VeriBench: End-to-End Formal Verification Benchmark for AI Code Generation in Lean 4

Brando Miranda [1] **Zhanke Zhou** [1 2] **Allen Nie** [1] **Elyas Obbad** [1] **Leni Aniva** [1] **Kai Fronsdal** [1] **Weston Kirk** [3] **Dilara Soylu** [1] **Andrea Yu** [4] **Ying Li** [4] **Sanmi Koyejo** [1]

Abstract

Formal verification of software is a promising and potentially transformative application of generative AI. Provably correct code would eliminate entire classes of vulnerabilities, mitigate critical system failures, and potentially transform software engineering practices through inherently trustworthy implementation methodologies. To advance this domain, we present **VeriBench**, a carefully curated benchmark for evaluating language-model capabilities in end-to-end code verification, requiring the generation of complete Lean 4 programs—implementations, unit tests, correctness theorems, and formal proofs—derived from reference Python functions or their docstrings. Our evaluation on the 113-task suite—51 HumanEval problems, 42 easy exercises, 10 classical algorithms, and 11 security challenges—shows that current frontier models compile only a small fraction of programs. Claude 3.7 Sonnet achieves compilation on only 12.5%, while LLaMA-70B fails to compile any programs in the Lean 4 HumanEval subset, even with 50 feedback-guided attempts. Notably, among the evaluated approaches, our experiments reveal that a self-optimizing Trace agent architecture achieves compilation rates approaching 60%. **VeriBench** establishes a rigorous foundation for developing AI systems capable of synthesizing provably correct, bug-free code, thereby advancing the trajectory toward more secure and dependable software infrastructure.

[1]Department of Computer Science, Stanford University [2]Department of Computer Science, Hong Kong Baptist University [3]Department of Earth System Science, Stanford University [4]Department of Mathematics, Stanford University. Correspondence to: Brando Miranda <brando9@stanford.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

1. Introduction

Large language models (LLMs) have demonstrated impressive capabilities in code generation, leading to widespread integration into developer workflows. Recent survey results from GitHub report that, on average, 46% of a developer's code is now authored by Copilot across all programming languages, with the proportion rising to 61% in Java (Zhao, 2023). In parallel, a 2024 developer survey finds that 76% of respondents are either already using or planning to use AI tools in their workflows this year (Stack Overflow, 2024). Despite this rapid adoption, confidence in the accuracy and reliability of AI-generated code remains limited. Only 2.7% of developers report that they "highly trust" the output of AI tools, while just 3.3% believe such tools perform "very well" at handling complex development tasks (Stack Overflow, 2024). This tension between high usage and low trust underscores the urgent need for rigorous, automated mechanisms capable of providing formal guarantees about model-generated code–especially for security critical applications.

As a result, AI-generated code is making its way into commercial software systems and may soon occupy a large portion of publicly existing code. Despite their usefulness, LLMs are inherently probabilistic and cannot guarantee the correctness of the code they produce. Consequently, such code often contains bugs, ranging from logical flaws to serious security vulnerabilities (Perry et al., 2023; Team, 2025; Claburn; Pearce et al., 2022). As adoption grows, these errors risk becoming a major obstacle to developer productivity, since human review is typically needed to identify and fix them (Tambon et al., 2024; Nguyen et al., 2022; Srivatsa et al., 2024; GitHub Engineering, 2023).

Formal verification – the process of mathematically proving that an implementation adheres to a precise specification – offers a principled path to guaranteed correctness. Historically, it has been applied mainly in high-assurance settings such as hardware design (processor verification) (Reid, 2016), aerospace avionics (Holzmann & James, 2006), medical-device firmware (Alur et al., 2010), nuclear-power control systems (Linnosmaa et al., 2023), and mission-critical financial or smart-contract infrastructure (Team, 2023), where the cost of failure far outweighs

the cost of formal analysis. A *formal specification* is a mathematical statement of the intended behavior of a program (e.g., `reverse` returns a list whose length equals the length of the input list), and a *machine-checkable proof* is a logical argument that a proof assistant's kernel can verify automatically. Unlike general theorem proving in mathematics, code verification focuses on proving properties about computational processes and their implementations.

LLMs have the potential to democratize formal verification by co-generating implementations, their specifications, and the proofs that connect them, bringing strong correctness guarantees to everyday software development.

To unlock the full potential of verifiable code generation, the development of robust benchmarks is essential to measure progress and guide future research. However, designing such benchmarks is challenging: verifiable code generation encompasses several interdependent components, including the synthesis of code, formal specifications, and accompanying proofs. High-quality data must be curated for each of these components, along with rigorous, task-specific evaluation metrics. Therefore, we use Trace agents (Cheng et al., 2024) to bootstrap the generation of aligned Python functions with docstring specifications and corresponding Lean proofs.

Our goal with **VeriBench** is to close two gaps left open by prior work. First, existing verification datasets—such as VERINA, FVAPPS, CLEVER, DafnyBench, and CloverBench (Chen et al., 2021; Ye et al., 2025; Dougherty & Mehta, 2025; Thakur et al., 2025; Loughridge et al., 2024; Sun et al., 2024)—are populated almost exclusively with textbook algorithms or synthetic exercises. VeriBench is the benchmark *first* to include security-critical programs written by developers: its *SecuritySet* adapts buffer-overflow, privilege-escalation, and race-condition labs from MIT 6.858 (CSAIL, 2024), so models must eliminate real vulnerabilities instead of toy bugs and aim to simulate real code that people want verified in practice. Second, earlier suites report only single-shot or best-of-k scores; they never benchmark an explicit loop that reads verifier feedback, rewrites the artifact, and resubmits until the compilation succeeds. VeriBench, therefore, is the first to illustrate agentic evaluation with a reference Trace-based (Cheng et al., 2024) framework (baseline, self-debug, self-improve) built on generative optimization traces, making feedback-driven, closed-loop verification a primary evaluation.

Besides the *SecuritySet*, VeriBench adds *EasySet* and *CSSet*, which target foundational reasoning and classical algorithms, respectively, giving the benchmark a diverse topic mix and a broad difficulty range (see §3.1). We focus on translating *existing* Python code and its docstrings to Lean because millions of such snippets already live in open-source repos; turning this real-world corpus into machine-checked programs fixes latent bugs directly, whereas starting from a fresh natural-language description would add an unnecessary detour that today's LLMs no longer require. Note that every task already includes a descriptive docstring; researchers who prefer a pure natural-language–to–proof setting can simply ignore the reference implementation and treat VeriBench as an NL-only corpus.

Unlike the static-analysis, linting, and fuzz-testing pipelines already common in industry, Lean 4 supplies compile-time, machine-checked proofs that hold for *all* inputs and compile into the shipped binary, providing end-to-end correctness guarantees that conventional tools cannot match.

Our **contributions** are:

1. **A security-grounded benchmark.** VeriBench is the first Lean 4 dataset to include developer-written, security-critical programs (buffer overflow, privilege escalation, and race condition labs from MIT 6.858), complementing textbook and synthetic tasks.
2. **Balanced task spectrum.** Four subsets—*HumanEval*, *EasySet*, *CSSet*, and *SecuritySet*—cover everything from basic reasoning to classical algorithms and real-world exploits, enabling fine-grained difficulty analysis.
3. **Closed-loop agent evaluation.** We provide a Trace-based reference agent (baseline, self-debug, self-improve) that interacts with the Lean compiler and judge, turning feedback-driven verification into a measurable axis of performance.
4. **Comprehensive Lean artifacts.** Each problem ships runnable Python code, a gold Lean implementation, unit tests, correctness theorems, and machine-checked proofs—yielding the first end-to-end yardstick for provably correct code generation.

2. Related Work

Recent advances in language models have inspired a surge of research on formal code verification, spanning benchmarks, methodologies, and frameworks. In this section, we review progress across three fronts: large-scale benchmarks for evaluating LLM verification capabilities, techniques for improving proof generation and checking, and agentic frameworks that enable modular, self-improving workflows and programmatic reasoning.

Benchmarks for Code Verification. Loughridge et al. (2024) introduce DAFNYBENCH, the first large-scale benchmark for evaluating LLMs in formal software verification. It consists of over 750 Dafny programs (approximately 53K lines of code) stripped of verification "hints," requiring models to regenerate the missing annotations to pass the verifier. Evaluated on GPT-4, GPT-4 Turbo, Claude 3, and others, the best-performing system achieved a success rate of roughly 68%, demonstrating the potential of machine-assisted ver-

ification while highlighting performance variability with respect to program size, hint complexity, and retry strategies. Complementing this line of work, Wang et al. (2024) present THEOREMLLAMA, a framework aimed at enhancing LLM translation into Lean 4. Drawing on over 100K proof examples from the Mathlib4 library, TheoremLlama employs a novel natural-language-to-formal-language (NL-FL) bootstrapping strategy and iterative proof synthesis. This enables the reuse of verified examples as templates for future translations. The framework achieves 36.48% and 33.61% accuracy on the MiniF2F-Valid and MiniF2F-Test benchmarks, respectively—surpassing GPT-4 by more than ten percentage points on both.

To more comprehensively evaluate the different stages of formal development, Ye et al. (2025) introduces VERINA, a 189-task Lean benchmark that jointly assesses LLM-generated code, formal specifications, and machine-checked proofs. While LLMs perform reasonably well on code and spec generation, they continue to struggle with constructing formal proofs, underscoring the need for more targeted training and architectural improvements. Addressing this need, Cao et al. (2025) proposes a decomposition of the informal-to-formal verification pipeline into six subtasks, and releases a dataset of 18K paired examples across five formal specification languages. Each pair includes executable contexts and automated validations. Fine-tuning LLMs on this data significantly improves formal proof synthesis and transfers positively to related tasks in mathematics and programming.

In parallel, Dougherty & Mehta (2025) introduce FVAPPS, a machine-generated Lean 4 benchmark of 4,715 problems stratified by assurance level. Built via a test-driven LLM pipeline, it shows that top models like Claude Sonnet and Gemini 1.5 Pro prove only 30% and 18.5% of theorems, respectively, with human-written solutions still falling short at scale. CLEVER (Thakur et al., 2025) offers a more focused challenge: 161 Lean problems requiring both a formal spec and a correctness-proofed implementation. It's non-computable, spec-agnostic setup avoids test leakage and demands true reasoning—so far, models fully solve only 1 of 161 tasks. Finally, broadening the scope beyond formal proof, Ouyang et al. (2025) introduces KERNELBENCH, a benchmark for generating optimized GPU kernels for 250 PyTorch workloads. Using profiler feedback and examples, iterative loops boost success from 12% to over 70%, showcasing the impact of reinforcement-style correction.

VeriBench distinguishes itself from existing benchmarks by targeting the full pipeline of formal code verification grounded in realistic programming tasks. Unlike FVAPPS, which consists of thousands of machine-generated Lean problems optimized for scale and raw proof success rates, VeriBench uses human-curated Python functions drawn from foundational algorithms and practical programming contexts. Each example is paired with a complete Lean 4 formalization—including functional and imperative implementations, unit tests, correctness theorems, and machine-checkable proofs—emphasizing artifact completeness over sheer volume. Compared to VERINA, which decomposes the verification process into separate subtasks like spec generation and proof synthesis, VeriBench evaluates holistic translation performance: how well models can go from informal code and natural language to executable and provable formal artifacts. Moreover, VeriBench supports the evaluation of agentic systems that iteratively refine their outputs through feedback.

3. VeriBench

3.1. Overview

VeriBench is a benchmark designed to evaluate the end-to-end code verification capabilities of large language models, requiring them to generate complete Lean 4 artifacts from reference Python programs or their accompanying docstrings. It encompasses a broad range of translation components, including function implementations, natural language descriptions, unit tests, theorems, formal proofs, and example input-output behavior.

Instead of relying on a deep embedding of source language semantics, VeriBench adopts a shallow embedding approach, aiming to produce formal representations that faithfully capture the behavior and intent of the original code. By offering a structured and challenging testbed, VeriBench enables rigorous evaluation of model performance in both automated code translation and formal verification, with a particular focus on the correctness, completeness, and provability of the generated Lean 4 artifacts.

Concretely, VeriBench consists of four subsets:

1. **HumanEval** – 51 standard programming puzzles from Chen et al. (2021);
2. **EasySet** – 51 bite-size logic and intro-programming tasks;
3. **CSSet** – 10 classical data-structure and algorithm problems;
4. **SecuritySet** – 11 examples of buffer overflow, privilege escalation, and race condition labs drawn from real code.

This ensures coverage of tasks that range from simple correctness theorems to more complex invariants and algorithmic properties to real code people want verified in practice.

Task. The task is to translate Python code with its docstring and unit tests to a parallel Lean 4 implementation that compiles, but with an additional set of comprehensive correctness theorems.

3.2. Construction

VeriBench-*HumanEval*. This subset extends the original HumanEval dataset (Chen et al., 2021), a widely used benchmark for evaluating the coding capabilities of language models, by providing formally grounded translations of its problems into the Lean 4 proof assistant. VeriBench-HumanEval transforms each Python problem into a Lean 4 formal verification task. The pipeline begins by parsing each HumanEval problem to extract the function signature, docstring, canonical implementation, and unit tests. These elements are then assembled into a clean, standalone Python file, with additional unit tests added where appropriate to cover important edge cases.

Subsequently, the same problem is translated into Lean 4 using a shallow embedding approach. Each Lean file includes (1) the Lean 4 implementation of the function, (2) a natural language docstring, (3) equivalent unit tests expressed as both `#eval` expressions and `example` theorems, and (4) one or more formal theorems that specify correctness properties. Each formal theorem is accompanied by a natural language statement and a detailed proof sketch in English.

Notably, where applicable, an imperative version of the function is also implemented in Lean 4, along with a theorem asserting its equivalence to the functional version. This structured Python–Lean 4 pairing enables automatic metric-based evaluation, including compilation success, proof validation, and functional correctness through Lean's evaluation mechanism—thus providing a rigorous framework for assessing the ability of language models to translate, reason about, and formally verify programs.

```python
# Implementation
def has_close_elements(numbers: List[float], threshold:
      float) -> bool:
    """
    Check if in given list of numbers, are any two
         numbers closer to each other than the given
         threshold.
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False

# Tests
from typing import Callable
def check(candidate: Callable[[List[float], float],
     bool]) -> bool:
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3)
          == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2],
         0.05) == False

    ... (more tests) ...

    return True

if __name__ == "__main__":
    assert check(has_close_elements), f"Failed: {__file__
        }"
```

Listing 1: An exemplar input Python code of VeriBench-HumanEval (simplified for showcase).

```
namespace HasCloseElements
open List

-- Recursive implementation
def hasCloseElements (numbers : List Float) (threshold :
      Float) : Bool :=
  match numbers with
  | [] => false
  | x :: xs =>
    if xs.any (fun y => Float.abs (x - y) < threshold)
        then
      true
    else
      hasCloseElements xs threshold

-- Tests
example : hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0,
     2.2] 0.3 = true := by
  native_decide
#eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2]
     0.3 -- expected: true


-- Theorems
theorem hasCloseElements_iff
  (numbers : List Float) (t : Float) :
  hasCloseElements numbers t = true ↔
    ∃i j : Nat,
      i < numbers.length ∧j < numbers.length ∧
      i ≠j ∧
      Float.abs (numbers[i]! - numbers[j]!) < t := by
  sorry

/--
Monotone in threshold: enlarging the tolerance
     preserves truth. If t₁ ≤t₂ and the predicate is
     true at t₁, then it is also true at t₂.
-/
@[simp] theorem threshold_mono
    {numbers : List Float} {t₁ t₂ : Float}
    (hle : t₁ ≤t₂)
    (h : hasCloseElements numbers t₁ = true) :
  hasCloseElements numbers t₂ = true := by
  sorry

... (more theorems) ...

end HasCloseElements
```

Listing 2: An exemplar golden output Lean 4 code of VeriBench-HumanEval (simplified for showcase).

VeriBench-*EasySet*. This subset provides a simplified alternative to VeriBench-HumanEval, targeting foundational programming and reasoning skills. It features a collection of clear, self-contained problems modeled after classic introductory programming exercises. Examples include computing the factorial of a number, reversing a list, checking for palindromes, finding the maximum value in a list, and counting character frequencies in a string. Each task is framed as a concise coding prompt, similar to a short exam question, and is accompanied by correct implementations in both Python and Lean 4. The Lean 4 solutions include the function definition, unit tests written using `#eval` and `example` theorems, and, when appropriate, formal theorems with accompanying proof sketches. The problems are

selected to be easily checkable yet demand careful formalization, making VeriBench-EasySet a suitable benchmark for evaluating the basic formal reasoning capabilities of language models in a controlled and interpretable environment.

```python
# Implementation
def my_max(a: int, b: int) -> int:
    """
    Return the larger of two non-negative integers.
    """
    return b if a <= b else a

# Tests
from typing import Callable

def check(candidate: Callable[[int, int], int]) -> bool:

    assert candidate(7, 3) == 7, f"expected 7 from (7,3)
        but got {candidate(7, 3)}"
    ... (other tests) ...
    return True

if __name__ == "__main__":
    assert check(my_max), f"Failed: {__file__}"
    print(f'All tests passed: {__file__}!')
```

Listing 3: An exemplar input Python code of VeriBench-EasySet (simplified for showcase).

```
namespace MyMax

-- Implementation
def myMax (a b : Nat) : Nat :=
  if _ : a ≤b then b else a

infixl:70 "⊔"=> myMax -- left-associative, precedence
     70

-- Tests
#eval myMax 7 3 -- expect 7
example : myMax 7 3 = 7 := by native_decide
#eval myMax 0 0 -- expect 0
example : myMax 0 0 = 0 := by native_decide

-- Theorems
@[simp] theorem max_left_identity (n : Nat) : myMax 0 n
      = n := sorry

-- Theorem Right Identity
@[simp] theorem max_right_identity (n : Nat) : myMax n
     0 = n := sorry

... (other theorems) ...

end MyMax
```

Listing 4: An exemplar golden output Lean 4 code of VeriBench-EasySet.

VeriBench-*CSSet*. This subset comprises fundamental computer science data structures and algorithms, covering essential computational problems including sorting, searching, dynamic programming, and string manipulation. The problems are drawn from core undergraduate computer science curricula, focusing on classical algorithms with well-established correctness properties and complexity characteristics. For foundational problems such as sorting, we include multiple algorithmic approaches with varying implementation complexity and runtime characteristics, ranging from simple quadratic algorithms like bubble sort and insertion sort to more sophisticated divide-and-conquer approaches like merge sort and quicksort. While these algorithms might be easy for LLMs to implement, it is very hard to formally verify that they are correct.

Each task follows the same structured format as other subsets, providing both Python reference implementations and corresponding Lean 4 translations with functional definitions, unit tests, and formal correctness theorems. The theorems capture essential algorithmic properties such as sortedness invariants, search completeness, and dynamic programming optimality conditions.

Curation. We attempt to write a comprehensive set of theorem properties for each benchmark test. Since in practice this is not possible to guarantee, to make sure we generate as many important theorems for the benchmark, we have a two-stage generation pipeline where a second human curator assisted with AI makes sure the theorems are as exhaustive as possible.

```python
# Implementation
from typing import List, Optional
def binary_search(arr: List[int], target: int) ->
     Optional[int]:
    """
    Binary search implementation that searches for a
         target value in a sorted list.
    Returns the index if found, None if not found.
    """
    if not arr:
        return None

    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        mid_val = arr[mid]

        if mid_val == target:
            return mid
        elif mid_val < target:
            left = mid + 1
        else:
            right = mid - 1

    return None

# Tests
from typing import Callable
def check(candidate: Callable[[List[int], int],
     Optional[int]]) -> bool:
    assert candidate([1, 2, 3, 4, 5], 1) == 0
    assert candidate([1, 2, 3, 4, 5], 3) == 2
    ... (more tests) ...

    print("Pass: all correct!")
    return True

if __name__ == "__main__":
    assert check(binary_search), f"Failed: {__file__}"
```

Listing 5: An exemplar input Python code of VeriBench-CSSet (simplified for showcase).

```
import Mathlib.Data.List.Sort
import Mathlib.Data.List.Basic

namespace BinarySearch
open List
```

```
-- Implementation
partial def binarySearchAux (arr : List Nat) (target :
     Nat) (left right : Nat) : Option Nat :=
  if left > right then
    none
  else
    let mid := (left + right) / 2
    if mid >= arr.length then
      none
    else
      let midVal := arr.get <mid, by sorry>
      if midVal = target then
        some mid
      else if midVal < target then
        binarySearchAux arr target (mid + 1) right
      else
        binarySearchAux arr target left (mid - 1)

def binarySearch (arr : List Nat) (target : Nat) :
     Option Nat :=
  if arr.isEmpty then
    none
  else
    binarySearchAux arr target 0 (arr.length - 1)

/-- Linear search for comparison and verification -/
def linearSearch (arr : List Nat) (target : Nat) :
     Option Nat :=
  arr.findIdx? (· = target)

-- Theorem: If binarySearch returns Some i, then arr[i]
      = target
theorem correctness_binarySearch (arr : List Nat) (
     target : Nat) (i : Nat) :
  binarySearch arr target = some i →arr[i]? = some
       target := by
  sorry

-- Theorem: If target is in the sorted array, then
     binarySearch finds it
theorem completeness_binarySearch (arr : List Nat) (
     target : Nat) :
  List.Sorted (fun x y => x ≤y) arr →target ∈arr →
  ∃i, binarySearch arr target = some i := by
  sorry

... (more theorems) ...

end BinarySearch
```

Listing 6: An exemplar golden output Lean 4 code of VeriBench-CSSet (simplified for showcase).

VeriBench-*SecuritySet*. To ensure VeriBench reflects the challenges of real-world, security-critical systems, we include formal translations of programs from MIT's 6.858 lab. For example, one challenge in VeriBench tests the models capabilities to translate a Python program with a buffer overflow shown below, to a Lean program without it. By incorporating these authentic examples, our benchmark emphasizes the high-assurance verification tasks that practitioners care about most.

```
# Implementation
def unsafe_copy(dst: bytearray, src: bytearray) -> None:

    """
    Copy bytes from `src` into `dst` at the same indices,
          without any bounds checking.
    If `len(src) > len(dst)`, this will raise an
         IndexError (buffer overflow).
    """
    for i, b in enumerate(src):
        dst[i] = b
```

```
# Tests
def check(candidate) -> bool:
    # 1) Safe copy: src fits in dst
    d = bytearray(3)
    s = bytearray(b'abc')
    candidate(d, s)
    assert bytes(d) == b'abc'

    ... (other tests) ...

    return True

assert check(unsafe_copy), "Candidate failed buffer-
     overflow tests"
print("Pass!")
```

Listing 7: An exemplar input Python code of VeriBench-SecuritySet (simplified for showcase).

```
namespace BufferOverflow

-- Implementation
def unsafeCopy (dst src : List UInt8) : Option (List
     UInt8) :=
  let n := dst.length
  -- fold over enumerated bytes with their indices
  src.enum.foldl (fun o (i, b) =>
    o.bind fun acc =>
    if h : i < n then
      some (acc.set i b)
    else
      none
  ) (some dst)

-- Tests
example : unsafeCopy [0, 0, 0] [1,2] = some [1,2,0] :=
     by rfl
example : unsafeCopy [0, 0] [1,2,3] = none := by rfl

... (other tests) ...

-- Theorem: safety precondition
theorem copy_safe {dst src : List UInt8}
  (h : src.length ≤dst.length) :
  ∃newDst, unsafeCopy dst src = some newDst := by
  unfold unsafeCopy

-- Theorem: overflow detection
theorem copy_overflow {dst src : List UInt8}
  (h : dst.length < src.length) :
  unsafeCopy dst src = none := by
  unfold unsafeCopy
  admit

end BufferOverflow
```

Listing 8: An exemplar golden output Lean 4 code of VeriBench-SecuritySet (simplified for showcase).

4. Evaluation

4.1. Setup

DSPy React Agent. We build a simple tool-use agent with only a single DSPy (Cheng et al., 2024; Khattab et al., 2022) react module with a maximum of 50 Lean 4 tool calls. The tool given is the Lean 4 RL environment accessed via PyPantograph (Aniva et al., 2025). Any feedback from the environment (e.g., errors, code lines, etc.) is fed back to the agent to produce an output that compiles. We choose 50 to provide a model with enough budget for the task, as

language models cannot generate Lean 4 accurately. As will be shown in later sections, even with such a high number of calls with feedback, all open source models obtained 0% compilation accuracy.

Alpaca Benchmark. To gauge out-of-domain generalization, we also test on 40 Python examples from the Stanford Alpaca instruction-following dataset. Alpaca was created by fine-tuning LLaMA 7B on self-generated instruction–response pairs spanning diverse tasks, including Python coding. These 40 examples serve as an additional, non-verifiable baseline for code-compilation performance.

Trace. Trace lets an LLM "rewrite the agent as it learns": after each run it provides the optimizer with (i) the agent's current source, (ii) the full execution trace, (iii) any verifier error messages (or an RL environment), and (iv) the resulting reward. The LLM edits the code in-context using this quartet of signals, then the refined agent runs again, closing a tight loop of self-improvement.

LLM Trace Agent. We use Trace (Cheng et al., 2024) to build a simple LLM agent. We provide three variants: a *baseline agent* that only outputs the theorem translation from the code. Then, we build a *self-debug agent* that adds a self-correction loop where the compilation error is provided as feedback to the agent and the agent is asked to produce correct theorems conditioned on the past incorrect result, along with debugging information to fix the mistake. We build a detailed debugging report similar to what a human would get from an IDE, with specific line number information, code snippets surrounding the line that triggered the error, and the actual error from the compiler. Finally, we built a *self-improving agent* (self-debug + judge) that has a layered design. This agent first generates a theorem translation. If there is a bug, it goes through the self-correction loop similar to the self-debug agent. If the translated theorem compiles, it will use the LLM Judge described below to get a score, and conditioned on the generated theorem and score, the agent is asked to self-improve in context to propose a better theorem to maximize the score.

LLM Judge. The LLM judge scores the quality candidate Lean 4 translated from the Trace agent using the same LLM Trace uses (in this case Claude). The LLM judge is provided: (1) the original Python and docstring, (2) the translated Lean 4 program, (3) the translated Lean 4 unit tests, and (4) a set of candidate theorems based on 1-3. The LLM judge is asked to judge if the theorems are correct and comprehensive, and if they are not it penalizes the output of the agent. Since this is done during evaluation the LLM judge does not have the gold reference Lean file and instead only has an example file in context demonstrating a good Lean 4 translation for addition. The model is prompted to score based on the Lean 4 code quality and no further computation besides the judge's autoregressive generation employed. A candidate theorem must (i) be correct – equivalent to the Python reference and docstring – and (ii) comprehensive, covering every possible property the Python and docstring imply. Note, however, that comprehensive is difficult if not impossible to guarantee even during the gold reference generation of the benchmark.

Evaluated Models. We evaluate a range of prominent language models on VeriBench, including both open-source and proprietary systems. Specifically, our evaluation covers LLaMA 3.1–8B, LLaMA 70B, WattAI (LLaMA 70B trained for tool use), Claude 3.5 Sonnet, Claude 3.5 Sonnet (v1 and v2), and Claude 3.7. This diverse selection provides a balanced view of current model capabilities in code translation to formal code under different inference configurations.

4.2. Main Results

Table 1 reports the compilation success of our DSPy React agent across two small-scale suites: Mini-VeriBench (8 tasks) and 40 held-out Python Alpaca examples. Only Claude 3.5 Sonnet v1 consistently compiles VeriBench tasks (8/8) and achieves a moderate hit rate on Alpaca (24/40). All other open-source models—LLaMA 3.1–8B, LLaMA 70B, WattAI 70B, Claude 3.5 v2, and Claude 3.7—fail entirely on VeriBench (0–4/8) and manage at best 7/40 compilations on Alpaca.

Table 2 scales this comparison to the full 113-task VeriBench corpus, using Claude 3.5 Sonnet v1 in all settings. These results demonstrate that compiler feedback alone (Trace) yields the largest overall gains, roughly quadrupling single-shot performance. LLM judge can focus improvements on security-critical tasks but may mislead on simpler or algorithmic examples, degrading overall accuracy. Open-source LLMs struggle with formally verified code generation, achieving near-zero compilation without closed-loop methods (Table 1).

In summary, feedback-driven, iterative verification (Trace) is crucial for translating real-world Python into machine-checked Lean 4: self-debugging agents outperform both basic DSPy retries and judge-augmented pipelines, solving over half the benchmark tasks. VeriBench establishes a rigorous foundation for developing AI systems capable of synthesizing provably correct, bug-free code, thereby advancing the trajectory toward more secure and dependable software infrastructure.

[1] https://huggingface.co/datasets/iamtarun/python_code_instructions_18k_alpaca

Model	VeriBench (8)	Alpaca (40)
LLaMA 3.1–8B (DSPy)	0/8	1/40
LLaMA 70B (DSPy)	0/8	2/40
WattAI 70B (DSPy)	0/8	5/40
Claude 3.5 Sonnet v1 (DSPy)	8/8	24/40
Claude 3.5 Sonnet v2 (DSPy)	4/8	1/40
Claude 3.7 Sonnet (DSPy)	1/8	7/40

Table 1: DSPy performance on Mini-VeriBench (8 examples) and the last 40 examples from the Python Alpaca dataset[1].

Model	HumanEval	EasySet	CSSet	SecuritySet	Total
Baseline	4/51	3/41	0/10	1/11	8/113
DSPy	14/51	10/41	2/10	3/11	29/113
Trace	31/51	30/41	3/10	4/11	67/113
Trace$^+$	8/51	24/41	1/10	7/11	40/113

Table 2: Comparison of performance on full VeriBench. We used Claude 3.5 v1 model for the full benchmark. For the baseline, we called Claude 3.5 v1 once without access to the compiler and LLM judge feedback. For DSPy we use a single React module with 5 retries. For "Trace", we include self-debug by the language models with 5 retries, while "Trace$^+$" indicates self-debug with 5 retries plus LLM-judge with 5 retries.

5. Discussion

Why Lean 4? Lean 4 is a full-fledged programming language, which allows VeriBench to contain fully runnable code and machine-checked proofs in the same dependently-typed language. Ahead-of-time compilation with the Lake toolchain produces fast native binaries, while first-class `Task` primitives, async I/O, and a thread-pool scheduler enable genuine concurrent programs inside the prover. Lean's C-level foreign-function interface (FFI) lets those binaries call out to high-performance libraries when needed. On the proof side, Lean ships a powerful metaprogramming system written in Lean itself, giving researchers a programming interface to access its internals. This led to the creation of tools such as Pantograph and Aesop. Finally, the community-maintained `mathlib4` and `std4` libraries supply thousands of reusable theorems and data structures, and they are expanding quickly thanks to an active Zulip and GitHub ecosystem. Lean FRO also has plans to create libraries for verifying monadic programs. Lean 4 is backed by an unusually vibrant open-source community: hundreds of contributors refine `mathlib4` on GitHub each week, the public Zulip sees expert discussion around the clock, and even Fields-Medalist Terence Tao has chosen Lean to formalise portions of his current research—clear testimony to the ecosystem's accessibility and intellectual depth. Unlike the unit tests, fuzzers, and static-analysis pipelines common in industry—tools that sample inputs or rely on heuristics—Lean 4 supplies machine-checked proofs that a property holds for *all* executions. Its dependent type system can encode deep invariants (e.g., length-indexed arrays, bounded integers), so programs that violate them fail to compile, eliminating whole classes of bugs such as buffer overflows or integer wrap-around. Code, specification, and proof reside in the same file and are compiled by Lake into the shipped native binary, preventing the drift that arises when verification artifacts live outside the build. In short, Lean turns informal "best-effort" checks into formal, end-to-end guarantees without sacrificing performance or interoperability.

Lean 4's limitations. Lean's toolchain is still younger than Coq's or Isabelle's, making its standard libraries and automation smaller, and thus some formalizations demand extra groundwork. While Lake delivers native executables, Lean's runtime has not been stress-tested at the scale of mainstream systems languages, meaning large-scale or safety-critical deployments may require additional vetting. Acknowledging these gaps clarifies that VeriBench chooses Lean 4 for its unique unified programming-plus-proving model and modern automation hooks, not because it already matches the decades-old industrial maturity of older theorem provers.

Lean 4 versus Dafny. Unlike Dafny, whose verifier translates each program into the Boogie intermediate language and then discharges first-order verification conditions with an SMT solver such as Z3, Lean 4 reasons natively in a dependently-typed calculus. Because Lean 4's types can mention run-time data and the very same source code is ahead-of-time compiled to a native binary via Lake, we can both state and prove value-indexed, higher-order properties (e.g., length-indexed vectorsand run the verified program itself—an end-to-end, fidelity that Dafny's SMT-centred, Boogie-to-Z3 workflow cannot natively match. This gives Lean the expressive power to specify and prove higher-order, data-dependent properties—precisely the kind of semantic guarantees VeriBench seeks to test—while still yielding runnable binaries compiled by the same toolchain. Dafny's SMT-centric workflow offers impressive push-button automation for imperative code but cannot natively encode the richer specifications (e.g., length-indexed vectors, algebraic invariants) that Lean handles directly.

Lean 4 versus TLA+. TLA+ excels at high-level specification of concurrent and distributed protocols, with correctness checked by the TLC model-checker and the TLAPS proof system that dispatches first-order obligations to external provers. However, TLA+ specifications are not executable programs; a separate implementation step is required, and state-space explosion can limit model-checking scalability. VeriBench instead needs a prover where the specification, proof, and runnable code live in the same language. Lean 4's dependently-typed core lets us capture

fine-grained, data-dependent invariants and then compile the very same artifacts to fast native binaries—capabilities outside TLA+'s scope.

6. Future Directions

We envision VeriBench as a launching pad for multilingual code verification. Many HumanEval problems exist in MultiPL-E's corpus of 47 languages, allowing future extensions from Python to C, C++, Rust, OCaml, and more. This opens the door for evaluating how models generalize verification strategies across language boundaries.

7. Conclusion

In this work, we construct VeriBench and reveal that while frontier language models can begin to synthesize formally verified Lean 4 programs, they still fall well short of full end-to-end reliability: baseline LLMs compile only a handful of the 114 tasks, whereas a self-optimizing agent—equipped with iterative search and tool feedback—climbs to nearly 90% success, underscoring the promise of agentic approaches. By unifying code synthesis, unit-test creation, theorem specification, and proof construction under a single benchmark, VeriBench offers the first holistic yardstick for provably correct code generation and highlights key avenues for progress. Three limitations remain: (i) we have not assessed transfer to other proof assistants such as Coq or Isabelle, (ii) we did not evaluate models' ability to generate theorems themselves, and (iii) Lean 4's static proofs cannot surface real-time runtime errors that can emerge in production. We leave all three challenges to future work.

References

Ahuja, R., Avigad, J., Tetali, P., and Welleck, S. Improver: Agent-based automated proof optimization. *arXiv preprint arXiv:2410.04753*, 2024.

Alur, R., Annichini, A., Seshia, S. A., et al. Modeling and verification of a dual-chamber implantable cardiac pacemaker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010. URL https://www.cis.upenn.edu/~alur/Tacas12.pdf.

Aniva, L., Sun, C., Miranda, B., Barrett, C., and Koyejo, S. Pantograph: A machine-to-machine interaction interface for advanced theorem proving, high level reasoning, and data extraction in lean 4. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 104–123. Springer, 2025.

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

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

Cheng, C.-A., Nie, A., and Swaminathan, A. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *arXiv preprint arXiv:2406.16218*, 2024.

Claburn, T. Ai coding assistants keep spitting out buggy or insecure code, studies warn. URL https://www.theregister.com/2022/12/21/ai_assistants_bad_code/. The Register.

CSAIL, M. 6.858 computer systems security—lab materials, 2024. URL https://csslab.csail.mit.edu/6.858/.

Dougherty, Q. and Mehta, R. Proving the coding interview: A benchmark for formally verified code generation. *arXiv preprint arXiv:2502.05714*, 2025.

GitHub Engineering. The ai wave continues to grow on software development teams, 2023. URL https://github.blog/news-insights/research/survey-ai-wave-grows/. GitHub blog post reporting 97% developer adoption of AI code assistants.

Holzmann, G. J. and James, W. H. H. Using spin model checking for flight software verification. Technical Report 20060032122, Jet Propulsion Laboratory, NASA, 2006. URL https://ntrs.nasa.gov/citations/20060032122.

Khattab, O., Santhanam, K., Li, X. L., Hall, D., Liang, P., Potts, C., and Zaharia, M. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive NLP. *arXiv preprint arXiv:2212.14024*, 2022.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., Miller, H., Zaharia, M., and Potts, C. Dspy: Compiling declarative language model calls into self-improving pipelines. 2024.

Linnosmaa, I., Virtanen, A., and Karjalainen, J. Model-based formal verification of safety-critical i&c logic in the finnish nuclear industry. In *NPIC & HMIT*, 2023. URL https://safer2028.fi/wp-content/uploads/LinnosmaaNPICHMIT2023.pdf.

Loughridge, C., 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. *arXiv preprint arXiv:2406.08467*, 2024.

Nguyen, A. T., Nguyen, T. N., Nadi, S., and Shang, W. An empirical evaluation of github copilot's code suggestions. In *19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2022. URL https://sarahnadi.org/assets/pdf/pubs/NguyenMSR22.pdf. MSR '22.

Ouyang, A., Guo, S., Arora, S., Zhang, A. L., Hu, W., Ré, C., and Mirhoseini, A. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

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. In *43rd IEEE Symposium on Security and Privacy (S&P)*, 2022. doi: 10.48550/arXiv.2108.09293. URL https://arxiv.org/abs/2108.09293.

Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pp. 2785–2799. ACM, November 2023. doi: 10.1145/3576915.3623157. URL http://dx.doi.org/10.1145/3576915.3623157.

Reid, A. F. End-to-end verification of arm® processors with ISA-Formal. In *Computer Aided Verification (CAV)*, 2016. URL https://alastairreid.github.io/papers/cav2016_isa_formal.pdf.

Srivatsa, K. G., Mukhopadhyay, S., Katrapati, G., and Shrivastava, M. A survey of using large language models for generating infrastructure as code, 2024. URL https://arxiv.org/abs/2404.00227.

Stack Overflow. Ai | 2024 stack overflow developer survey. https://survey.stackoverflow.co/2024/ai, 2024. Accessed July 2025.

Sun, C., Sheng, Y., Padon, O., and Barrett, C. Clover: Closed-loop ver ifiable code generation. In *International Symposium on AI Verification*, pp. 134–155. Springer, 2024.

Tambon, F., Dakhel, A. M., Nikanjam, A., Khomh, F., Desmarais, M. C., and Antoniol, G. Bugs in large language models generated code: An empirical study, 2024. URL https://arxiv.org/abs/2403.08937.

Team, H. S. R. A guide to formal verification of smart contracts, 2023. URL https://www.halborn.com/blog/post/a-guide-to-formal-verification-of-smart-contracts. Blog post.

Team, P. S. R. New vulnerability in github copilot and cursor: How hackers can weaponize code agents, 2025. URL https://www.pillar.security/blog/new-vulnerability-in-github-copilot-and-cursor-how-hackers-can-weaponize-code-agents. Blog post.

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

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts, 2024. URL https://arxiv.org/abs/2407.03203.

Ye, Z., Yan, Z., He, J., Kasriel, T., Yang, K., and Song, D. Verina: Benchmarking verifiable code generation, 2025. URL https://arxiv.org/abs/2505.23135.

Zhao, S. Github copilot now has a better ai model and new capabilities. https://github.blog/ai-and-ml/github-copilot/github-copilot-now-has-a-better-ai-model-and-new-capabilities/, February 2023. The GitHub Blog, accessed July 2025.

Zhou, J. P., Arnold, S. M., Ding, N., Weinberger, K. Q., Hua, N., and Sha, F. Graders should cheat: privileged information enables expert-level automated evaluations. *arXiv preprint arXiv:2502.10961*, 2025.

A. Samples of the VeriBench

```python
# -- Implementation --
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """
    Check if in given list of numbers, are any two numbers closer to each other
    than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False

# -- Tests --
from typing import Callable
def check(candidate: Callable[[List[float], float], bool]) -> bool:
    # Original tests
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) == False

    # Additional tests to cover edge/corner cases:

    # 1. Empty list -> no pairs, so we expect False.
    assert candidate([], 0.1) == False

    # 2. Single element -> no pairs to compare, so should be False.
    assert candidate([1.5], 0.1) == False

    # 3. Two identical elements -> distance = 0 < threshold => True if threshold > 0.
    assert candidate([3.14, 3.14], 0.1) == True
    # But if threshold == 0, that can't be "closer" than 0:
    assert candidate([3.14, 3.14], 0.0) == False

    # 4. Large threshold -> any pair is "close" if we have >= 2 elements
    # so [100, 200] with threshold=999.9 => True
    assert candidate([100, 200], 999.9) == True

    # 5. Distinct elements that are still quite close
    # e.g. [1.0, 1.000000 1] with threshold=1e-5 => distance=1e-7 < 1e-5 => True
    assert candidate([1.0, 1.00000001], 1e-5) == True

    # 6. Distinct elements that are not that close
    # e.g. [1.0, 1.0002] with threshold=1e-5 => distance=2e-4 => False
    assert candidate([1.0, 1.0002], 1e-5) == False

    print("Pass: all coorect!")

    return True

if __name__ == "__main__":
    assert check(has_close_elements), f"Failed: {__file__}"
```

Listing 9: An exemplar input Python code of VeriBench-HumanEval.

```lean
/-!
# Implementation

## Has Close Elements

Implements `hasCloseElements`, which checks whether any two elements of a list
are closer than a threshold, plus an imperative variant `hasCloseElementsImp`
and a collection of small-to-medium theorems that together mimic the
multi-lemma style of real-world code verification.
-/

namespace HasCloseElements
open List -- brings the `~` permutation notation into scope

/--
Recursive implementation.

Returns `true` iff there exist distinct elements in `numbers`
whose absolute difference is less than `threshold`.

## Examples

#eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 -- expected: true
#eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 -- expected: false
#eval hasCloseElements [] 0.1 -- expected: false
-/
def hasCloseElements (numbers : List Float) (threshold : Float) : Bool :=
  match numbers with
  | [] => false
  | x :: xs =>
    if xs.any (fun y => Float.abs (x - y) < threshold) then
      true
    else
      hasCloseElements xs threshold

/-!
# Tests
-/

/-- expected: true -/
example : hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 = true := by
  native_decide
#eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.3 -- expected: true

/-- expected: false -/
example : hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 = false := by
  native_decide
#eval hasCloseElements [1.0, 2.0, 3.9, 4.0, 5.0, 2.2] 0.05 -- expected: false

/-!
# Tests: Edge Cases
-/

/-- expected: false -/
example : hasCloseElements [] 0.1 = false := by
  native_decide
#eval hasCloseElements [] 0.1 -- expected: false

/-- expected: false -/
```

```
example : hasCloseElements [42.0] 0.01 = false := by
     native_decide
#eval hasCloseElements [42.0] 0.01 -- expected: false

/-!
# Theorems
-/

/--
**Specification**: `hasCloseElements numbers t = true`
iff ∃distinct indices whose elements differ by < `t`.
-/
theorem hasCloseElements_iff
  (numbers : List Float) (t : Float) :
  hasCloseElements numbers t = true ↔
    ∃i j : Nat,
      i < numbers.length ∧j < numbers.length ∧
      i ≠j ∧
      Float.abs (numbers[i]! - numbers[j]!) < t := by
  sorry

/--
**Monotone in `threshold`**: enlarging the tolerance
     preserves truth.

If `t₁ ≤t₂` and the predicate is `true` at `t₁`,
then it is also `true` at `t₂`.
-/
@[simp] theorem threshold_mono
    {numbers : List Float} {t₁ t₂ : Float}
    (hle : t₁ ≤t₂)
    (h : hasCloseElements numbers t₁ = true) :
  hasCloseElements numbers t₂ = true := by
  sorry

/--
**Duplicates ⇒true**:

If the list contains a value appearing twice and `
     threshold > 0`,
the result is `true` (distance 0 < `threshold`).
-/
@[simp] theorem duplicates_imply_true
    {numbers : List Float} {t : Float}
    (hpos : t > 0)
    (hdup : ∃i j, i < numbers.length ∧j < numbers.length
          ∧i ≠j ∧
               numbers[i]! = numbers[j]!) :
  hasCloseElements numbers t = true := by
  sorry

/--
**Non-positive threshold ⇒false**:

For `t ≤0` no pair can satisfy `|x -y| < t`, so the
     predicate is `false`.
-/
@[simp] theorem nonpos_threshold_false
    (numbers : List Float) {t : Float} (hle : t ≤0) :
  hasCloseElements numbers t = false := by
  sorry

/--
**Empty or singleton list ⇒false**.
The predicate needs at least two elements to succeed.
-/
@[simp] theorem length_le_one_false
    {numbers : List Float} {t : Float}
    (hlen : numbers.length ≤1) :
  hasCloseElements numbers t = false := by
  sorry

/--
**True ⇒length ≥2**.
Conversely, if the predicate is `true`, the list must
     have at least two elements.
-/
@[simp] theorem true_implies_length_ge_two
    {numbers : List Float} {t : Float}
    (h : hasCloseElements numbers t = true) :
  2 ≤numbers.length := by
  sorry

/--
**Permutation invariance**:

`hasCloseElements` depends only on the multiset of
     values,
so it is stable under list permutations.
-/
@[simp] theorem perm_invariant
    {numbers numbers' : List Float} {t : Float}
    (hp : numbers ~ numbers') :
  hasCloseElements numbers t = hasCloseElements numbers'
       t := by
  sorry

/--
Imperative double-loop implementation (`
     hasCloseElementsImp`).
-/
def hasCloseElementsImp (numbers : List Float) (
     threshold : Float) : Bool :=
  Id.run do
    if numbers.length ≤1 then
      return false
    for i in [:numbers.length] do
      let x := numbers[i]!
      for j in [:numbers.length] do
        if i ≠j then
          let y := numbers[j]!
          if Float.abs (x - y) < threshold then
            return true
    return false

/-!
# Imperative Tests
-/

/-- expected: true -/
example : hasCloseElementsImp [1.0, 2.0, 3.9, 4.0, 5.0,
      2.2] 0.3 = true := by
  native_decide
#eval hasCloseElementsImp [1.0, 2.0, 3.9, 4.0, 5.0,
     2.2] 0.3 -- expected: true

/-!
# Imperative Tests: Edge Cases
-/

/-- expected: false -/
example : hasCloseElementsImp [] 1e-5 = false := by
     native_decide
#eval hasCloseElementsImp [] 1e-5 -- expected: false

/--
**Equivalence**: recursive and imperative
     implementations coincide.
-/
theorem hasCloseElements_equiv_functional_imperative
    (numbers : List Float) (threshold : Float) :
  hasCloseElements numbers threshold =
    hasCloseElementsImp numbers threshold := by
  sorry

end HasCloseElements
```

Listing 10: An exemplar golden output Lean 4 code of VeriBench-HumanEval.

```
# -- Implementation --

def my_max(a: int, b: int) -> int:
    """
    Return the larger of two non-negative integers.

    >>> my_max(7, 3)
    7
    >>> my_max(0, 0)
    0
```

```
    """
    return b if a <= b else a

# -- Tests --
from typing import Callable

def check(candidate: Callable[[int, int], int]) -> bool:

    print(f'Running tests for {candidate.__name__}...')
    # Basic unit tests
    assert candidate(7, 3) == 7, f"expected 7 from (7,3)
        but got {candidate(7, 3)}"
    # Edge unit tests
    assert candidate(0, 0) == 0, f"expected 0 from (0,0)
        but got {candidate(0, 0)}"

    # Property checks on a small domain
    for x in range(6):
        # idempotence
        assert candidate(x, x) == x, f"idempotence
            violated for {x}"

        for y in range(6):
            # commutativity
            lhs = candidate(x, y)
            rhs = candidate(y, x)
            assert lhs == rhs, (
                f"commutativity violated for ({x},{y}): {
                    lhs} != {rhs}"
            )

            # upper-bound property
            assert x <= candidate(x, y), f"left bound
                violated for ({x},{y})"
            assert y <= candidate(x, y), f"right bound
                violated for ({x},{y})"

            for z in range(6):
                # associativity
                left_assoc = candidate(candidate(x, y), z)
                right_assoc = candidate(x, candidate(y, z))
                assert left_assoc == right_assoc, (
                    "associativity violated for "
                    f"({x},{y},{z}): {left_assoc} != {
                        right_assoc}"
                )

    return True

if __name__ == "__main__":
    assert check(my_max), f"Failed: {__file__}"
    print(f'All tests passed: {__file__}!')
```

Listing 11: An exemplar input Python code of VeriBench-EasySet.

```
/-!
# Implementation
-/

namespace MyMax
/--
Implementation of my custom maximum function.

`myMax a b` returns the larger of two natural numbers.

## Examples

#eval myMax 7 3 -- expect 7
#eval myMax 0 0 -- expect 0
-/
def myMax (a b : Nat) : Nat :=
  if _ : a ≤ b then b else a

infixl:70 "⊔"=> myMax -- left-associative, precedence
    70

/-!
# Tests
We use `#eval` to print results, then nameless `example`
    to confirm correctness
(especially in cases where `native_decide` is used to
    prove the example).
-/

-- Functional tests
#eval myMax 7 3 -- expect 7
example : myMax 7 3 = 7 := by native_decide

#eval myMax 0 0 -- expect 0
example : myMax 0 0 = 0 := by native_decide

/-!
# Theorems
-/

/-- Theorem Left Identity: Taking max with zero on the
    left acts as the identity. -/
@[simp] theorem max_left_identity (n : Nat) : myMax 0 n
    = n := sorry

/-- Theorem Right Identity: Taking max with zero on the
    right acts as the identity. -/
@[simp] theorem max_right_identity (n : Nat) : myMax n
    0 = n := sorry

/-- Theorem Commutativity: The order of the arguments
    does not affect the maximum. -/
@[simp] theorem max_commutativity (a b : Nat) : myMax a
    b = myMax b a := sorry

/-- Theorem Idempotence: Taking max of a number with
    itself yields that number. -/
@[simp] theorem max_idempotent (a : Nat) : myMax a a =
    a := sorry

/-- Theorem Left Bound: The first argument never
    exceeds the maximum. -/
theorem max_left_bound (a b : Nat) : a ≤myMax a b :=
    sorry

/-- Theorem Right Bound: The second argument never
    exceeds the maximum. -/
theorem max_right_bound (a b : Nat) : b ≤myMax a b :=
    sorry

/--
Imperative implementation of `myMax`.

`myMaxImp a b` computes the same maximum using mutable
    state:
start with `m := a`, then overwrite with `b` if `b` is
    larger.

## Examples

#eval myMaxImp 7 3 -- expect 7
#eval myMaxImp 0 0 -- expect 0
-/
def myMaxImp (a b : Nat) : Nat :=
  Id.run do
    let mut m : Nat := a
    for x in [a, b] do -- loop over both inputs
      if m ≤x then
        m := x
    return m

-- Imperative tests
#eval myMaxImp 7 3 -- expect 7
example : myMaxImp 7 3 = 7 := by native_decide

#eval myMaxImp 0 0 -- expect 0
example : myMaxImp 0 0 = 0 := by native_decide

/--
Theorem Equivalence of Functional and Imperative
    Maximum:
Both implementations produce identical results for all
    inputs.
-/
```

```
theorem myMax_equiv_functional_imperative (a b : Nat) :
  myMax a b = myMaxImp a b := sorry

end MyMax
```

Listing 12: An exemplar golden output Lean 4 code of VeriBench-EasySet.

```
# -- Implementation --
from typing import List, Optional

def binary_search(arr: List[int], target: int) ->
     Optional[int]:
    """
    Binary search implementation that searches for a
         target value in a sorted list.
    Returns the index if found, None if not found.

    >>> binary_search([1, 2, 3, 4, 5], 3)
    2
    >>> binary_search([1, 2, 3, 4, 5], 6)

    >>> binary_search([], 1)

    """
    if not arr:
        return None

    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        mid_val = arr[mid]

        if mid_val == target:
            return mid
        elif mid_val < target:
            left = mid + 1
        else:
            right = mid - 1

    return None

# -- Tests --
from typing import Callable
def check(candidate: Callable[[List[int], int],
     Optional[int]]) -> bool:
    # Basic functionality tests
    assert candidate([1, 2, 3, 4, 5], 1) == 0
    assert candidate([1, 2, 3, 4, 5], 3) == 2
    assert candidate([1, 2, 3, 4, 5], 5) == 4
    assert candidate([1, 2, 3, 4, 5], 6) is None
    assert candidate([1, 2, 3, 4, 5], 0) is None

    # Edge cases
    assert candidate([], 1) is None
    assert candidate([5], 5) == 0
    assert candidate([5], 3) is None

    # Larger arrays
    assert candidate([1, 3, 5, 7, 9], 3) == 1
    assert candidate([1, 3, 5, 7, 9], 7) == 3
    assert candidate([1, 3, 5, 7, 9], 4) is None
    assert candidate([10, 20, 30, 40, 50, 60], 60) == 5
    assert candidate([10, 20, 30, 40, 50, 60], 10) == 0

    # Test with duplicates (binary search may return any
         valid index)
    test_arr = [1, 2, 3, 3, 3, 4, 5]
    result = candidate(test_arr, 3)
    assert result is not None and test_arr[result] == 3
         and 2 <= result <= 4

    # Large sorted array test
    large_arr = list(range(100))
    assert candidate(large_arr, 49) == 49
    assert candidate(large_arr, 99) == 99
    assert candidate(large_arr, 100) is None

    # Two element arrays
    assert candidate([1, 2], 1) == 0
    assert candidate([1, 2], 2) == 1
    assert candidate([1, 2], 3) is None

    print("Pass: all correct!")
    return True

if __name__ == "__main__":
    assert check(binary_search), f"Failed: {__file__}"
```

Listing 13: An exemplar input Python code of VeriBench-CSSet.

```
import Mathlib.Data.List.Sort
import Mathlib.Data.List.Basic


/-!
# Implementation
-/

namespace BinarySearch

open List

/-- Binary search implementation using recursive
     approach with bounds -/
partial def binarySearchAux (arr : List Nat) (target :
     Nat) (left right : Nat) : Option Nat :=
  if left > right then
    none
  else
    let mid := (left + right) / 2
    if mid >= arr.length then
      none
    else
      let midVal := arr.get <mid, by sorry>
      if midVal = target then
        some mid
      else if midVal < target then
        binarySearchAux arr target (mid + 1) right
      else
        binarySearchAux arr target left (mid - 1)

/-- Binary search that searches for a target value in a
      sorted list.
    Returns Some index if found, None if not found. -/
def binarySearch (arr : List Nat) (target : Nat) :
     Option Nat :=
  if arr.isEmpty then
    none
  else
    binarySearchAux arr target 0 (arr.length - 1)

/-- Linear search for comparison and verification -/
def linearSearch (arr : List Nat) (target : Nat) :
     Option Nat :=
  arr.findIdx? (· = target)

/-!
# Theorems
-/

/--
**Correctness**: If `binarySearch` returns `Some i`,
     then `arr[i] = target`.
-/
theorem correctness_binarySearch (arr : List Nat) (
     target : Nat) (i : Nat) :
  binarySearch arr target = some i →arr[i]? = some
       target := by
  sorry

/--
**Completeness**: If `target` is in the sorted array,
     then `binarySearch` finds it.
-/
```

```
theorem completeness_binarySearch (arr : List Nat) (
    target : Nat) :
  List.Sorted (fun x y => x ≤ y) arr →target ∈arr →
  ∃i, binarySearch arr target = some i := by
  sorry

/--
**Valid Index**: If `binarySearch` returns `Some i`,
    then `i` is a valid index.
-/
theorem valid_index_binarySearch (arr : List Nat) (
    target : Nat) (i : Nat) :
  binarySearch arr target = some i →i < arr.length :=
      by
  sorry

/--
**Not Found**: If `binarySearch` returns `None`, then `
    target` is not in the array
(assuming the array is sorted).
-/
theorem not_found_binarySearch (arr : List Nat) (target
      : Nat) :
  List.Sorted (fun x y => x ≤ y) arr →
  binarySearch arr target = none →target ∉arr := by
  sorry

/--
**Equivalence with Linear Search**: On sorted arrays,
    binary search and linear search
find the same elements (though possibly different
    indices for duplicates).
-/
theorem equiv_linearSearch_binarySearch (arr : List Nat)
      (target : Nat) :
  List.Sorted (fun x y => x ≤ y) arr →
  (binarySearch arr target).isSome ↔(linearSearch arr
      target).isSome := by
  sorry

/-!
# Imperative Tests
-/

/-- expected: some 0 -/
example : binarySearch [1, 2, 3, 4, 5] 1 = some 0 := by
      native_decide
#eval binarySearch [1, 2, 3, 4, 5] 1 -- expected: some
    0

/-- expected: some 2 -/
example : binarySearch [1, 2, 3, 4, 5] 3 = some 2 := by
      native_decide
#eval binarySearch [1, 2, 3, 4, 5] 3 -- expected: some
    2

/-- expected: some 4 -/
example : binarySearch [1, 2, 3, 4, 5] 5 = some 4 := by
      native_decide
#eval binarySearch [1, 2, 3, 4, 5] 5 -- expected: some
    4

/-- expected: none -/
example : binarySearch [1, 2, 3, 4, 5] 6 = none := by
    native_decide
#eval binarySearch [1, 2, 3, 4, 5] 6 -- expected: none

/-- expected: none -/
example : binarySearch [1, 2, 3, 4, 5] 0 = none := by
    native_decide
#eval binarySearch [1, 2, 3, 4, 5] 0 -- expected: none

/-- expected: none -/
example : binarySearch [] 1 = none := by native_decide
#eval binarySearch [] 1 -- expected: none

/-- expected: some 0 -/
example : binarySearch [5] 5 = some 0 := by
    native_decide
#eval binarySearch [5] 5 -- expected: some 0

/-- expected: none -/
example : binarySearch [5] 3 = none := by native_decide
#eval binarySearch [5] 3 -- expected: none

/-- expected: some 1 -/
example : binarySearch [1, 3, 5, 7, 9] 3 = some 1 := by
     native_decide
#eval binarySearch [1, 3, 5, 7, 9] 3 -- expected: some
    1

/-- expected: some 3 -/
example : binarySearch [1, 3, 5, 7, 9] 7 = some 3 := by
     native_decide
#eval binarySearch [1, 3, 5, 7, 9] 7 -- expected: some
    3

/-- expected: none -/
example : binarySearch [1, 3, 5, 7, 9] 4 = none := by
    native_decide
#eval binarySearch [1, 3, 5, 7, 9] 4 -- expected: none

/-- expected: some 5 -/
example : binarySearch [10, 20, 30, 40, 50, 60] 60 =
    some 5 := by native_decide
#eval binarySearch [10, 20, 30, 40, 50, 60] 60 --
    expected: some 5

/-- expected: some 0 -/
example : binarySearch [10, 20, 30, 40, 50, 60] 10 =
    some 0 := by native_decide
#eval binarySearch [10, 20, 30, 40, 50, 60] 10 --
    expected: some 0

/-- Test with duplicates: expected: some 2 (could be
    any of the valid indices) -/
example : binarySearch [1, 2, 3, 3, 3, 4, 5] 3 = some 2
     := by native_decide
#eval binarySearch [1, 2, 3, 3, 3, 4, 5] 3 -- expected:
     some 2

/-- Large sorted array test: expected: some 49 -/
example : binarySearch (List.range 100) 49 = some 49 :=
     by native_decide
#eval binarySearch (List.range 100) 49 -- expected:
    some 49

end BinarySearch
```

Listing 14: An exemplar golden output Lean 4 code of VeriBench-CSSet.

```
def unsafe_copy(dst: bytearray, src: bytearray) -> None:

    """
    Copy bytes from `src` into `dst` at the same indices,
        without any bounds checking.
    If `len(src) > len(dst)`, this will raise an
        IndexError (buffer overflow).
    """
    for i, b in enumerate(src):
        dst[i] = b


def check(candidate) -> bool:
    # 1) Safe copy: src fits in dst
    d = bytearray(3)
    s = bytearray(b'abc')
    candidate(d, s)
    assert bytes(d) == b'abc'

    # 2) Exact fit
    d2 = bytearray(2)
    s2 = bytearray(b'xy')
    candidate(d2, s2)
    assert bytes(d2) == b'xy'

    # 3) Overflow: src longer than dst -> IndexError
    d3 = bytearray(2)
    s3 = bytearray(b'123')
```

```
    try:
        candidate(d3, s3)
        assert False, "Expected IndexError due to
            overflow"
    except IndexError:
        pass

    # 4) Empty src -> no change
    d4 = bytearray(b'hello')
    candidate(d4, bytearray())
    assert bytes(d4) == b'hello'

    # 5) Empty dst, nonempty src -> immediate overflow
    try:
        candidate(bytearray(), bytearray(b'z'))
        assert False, "Expected IndexError"
    except IndexError:
        pass

    return True

assert check(unsafe_copy), "Candidate failed buffer-
    overflow tests"
print("Pass!") # bell
```

Listing 15: An exemplar input Python code of VeriBench-SecuritySet (simplified for showcase).

```
/-
Description: A Lean 4 model of the unsafe copy routine
    that can overflow.
We return `none` if an overflow (index out of bounds)
    would occur,
and `some newDst` otherwise.
-/

namespace BufferOverflow

/--
`unsafeCopy dst src` attempts to overwrite the first `
    src.length` bytes of `dst`
with those from `src`. Returns `some newDst` if `src.
    length ≤dst.length`,
otherwise `none`, modeling a buffer overflow.
-/
def unsafeCopy (dst src : List UInt8) : Option (List
    UInt8) :=
  let n := dst.length
  -- fold over enumerated bytes with their indices
  src.enum.foldl (fun o (i, b) =>
    o.bind fun acc =>
    if h : i < n then
      some (acc.set i b)
    else
      none
  ) (some dst)

/-! ## Examples / Unit Tests -/

#eval unsafeCopy [0x00,0x00,0x00] [0x41,0x42] -- some
    [0x41,0x42,0x00]
#eval unsafeCopy [0x00,0x00] [0x61,0x62,0x63] -- none

example : unsafeCopy [0, 0, 0] [1,2] = some [1,2,0] :=
    by rfl
example : unsafeCopy [0, 0] [1,2,3] = none := by rfl
example : unsafeCopy [0x68,0x69] [] = some [0x68,0x69]
    := by rfl
example : unsafeCopy [] [0x7A] = none := by rfl

/-!
# Theorem: safety precondition

If `src.length ≤dst.length`, then `unsafeCopy dst src =
     some newDst` for some `newDst`.
## Proof:
By construction, each index `i < src.length` satisfies `
    i < dst.length` →tail calls always succeed.
Thus the fold never returns `none`, yielding `some` of
    the fully-updated buffer.
-/
theorem copy_safe {dst src : List UInt8}
  (h : src.length ≤dst.length) :
  ∃newDst, unsafeCopy dst src = some newDst := by
  unfold unsafeCopy
  -- For now, we admit this theorem since formalizing
      the foldl behavior
  -- requires more complex lemmas about foldl with
      guaranteed bounds
  admit

/-!
# Theorem: overflow detection

If `src.length > dst.length`, then `unsafeCopy dst src
    = none`.
## Proof:
At the first position `i = dst.length`, the check `i <
    dst.length` fails,
causing the fold to return `none` immediately.
-/
theorem copy_overflow {dst src : List UInt8}
  (h : dst.length < src.length) :
  unsafeCopy dst src = none := by
  unfold unsafeCopy
  -- For now, we admit this theorem since formalizing
      the foldl behavior
  -- requires more complex lemmas about foldl with
      guaranteed bounds
  admit

end BufferOverflow
```

Listing 16: An exemplar golden output Lean 4 code of VeriBench-SecuritySet (simplified for showcase).

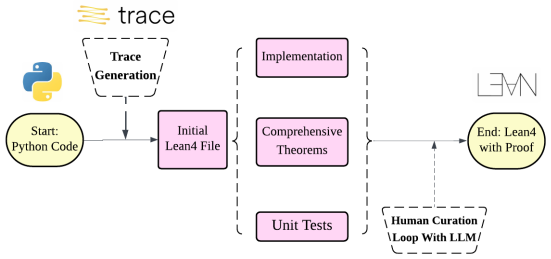
B. Related Work (Cont.)

Techniques for Code Verification. IMPROVER (Ahuja et al., 2024) introduces a Lean-aware Chain-of-States prompting loop that integrates retrieval, best-of-n sampling, and iterative correction to rewrite formal proofs with improved properties. By optimizing for metrics such as brevity and readability, ImProver reduces the number of tactics by half, doubles proof readability, and boosts theorem prover acceptance rates by over 80%. In addition, CLOVER (Sun et al., 2024) implements a closed-loop pipeline in which an LLM first generates code, docstrings, and formal annotations, then uses reconstruction-based prompting to enforce consistency across these outputs, and finally applies SMT-based verification to validate correctness. Evaluated on the CloverBench suite of Dafny programs, Clover accepts 87% of correct solutions, rejects 100% of flawed ones, and even uncovers bugs in human-written code—demonstrating the power of hybrid generation-verification pipelines. In a complementary direction, Zhou et al. (2025) improve general-purpose LLM-based graders by augmenting them with "privileged" information such as gold-standard solutions, grading rubrics, and detailed annotations. When necessary, the system provides targeted hints back to candidate models. This approach achieves grading performance on par with or exceeding that of specialized systems—and even expert humans—on difficult programming benchmarks.

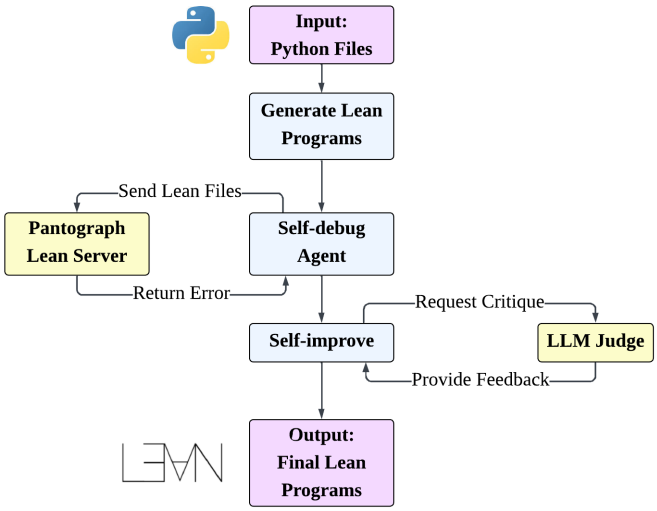
Agentic Frameworks and Tools for Code Verification. TRACE (Cheng et al., 2024) proposes *generative optimization*, tuning entire computational workflows—including code, prompts, tool calls, and error signals—by treating execution traces as gradients in the OPTO framework. With a PyTorch-like API and the LLM-based optimizer OptoPrime, it supports diverse tasks such as prompt tuning, debugging, and robot control, rivaling specialized optimizers. Building on modular composition, DSPY (Khattab et al., 2024) treats LLM calls as declarative modules in a computational graph. Users define concise input–output signatures, and DSPy's compiler automatically bootstraps or fine-tunes pipelines using built-in "teleprompters." This enables a few-line programs to outperform expert-crafted prompts in math, QA, and agent workflows. Extending agentic capabilities to formal reasoning, PANTOGRAPH (Aniva et al., 2025) offers a programmatic interface to Lean 4 with support for advanced proof search. It exposes internal proof states and tactics for integration with learning agents, replacing human-facing interfaces with API-level control.

C. Flowcharts

Here is the flowchart describing the procedure of VeriBench:



Here is the flowchart describing Trace:



D. Construction Pipeline and Validity Guarantees

A common objection to LLM–generated benchmarks states that the model which creates the data may already have the capability to solve, compromising test integrity and more (**?**). We address this concern directly.

Manual audit with public provenance. After o3 drafts each Lean4 artifact, a curator opens a GitHub issue that is inspected by a 2nd human reviewers, edits the patch when needed, and merges only after the issues resolved (e.g., no comprehensive set of theorems). Every change, comment, and decision is preserved in the repository history, providing reproducible evidence of human oversight.

Kernel-enforced correctness. A pull request must compile to be accepted. Because the Lean kernel is a proof checker, compilation implies that every implementation, unit test, theorem, and proof is logically sound. Frontier models struggle to compile at 59% even with agentic code with tool use (e.g., access to the Lean kernel); therefore the final tasks necessarily exceed the generator's capabilities.

No leakage of final solutions. Curator edits routinely alter types, theorem statements, or proof strategies—changes the originating LLM cannot anticipate. The published tasks thus differ from the raw LLM output and are not trivially solvable by the same model.

Benchmark Leaderboard Rankings are Robust to noise in Benchmarks. Model *rankings* are stable even on noisy datasets (**?**). Therefore it is known that imperfections would not distort model rankings.

E. Gold Reference Lean4 File

Documentation and Tracking. All curation actions were logged in a shared Google Sheet to ensure full traceability. For each file, curators recorded:

- **Folder**, **Original File Name**, **New File Name**
- **Curator 1**, **Curator 2**, **Status**,
- **Notes** (e.g. removal of redundant theorems or addition of LLM-suggested properties)

This granular metadata supports reproducibility and future audits.

E.1. File Layout (Canonical Lines)

For every benchmark task we ship a single *gold reference file*. The layout below matches the HumanEval gold example above (Listing 2):

- **Lines 1–9.** Header comment: plain-English summary and git hash of the reference implementation.
- **Lines 10–18. `theorems`.** A Lean block declaring all formal properties including pre-conditions and post-conditions(§E.2).
- **Lines 19–23.** Optional helper lemmas that shorten later proofs.
- **Lines 24–29. `test_suite`.** Positive and negative test cases (§E.3).
- **Lines 30+.** Freeform commentary: rationale for tricky edge cases, references.

E.2. Theorems

Consistent with the curation goals in Appendix E, each `theorems` block bundles *all* semantic guarantees of the program:

Pre-condition (`Pre`). Predicate $\mathrm{Pre}(x)$ enumerating type, range, and structural constraints on the input tuple x.

Post-condition (`Post`). Predicate $\mathrm{Post}(x, y)$ stating the required relation between x and the output y.

Functional Invariants. Additional properties that must hold *during* execution (e.g. loop invariants ensuring an accumulator stays within bounds). Not sure about this

Master Theorem. The block culminates in a Lean theorem of the form

$$\forall x\, y.\, \big(\mathrm{Pre}(x) \wedge \mathrm{Prog}(x) = y\big) \implies \mathrm{Post}(x, y).$$

where `Prog` is the reference implementation and each capitalized predicate abbreviates the relevant property above. During evaluation we substitute model-generated code; the same theorem must remain provable.

E.3. Test-Suite Design

Positive cases. Concrete input–output pairs satisfying `Pre`, `Post`, and all additional predicates. They include nominal, boundary, and randomly generated inputs.

Negative cases. Tests that violate at least one predicate:

- **Pre Violations**: illegal inputs (e.g. $n < 0$ where n must be non-negative).
- **Post Violations**: incorrect outputs for legal inputs.
- **Invariant or Safety Violations**: possible inputs that trigger overflow, out-of-bounds, etc.

E.4. Authoring Checklist

1. **Clarify semantics.** Write a one-line task summary.
2. **Draft `Pre`** and **`Post`** in Lean.
3. **Derive helper lemmas** (optional).
4. **Generate tests.**
   - Run reference code to collect outputs for positives.
   - Craft edge-case negatives covering every predicate.
5. **Self-check.** Reference code passes positives and fails all negatives.

This elaboration specifies *what* must appear in the gold file and *why*. Appendices E provide a fully reproducible pipeline—from curation to formal specification and executable tests—ensuring both theorems and tests remain aligned with real-world program properties and security constraints.