functional correctness.

**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 3, 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).

**EXAMPLE OF CANDIDATE INPUT CLASSIFICATION**

**Problem Description:** (VERINA #advanced_7) This task requires writing a Lean 4 function that converts a binary number represented as a list of digits into its corresponding decimal value. The list is ordered in big-endian format, meaning the most significant digit comes first.

**Ground-Truth Precondition:**

```
def binaryToDecimal_precond (digits :
List Nat) : Prop := digits.all (fun d =>
 d = 0 ∨ d = 1)
```

---

**Candidate Input:** `[1, 2, 1]`

**Syntax Filter:**

`#check binaryToDecimal_precond ([1, 2, 1])` ✔ Passed

**Stage 1: Bidirectional `decide`**

- `#guard decide (binaryToDecimal_precond ([1, 2, 1]))`
  → Failed (Expression did not evaluate to 'true')
- `#guard decide (¬ binaryToDecimal_precond ([1, 2, 1]))`
  ✔ Passed

**Stage 2: Bidirectional `plausible`**

- `example: binaryToDecimal_precond ([1, 2, 1]) := by`
  `unfold binaryToDecimal_precond`
  `simp_all!`
  `plausible`

  → Found a counter-example!
- `example: ¬ binaryToDecimal_precond ([1, 2, 1]) := by`
  `unfold binaryToDecimal_precond`
  `simp_all!`

  ✔ Passed

---

**Final Verdict: Unexpected Input**.

*Figure 2.* **A full trajectory of candidate input classification.** The input `[1,2,1]` will be classified as unexpected because `#guard descide` fails at Stage 1. According to our judging process, this example does not proceed to Stage 2; it is shown here for illustration only.

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

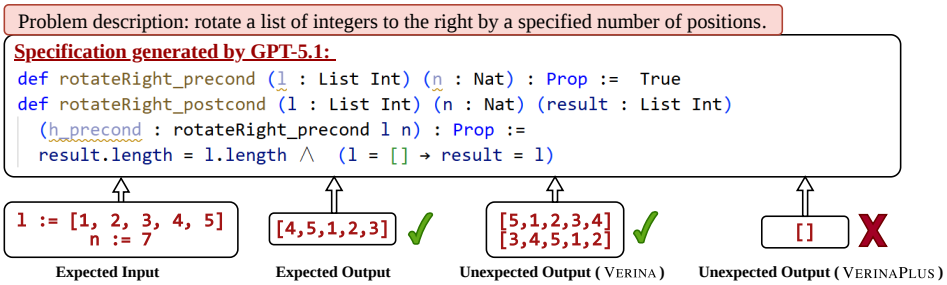


*Figure 3.* **Case study of a flawed specification on VERINA #advanced_3.** The generated postcondition erroneously accepts unexpected outputs from the original dataset, but successfully rejects our adversarially synthesized unexpected outputs.

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 eliminate redundant expected input-output pairs through adversary-killing reduction. Unexpected outputs are implicitly filtered alongside their corresponding inputs.

**Boundary-Preserving Reduction.** For unexpected inputs, we adopt a boundary-preserving reduction strategy rather than relying solely on random sampling or feature diversity. We extract lightweight structural signatures (e.g., empty containers, zero or negative values, mismatched lengths, and ordering patterns) to approximate failure modes. We then 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. Given an expected input, an adversarial implementation is considered "killed" if its output differs from the expected output. 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. A test case that kills more adversarial implementations has greater potential to reveal weaknesses in the LLM-generated specifications. 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. Figure 4 shows an example of adversary-killing reduction.

## 4. Experiments

### 4.1. The Augmented Benchmarks

We instantiate VeriScale on the VERINA dataset (Ye et al., 2026) to construct the augmented benchmarks VERINAPLUS and VERINALITE. During the expansion stage, we employ GPT-5.3-Codex for seed generation, problem decomposition, and adversarial implementation synthesis. To obtain highly diverse candidate specifications, we utilize an array of widely adopted prior models: Claude-Haiku-4.5, Claude-Sonnet-4.5, GPT-5.1, GPT-4.1, Qwen3-Max, and DeepSeek-V3.2. Detailed hyperparameter configurations of the expansion and reduction stages are provided in Appendix C. In total, the API cost incurred during the benchmark expansion phase was $58.89.

As demonstrated in Table 1, VERINAPLUS significantly scales up the test volume by over 83× compared to the original baseline to ensure rigorous boundary coverage, whereas VERINALITE optimizes computational overhead via the aforementioned reduction strategies to provide a lightweight 14× variant. Notably, the augmented benchmarks also consistently maintain 100% code coverage.

*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.

| Dataset | Expected Input-Output | Unexpected Output | Unexpected Input |
|---|---|---|---|
| VERINA | 5.89 (2–13) | 12.69 (2–34) | 0.65 (0–7) |
| VERINAPLUS | 370.07 (5–705) (×62.83) | 1114.01 (15–2973) (×87.79) | 119.00 (0–684) (×183.08) |
| VERINALITE | 52.34 (5–61) (×8.89) | 202.35 (15–467) (×15.95) | 15.80 (0–50) (×24.31) |

**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 4.* **An example of adversary-killing reduction.** The illustrated case kills six out of seven adversarial implementations, making it a high-quality case.

### 4.2. Evaluation Setup

We evaluate eight state-of-the-art LLMs, including 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.

These models are assessed across the VERINA, VERINAPLUS, and VERINALITE benchmarks, focusing on the SpecGen and CodeGen tasks. To ensure a rigorous assessment, we employ the official evaluation harness provided by VERINA (Ye et al., 2026) to compute and report the pass@1 scores. However, we intentionally remove the original 2-shot setting to facilitate the evaluation of the entire dataset.

All experiments are implemented using Lean v4.24.0. The evaluation was conducted on a server equipped with dual Intel Xeon Platinum 8380 CPUs, providing a total of 80 physical cores.

### 4.3. Evaluation Results

To systematically interpret our findings, we organize the evaluation results across three critical dimensions: the effectiveness of the augmented benchmarks in exposing hidden model vulnerabilities, the profound robustness gap between SpecGen and CodeGen, and the computational efficiency of our test-suite reduction strategy.

**Effectiveness of the Augmented Benchmarks.** As illustrated in Figure 5, all models exhibit significant performance degradation across both SpecGen and CodeGen tasks on VERINAPLUS and VERINALITE compared to the VERINA baseline. For instance, the top-performing model, GPT-5.5, achieves a code score of 96.83 and a specification soundness and completeness score of 68.78 on the baseline, which drop sharply to 86.24 and 44.44 on VERINAPLUS, respectively. This consistent decline across all models demonstrates that the original benchmark overestimates model capabilities due to underspecified test cases. By addressing this critical limitation, our adversarial expansions successfully systematically expose previously hidden vulnerabilities, providing a much more rigorous evaluation of the SpecGen and CodeGen tasks.

Furthermore, a closer examination of the SpecGen evaluation reveals a striking reduction in evaluation uncertainty, as evidenced by the significantly narrowed error bars on the

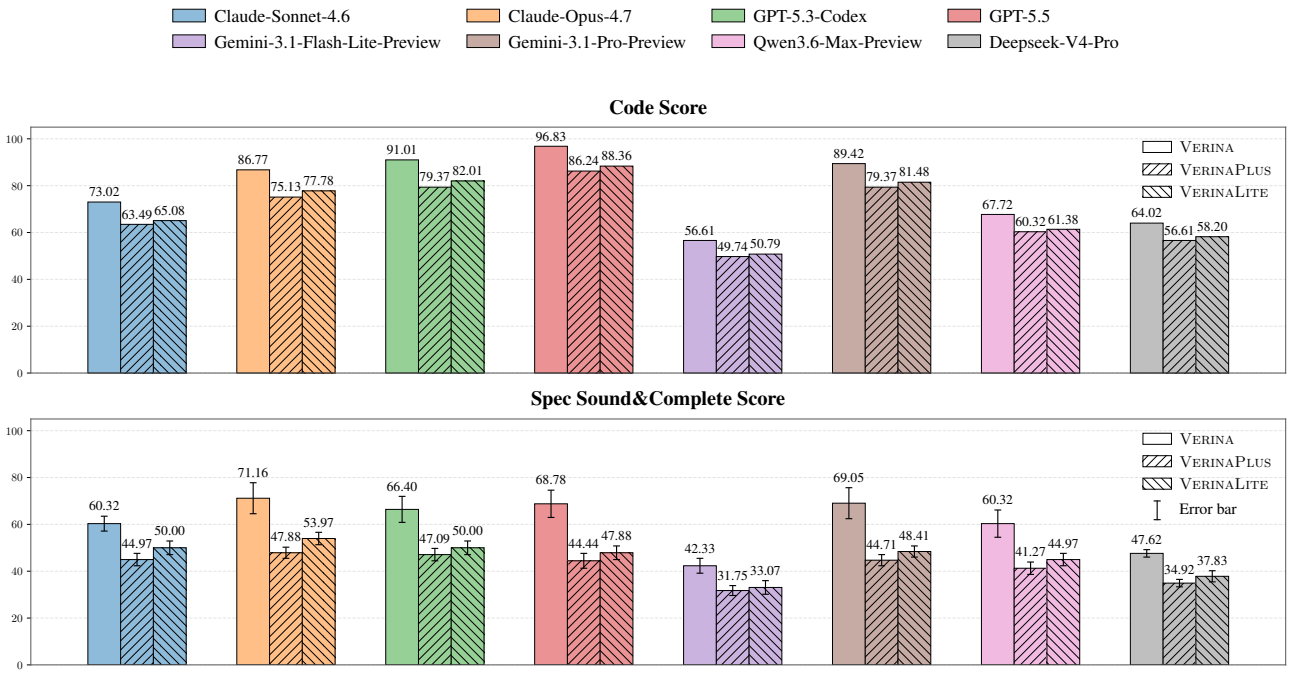


*Figure 5.* **Performance comparison of eight models on VERINA, VERINAPLUS, and VERINALITE.** The top panel shows the Code Score, while the bottom panel displays the Spec Sound&Complete Score. For the specification evaluation, we include error bars indicating the lower bound (treating unknown cases as not holding) and upper bound (treating unknown cases as holding).

augmented benchmarks. In the original VERINA baseline, the scarcity of test cases frequently leads to indeterminate verification outcomes, resulting in wide margins between the upper and lower bounds. For example, Claude-Opus-4.7 exhibits a substantial gap of 13.23% (77.78% vs. 64.55%) on the baseline. However, under the rigorous scrutiny of VERINAPLUS, this ambiguity is effectively resolved, drastically shrinking the gap for the same model to merely 4.76% (50.26% vs. 45.50%). This consistent compression of error margins across all evaluated models demonstrates that our adversarial scaling not only exposes model flaws, but also transforms a previously ambiguous evaluation into a highly definitive and statistically reliable assessment.

**Robustness Gap between SpecGen and CodeGen.** While the augmented benchmarks universally depress scores, this degradation is far from uniform: the decline in CodeGen is drastically outpaced by a precipitous drop in SpecGen. Taking the top-performing GPT-5.5 as a representative example again, the model experiences a moderate relative decline of roughly 10.59% in CodeGen when moving from the VERINA baseline to VERINAPLUS, yet suffers a staggering 24.34% relative reduction in its specification soundness and completeness score. This consistent disparity, observed across all evaluated LLMs, strongly indicates that synthesizing sound and complete specifications is fundamentally more challenging than generating functional code.

This stark contrast highlights a fundamental limitation in the current state-of-the-art models. Thanks to vast exposure to standard software repositories during pre-training, advanced LLMs have developed a remarkable proficiency for synthesizing functional programming logic. However, this empirical success is heavily reliant on statistical pattern matching and does not seamlessly transfer to the rigorous domain of formal verification. Synthesizing sound and complete specifications demands strict mathematical reasoning and a global awareness of constraints. Consequently, when tasked with formalizing properties that defend against adversarial edge cases, the reasoning capabilities of these models remain highly fragile.

**Efficiency of Test-Suite Reduction Strategy.** Beyond evaluation rigor, the practical utility of a benchmark heavily depends on its computational overhead. As detailed in Table 2, exhaustively evaluating models on the expanded VERINAPLUS incurs a substantial computational penalty due to the sheer volume of test cases. Specifically, this exhaustive approach multiplies the average time overhead to 1.80 and 1.50 times the baseline for the SpecGen and CodeGen tasks, respectively. Such pronounced time constraints can significantly hinder the widespread utilization of the benchmark.

Crucially, our minimized benchmark, VERINALITE, effectively resolves this scalability issue without sacrificing eval-

*Table 2.* **Evaluation time on VERINA, VERINAPLUS, and VERINALITE.** We report wall-clock time in seconds, with the average row additionally highlighting the relative time multipliers compared to the VERINA baseline.

| Model | SpecGen | | | CodeGen | | |
|---|---|---|---|---|---|---|
| | VERINA | VERINAPLUS | VERINALITE | VERINA | VERINAPLUS | VERINALITE |
| Claude-Sonnet-4.6 | 996 | 1772 | 1280 | 268 | 356 | 241 |
| Claude-Opus-4.7 | 1324 | 2758 | 2102 | 250 | 394 | 269 |
| GPT-5.3-Codex | 1018 | 1839 | 1395 | 252 | 413 | 327 |
| GPT-5.5 | 1191 | 2100 | 1669 | 264 | 413 | 284 |
| Gemini-3.1-Flash-Lite-Preview | 790 | 1333 | 1053 | 190 | 305 | 205 |
| Gemini-3.1-Pro-Preview | 1214 | 2081 | 1645 | 254 | 396 | 267 |
| Qwen3.6-Max-Preview | 962 | 1732 | 1333 | 212 | 329 | 227 |
| DeepSeek-V4-Pro | 757 | 1261 | 935 | 260 | 311 | 217 |
| **Average** | 1032 | 1860 (×1.80) | 1427 (×1.38) | 244 | 365 (×1.50) | 255 (×1.05) |

uative rigor. It drastically reduces the time overhead to merely 1.38 and 1.05 times the original baseline. More importantly, while achieving this efficiency, VERINALITE simultaneously maintains a performance degradation profile highly similar to the full VERINAPLUS suite (as evidenced in Figure 5). This optimal balance proves that our reduction strategies successfully prune redundant cases while fully retaining the essential diagnostic power, thereby establishing VERINALITE as a highly practical benchmark for evaluations of verifiable code generation.

## 5. Conclusion

In this paper, we proposed VeriScale, an adversarial framework that systematically expands and reduces test suites to rigorously evaluate verifiable code generation. We instantiated VeriScale on the original VERINA benchmark to construct VERINAPLUS (expanded by over 83×) and VERINALITE (a 14× variant). Our evaluations demonstrate that VERINAPLUS uncovers substantial weaknesses in state-of-the-art LLMs across both SpecGen and CodeGen tasks that original benchmarks fail to detect, while VERINALITE achieves the same discriminative power with significantly reduced evaluation cost. Together, these augmented benchmarks provide a crucial, highly reliable standard for advancing LLM-based verifiable code generation.

## Impact Statement

This paper advances verifiable code generation by introducing VeriScale, a scalable framework for more rigorous benchmark evaluation. By adversarially expanding and reducing test suites, VeriScale helps identify weaknesses in LLM-generated specifications and implementations, supporting more reliable code generation. However, evaluation on large-scale test suites requires substantial computational resources. In addition, stronger benchmarks may encourage optimization for benchmark performance rather than improvements in reasoning and verification capabilities.

## References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

de Moura, L. and Bjørner, N. Z3: An efficient smt solver. In Ramakrishnan, C. R. and Rehof, J. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Dougherty, Q. and Mehta, R. Proving the coding interview: A benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2502.05714.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=sD93GOzH3i5.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., and Hawblitzel, C. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. doi: 10.1145/3586037. URL https://doi.org/10.1145/3586037.

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

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. 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.

Liu, X., Dong, Z., Bai, Y., Li, Y., Liu, Y., and Luo, T. Decompose, structure, and repair: A neuro-symbolic framework for autoformalization via operator trees, 2026. URL https://arxiv.org/abs/2604.19000.

Lohn, E. and Welleck, S. minicodeprops: a minimal benchmark for proving code properties. In *Neurips Safe Generative AI Workshop 2024*, 2024. URL https://openreview.net/forum?id=6QFe3vPbYZ.

Loughridge, C. R., Sun, Q., Ahrenbach, S., Cassano, F., Sun, C., Sheng, Y., Mudide, A., Misu, M. R. H., Amin, N., and Tegmark, M. Dafnybench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=yBgTVWccIx.

Martinez, M. and Monperrus, M. Astor: Exploring the design space of generate-and-validate program repair beyond genprog. *J. Syst. Softw.*, 151(C):65–80, May 2019. ISSN 0164-1212. doi: 10.1016/j.jss.2019.01.069. URL https://doi.org/10.1016/j.jss.2019.01.069.

Misu, M. R. H., Lopes, C. V., Ma, I., and Noble, J. Towards ai-assisted synthesis of verified dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3643763. URL https://doi.org/10.1145/3643763.

Moura, L. d. and Ullrich, S. The lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Łukasz Kaiser, Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, J. H., Kiros, J., Knight, M., Kokotajlo, D., Łukasz Kondraciuk, Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., de Avila Belbute Peres, F., Petrov, M., de Oliveira Pinto, H. P., Michael, Pokorny, Pokrass, M., Pong, V. H., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K.,

Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M. B., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., and Zoph, B. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Park, J., Winterer, D., Zhang, C., and Su, Z. 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.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM*, 68(2):96–105, January 2025. ISSN 0001-0782. doi: 10.1145/3610721. URL https://doi.org/10.1145/3610721.

Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., and Irving, G. Red teaming language models with language models. In Goldberg, Y., Kozareva, Z., and Zhang, Y. (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/.

Satvaty, A., Verberne, S., and Turkmen, F. Undesirable memorization in large language models: A survey, 2026. URL https://arxiv.org/abs/2410.02650.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. 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.

Thakur, A., Lee, J., Tsoukalas, G., Sistla, M., Zhao, M., Zetzsche, S., Durrett, G., Yue, Y., and Chaudhuri, S. CLEVER: A curated benchmark for formally verified code generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=IbOacMF5qd.

Xu, R., Wang, Z., Fan, R.-Z., and Liu, P. Benchmarking benchmark leakage in large language models, 2024. URL https://arxiv.org/abs/2404.18824.

Yao, K., Wang, H., Qin, C., Zhu, H., Wu, Y., and Zhang, L. 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.

Ye, H., Yang, A. Z., Hu, C., Wang, Y., Zhang, T., and Le Goues, C. 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.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. VERINA: Benchmarking verifiable code generation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=0A4Uf88pog.

## A. Limitations and Future Work

While VeriScale significantly advances the rigorous evaluation of verifiable code generation, it has several limitations. First, the pipeline relies fundamentally on the availability of ground-truth preconditions and the reference implementations. Second, the quality of the generated unexpected outputs is inherently bounded by the reasoning and formalization capabilities of the LLM acting as the red team. If the model struggles with the strict typing constraints of Lean 4, it may fail to synthesize sophisticated adversarial implementations. Third, the construction phase of the expanded test suites remains highly compute-intensive, requiring massive LLM sampling and Lean executions.

Future work will evolve VeriScale into a closed-loop repair system, using the discovered counterexamples to guide models in iteratively patching flawed specifications. Furthermore, recognizing that SpecGen is fundamentally an autoformalization task severely bottlenecked by data scarcity, we plan to leverage VeriScale as an adversarial data engine. By synthesizing large-scale, high-quality training corpora, we aim to train a dedicated model capable of robust autoformalization. Finally, generalizing VeriScale to other formal verification frameworks, such as Dafny, remains a critical step toward verifiable code generation.

## B. 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

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

The corresponding parameter spaces are

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

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,

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

**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

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

The parameter spaces are

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

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:

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

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 $\Sigma_{\text{char}}$ 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, $\epsilon$ denotes the empty string, rev reverses a string, and $\Sigma_{\text{sp}}$ 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

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

## C. Hyperparameter Settings

Table 3 provides the detailed hyperparameter configurations used across the test-suite expansion and reduction stages of the VeriScale pipeline.

*Table 3.* 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 |

## D. Prompt Templates

This appendix provides the detailed prompt templates utilized across the expansion stage of the VeriScale pipeline. We present the exact instructions formulated for tasks including seed generation, problem decomposition, specification generation, and adversarial implementation synthesis.

**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 $\alpha$‘, ‘Array $\alpha$‘, ‘Option $\alpha$‘, 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}
```