Leanstral

Abstract

In this work, we present Leanstral, a series of generalist code-agent models for Lean 4 with 119B total and only 6B active parameters. Rather than a specialized prover scaffold, Leanstral operates directly within the open-sourced Mistral Vibe code-agent harness and uses no test-time-scaling method beyond context compaction, with performance scaling smoothly as the per-problem token budget grows. Despite its size, Leanstral delivers results that rival far larger and proprietary systems: it **saturates miniF2F**, **solves 587/672 problems on PutnamBench**, and **reaches a new state-of-the-art 34% on FATE-X** and **43.2% on FLTEval**. Beyond competition mathematics, Leanstral resolves issues in real repositories spanning graduate-level mathematics, mathematical finance, and code verification. A fully automated pipeline built on Leanstral uncovered previously unknown bugs in open-source software. In this report, we detail our training recipe and how performance of Leanstral scales with test-time-compute. We open-source Leanstral 1.5 with an Apache-2.0 license.

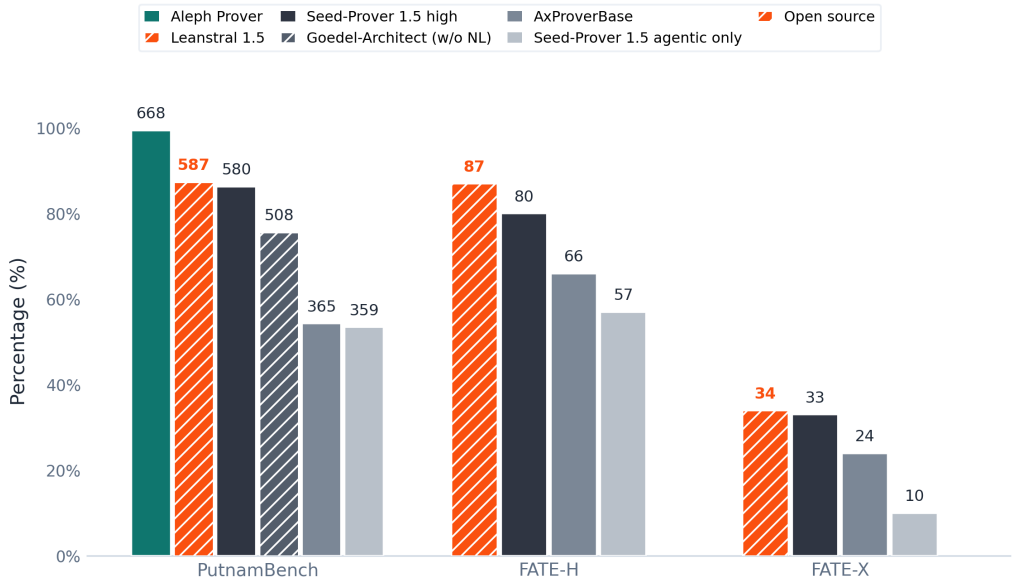

Figure 1: Leanstral 1.5 delivers SoTA open-source performance while costing over 10× less than the Seed-Prover 1.5 High setting, which uses 10 H20-days of compute per problem.

1 Introduction

Formal theorem proving offers a path to mathematical and software artifacts whose correctness can be checked by a trusted kernel. However, recent high-performing Lean systems often obtain their best results through specialized test-time scaling workflows: blueprint generation and refinement, recursive lemma decomposition, parallel prover orchestration, or other custom search procedures. These systems can be powerful, but their test-time scaling interface is not the interface that most Lean users work with day to day. Seed-Prover 1.5 High, for example, reports a 10 H20-days-per-problem workflow with conjecture proposing and lemma pools [Chen et al., 2025], while Goedel-Architect uses blueprint generation and parallel lemma proving to scale inference. In contrast, Leanstral is trained and evaluated as a general code agent whose test-time scaling comes from running longer inside the ordinary Mistral Vibe interaction loop. It interacts with Lean through Mistral Vibe [Mistral AI, 2025], the same code-agent interface used for ordinary software engineering: editing files, running shell commands, inspecting compiler feedback, using language-server tools, and compacting long contexts.

This framing is important for two reasons. First, it makes the model easy to use: users can ask Leanstral to work in a repository much like they would ask a general coding assistant such as Claude Code, while using an open model specialized for Lean proof engineering. Second, it exposes the model during training to the same long-horizon interaction pattern it must use at inference time. Lean-specific infrastructure is still necessary for reliable rewards and efficient feedback, but it supports a general code-agent loop rather than replacing the user-facing test-time workflow with a bespoke prover scaffold.

Our main contributions are:

1. We train Apache-2.0-licensed Lean code-agent models of 119B total parameters with only 6B active parameters that operate directly in the open-source Mistral Vibe harness, making test-time scaling accessible through an ordinary code-agent interface.
2. We build code-agent environments for Lean proof engineering, combining Lean compiler feedback, Lean LSP MCP, context compaction, and SafeVerify-style verification for theorem and repository-level tasks.
3. We show that a native code-agent approach is highly sample- and cost-efficient, solving 587/672 PutnamBench problems, reaching 34% on FATE-X, and outperforming much larger models on FLTEval.
4. Although Leanstral is primarily trained for mathematical proof engineering, we show that it also transfers to code verification, proving time-complexity properties of AVL trees and finding previously unknown bugs in translated Rust repositories.

2 Approach

2.1 Data

We collect Lean data for three stages of training: mid-training, supervised fine-tuning, and reinforcement learning. For mid-training, we start from the Mistral Small 4 [Mistral AI, 2026b] checkpoint and train on 6.5B Lean-specific tokens after deduplication, plus general code-agent data and other sources for instruction following and alignment. The Lean tokens include traces of generalist models attempting individual theorem-proving tasks as well as realistic proof-engineering issues in repositories.

For SFT, we use a Lean-code-agent-heavy mixture in which 50% of the trainable tokens come from Lean code-agent traces. The data is filtered for correctness, formatting, and style. We also filter behaviors that are especially harmful for long-horizon proof engineering, such as declaring success without compilation feedback, refusing difficult but feasible tasks, or hallucinating constraints that are not present in the repository.

For RL, we use a mixture of LeanGym PR data (50%), LeanGym single-theorem data (20%), and prove-or-disprove multiturn data (30%). The collection process of these data is introduced in the next paragraphs. We select RL samples that have low but non-zero solve rates under the SFT checkpoint,

estimated with 8 rollouts per sample. This removes examples that are already saturated and examples that provide little useful learning signal.

Single-Theorem Competition Mathematics Data We source a variety of self-contained competition mathematics theorem statements in Lean, and use our SFT checkpoint to filter out the easy fraction of the data. This is used with the LeanGym Single Theorem and Prove-or-Disprove Multiturn RL environments.

LeanGym PR Data In addition to competition data, we use real Lean repositories. The LeanGym pipeline extracts tasks from pull requests (PRs) of permissively licensed Lean 4 GitHub repositories:

1. Collect raw PRs from such repositories.
2. For each commit, build the repository in a container at the base commit.
3. Find altered theorems and definitions via metaprogramming and omit their proofs or bodies.
4. Apply the full PR patch and verify its correctness via SafeVerify. Discard the failed PRs.
5. Rewrite the task description so it contains the full context of the PR.

2.2 Environments

Leanstral is trained on two types of environments: Our multiturn environment allows the model to prove or refute given theorems and Lean feedback between attempts; Our LeanGym environment is a code-agent environment: the agent interacts with a repository, receives tool outputs, edits files, and decides how to proceed. The agent is given access to `bash` and `lean-lsp-mcp`. The LSP interface is important because repeatedly invoking `lake build` can be slow and coarse-grained. The language-server feedback gives the agent lower-latency access to goals, errors, type information, and local diagnostics.

2.2.1 Prove-or-Disprove Multiturn Environment

In our Multiturn environment, the model is given a sorried-out theorem statement and required imports, with instructions to either prove or refute it. Leanstral then attempts a proof of the theorem or of its negation. If the attempt is not successful (i.e., the proof does not compile, depends on additional axioms, or proves the wrong theorem), we provide Lean compiler feedback and prompt the model to try again. If this back-and-forth reaches a maximum number of turns or the maximum context length of the model without success, we declare the attempt a failure.

Partial reward is given for the model correctly formatting its response, i.e., having a thinking block and a formatted answer. The required answer format is a code block with similar setup as the original problem, and supplying a single Lean proof without lemmas. To be extra cautious, we also discourage `set_options` and verify there is no code after the proof using Lean's metaprogramming features. Full reward is given if, additionally, the proof compiles in our Lean Interact [Poiroux et al., 2025] based verifier. We use `#print axioms` to verify that the proof only uses the standard Lean axioms. Most notably, we disallow any use of `native_decide` because it is known to have consistency issues. The theorem statements are drawn from competition problems. They are mostly self-contained and require little context outside of what is given in the theorem statements, imports, and namespaces.

2.2.2 LeanGym Environment

We create the LeanGym environment for training Lean code agents. Compared to the Multiturn environment, it allows the agents to perform a wide range of Lean proof engineering tasks. Its design is informed by two distinguishing features of Lean code agent training compared to general code agents: they frequently need to build the Lean project which can be cpu-intensive, and instead of unit tests, we need to check proof compilation and exact type matching to verify the results.

At the abstract level, LeanGym takes as input a list of theorems and definitions that need completing, orchestrates an agent in a sandbox, and uses our modified SafeVerify [GasStationManager, 2024] to output a verdict based on the final state in the sandbox. Now, because of the high latency of building Lean projects, we perform the following optimizations: For each RL instance, we pre-fetch all its

dependencies, apply the startup patch, build the project to make sure the sandbox state is where we want the model to start with, and build a container for it. We use our own cache system to avoid overloading the shared one for Lean users.

In LeanGym, when trajectories become longer than the model's context window, we use context compaction to summarize prior work and let the model continue working. For Leanstral 1.5, we preserve the original task instructions in the first user message alongside the compacted history. This reduces objective drift: the model can recover the user's original goal even after many rounds of summarization.

LeanGym PR Environment For LeanGym PR data, Leanstral is run inside a clone of the target Lean repository set to the base commit of the PR. We apply a startup patch containing all PR changes, but we replace any modified proofs or definition bodies with `sorry`. The goal is to replace these `sorry`s with the correct definitions and proofs such that the target theorems are verified.

LeanGym Single-Theorem Environment For self-contained competition problems, we create a bare-bones Lean project containing the theorem statement, required imports, and necessary configuration files. The agent is tasked with filling in the proof, and can write additional lemmas and new Lean files should it choose to.

Modified SafeVerify Our modified version of SafeVerify (available at `https://github.com/mistralai/LeanstralSafeVerify`) provides strong assurance that a proof is correct, while giving the agent expressive freedom in how to structure the Lean code. The original version of SafeVerify supported adding additional lemmas, definitions, instances, and settings to prove a theorem. We add support for multi-file submissions and the ability to check proofs of theorem negations.

Also, to support the LeanGym PR data, we add the ability to complete a subset of declarations in a Lean project. One can specify the sorried theorems or definitions which the model is tasked with completing. We check that those are completed but other declarations remain unchanged. As many ongoing Lean projects contain sorries and axioms, we add support for theorems that are allowed to remain unsolved as well as custom axioms. This check prevents the model from using `sorry` directly, but allows the model to use the same sorried theorems used in the original ground truth proof. Similarly, we train Leanstral to fill in sorried definitions, but assert that the definition is completed with the same value as in the ground truth.

2.3 Training

After mid-training and SFT, we train Leanstral with reinforcement learning from verifiable Lean feedback. Our RL follows CISPO [MiniMax, 2025], using a truncated importance-weighted policy-gradient objective to reduce the effect of trainer-generator mismatch on long code-agent trajectories. For a group of trajectories o_i, model-generated tokens $a_{i,t}$, and normalized outcome advantages $\hat{A}_i$, we optimize

$$\mathcal{L}_{\mathrm{CISPO}}(\theta) = -\frac{1}{\sum_i |o_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \bar{\rho}_{i,t}\, \hat{A}_i \log \pi_\theta(a_{i,t} \mid s_{i,t}, T_i),$$

$$\bar{\rho}_{i,t} = \min\left(\frac{\pi_\theta(a_{i,t} \mid s_{i,t}, T_i)}{\pi_{\mathrm{sample}}(a_{i,t} \mid s_{i,t}, T_i)}, c \right),$$

where T_i denotes the interleaved tool calls and tool responses in the trajectory, and c is a finite truncation threshold. We stop gradients through $\bar{\rho}_{i,t}$. In our runs, the final verifier outcome is propagated to all model-generated tokens in the trajectory, while prompts and tool responses are used as context rather than training targets.

The reward is task-dependent. For the multiturn environment, full reward requires a verified proof or disproof with permitted axioms; partial reward is given for satisfying the required response format. For LeanGym, reward comes from SafeVerify after the model submits the final repository state. This combination trains the model not only to produce Lean proofs, but also to use the code-agent interface effectively: inspect feedback, edit files, build auxiliary lemmas, and persist over long proof-engineering trajectories.

3 Evaluation and results

We evaluate Leanstral on a series of benchmarks covering competition, undergraduate, graduate, and research mathematics, as well as FLTEval for proof engineering in real Lean repositories. For all benchmarks, we use Leanstral in Mistral Vibe and our modified SafeVerify for verification.

3.1 Formal Mathematics Benchmarks

3.1.1 miniF2F

MiniF2F [Zheng et al., 2022] is a standard formal benchmark composed of problems ranging from elementary to the level of the International Mathematics Olympiad. Leanstral 1.5, using pass@4 and a 2 million token per trajectory limit, saturates the benchmark with a perfect MiniF2F-valid score of 244/244 and a MiniF2F-test score of 242/244. We ran additional attempts on the two remaining unsolved problems, and one of them fell after 1 additional pass, the other after 37.

3.1.2 PutnamBench

The PutnamBench benchmark [Tsoukalas et al., 2024] contains 672 formalized problems from the Putnam Mathematical Competition. Table 1 shows the PutnamBench benchmark performance of Leanstral 1.5 compared to the top models on the PutnamBench Leaderboard.[1] Leanstral solves 587 problems using pass@8 with a 4-million-token limit per trajectory. It is the only fully open source system in the shortlist, and the only system using a general coding-agent scaffold rather than a specialized prover workflow.

Table 1: PutnamBench leaderboard. By PutnamBench's definition, Goedel-Architect, Hilbert, and AxProverBase are partially open-sourced, and Leanstral is fully open-sourced. *: The cost of Seed-Prover 1.5 is estimated based on $1/hour for H20s $\times$ the 10 H20 days/problem reported budget.

System	Problems Solved	Cost per Problem ($)
Aleph Prover high inference	668	68
Aleph Prover medium inference	637	54
Goedel-Architect (w/ NL guidance)	597	1.47
Leanstral 1.5	587	1.68
Seed-Prover 1.5	581	240*
Goedel-Architect (w/o NL guidance)	508	0.44
Aleph Prover low inference	500	23
Hilbert	462	244
AxProverBase	365	12.6

Figure 2 shows Leanstral 1.5's test-time scaling on PutnamBench for pass@8. By scaling the total trajectory token budget over multiple compactions, the performance increases from 44 problems solved with a limit of 50k tokens to 587 problems solved with a limit of 4M tokens. We enable auto-compaction every 200k tokens of accumulated trajectory context. The reported token limit is the total budget for the trajectory, not the per-window context size; for example, a 1M-token trajectory can use at most 5 compaction windows.

3.1.3 FATE

The FATE benchmark [Jiang et al., 2025] evaluates formal theorem proving in abstract algebra at three difficulty tiers: undergraduate (FATE-M), graduate (FATE-H), and expert/PhD-level (FATE-X). Table 2 shows the performance of Leanstral 1.5 and other models that reported their FATE scores. Leanstral 1.5 achieves a perfect 100% score on FATE-M, with only 2 attempts per problem. Among provers and reasoners that do not have access to natural language proofs, Leanstral 1.5 achieves the state-of-the-art performance of 87% on FATE-H and 34% on FATE-X with pass@8.

[1] `https://trishullab.github.io/PutnamBench/leaderboard.html`

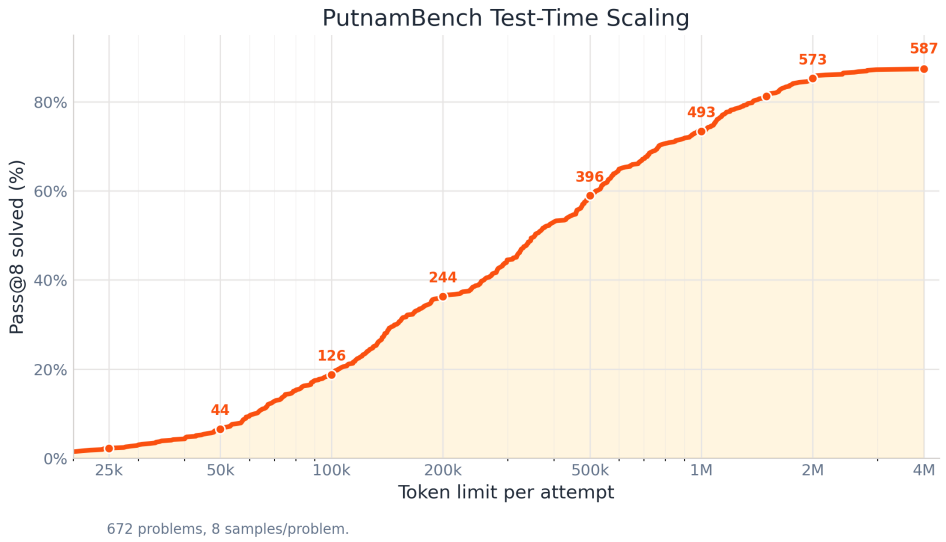

Figure 2: Test-time scaling on PutnamBench. Performance improves as the total trajectory token budget increases to 4M.

Table 2: FATE benchmark results across three difficulty tiers. Unless cited, all non-Leanstral results are from the initial benchmarking of Jiang et al. [2025].

System	FATE-M	FATE-H	FATE-X
Agentic provers			
Leanstral 1.5 pass@2	**100%**	78%	27%
Leanstral 1.5 pass@8	**100%**	**87%**	**34%**
Seed-Prover 1.5 pass@8x8 [Chen et al., 2025]	—	57%	10%
Seed-Prover 1.5 high [Chen et al., 2025]	—	80%	33%
AxProverBase [Pozo et al., 2026]	98%	66%	24%
Reasoners (all pass@64)			
DeepSeek-Prover-V2-671B	63%	3%	0%
OpenAI o3	51%	3%	0%
Goedel-Prover-V2-32B	49%	2%	0%
Kimina-Prover-72B	36%	2%	0%
Claude-Sonnet-4	45%	0%	0%
Gemini-2.5-Pro	40%	0%	0%
DeepSeek-R1	35%	0%	0%

3.1.4 ArXivLean

The ArXivLean benchmark [Dekoninck et al., 2026] evaluates the theorem proving abilities of agents on problems from recent arXiv papers. The statements in the benchmark are autoformalized and manually checked for correctness. Leanstral solves 7 of 41 problems (17.1%), matching Aristotle and GPT-5.5 xhigh while exceeding Gemini 3.1 Pro Preview and Claude-Opus-4.7 high, which both solve 14.6%. This is notable because Leanstral reaches the GPT-5.5 xhigh score at a fraction of the reported inference cost: the leaderboard reports GPT-5.5 xhigh at $4.21 per problem, while Leanstral costs less than $1.5. Aleph Prover tops the leaderboard at 34.2% with a custom test-time scaling scaffold around a frontier model.

3.1.5 FLTEval

To test our Lean SWE capabilities, we introduce a new benchmark, FLTEval, based on PRs from the held out `ImperialCollegeLondon/FLT` repository [Buzzard, 2025], which we chose since it is actively developed, uses the common Lean Blueprint [Massot, 2023] format, and represents an ambitious long-term project formalizing advanced mathematics. The benchmark contains 169 problems

collected using the LeanGym PR pipeline (Section 2.1). Each new or changed theorem/definition is replaced with `sorry`, and the model must complete it. Other PR changes, such as theorem statement changes, are given as a startup patch. Like in the LeanGym PR task, the goal of each problem is to fill in the theorem proofs and definition bodies from a given FLT PR. Unlike the LeanGym PR pipeline, we use the same prompt template for each problem, which includes the PR description (no prompt rewriting). We open-source FLTEval. Leanstral 1.5 pass@8 scores 43.2%, improving on the Leanstral 1.0 pass@8 score of 31.0% announced in our blog post [Mistral AI, 2026a]. Further, Leanstral 1.5 pass@8 outperforms Claude Opus 4.6 pass@1 using less than 1/7th the cost ($216 vs. $1650).

Table 3: FLTEval results. Leanstral models were benchmarked using Mistral Vibe as the scaffold with no evaluation-specific modifications, and Claude models were benchmarked with Claude Code.

Model	Metric	Cost ($)	Score (%)
Leanstral 1.5	pass@8	216	**43.2**
	pass@4	108	38.9
	pass@2	54	34.6
	pass@1	27	28.9
Leanstral 1.0	pass@16	290	31.9
	pass@8	145	31.0
	pass@4	72	29.3
	pass@2	36	26.3
	pass@1	18	21.9
Claude Opus 4.6	pass@1	1,650	39.6
Claude Sonnet 4.6	pass@1	549	23.7
Claude Haiku 4.5	pass@1	184	23.0

Leanstral also demonstrates an efficiency advantage over much larger open-source models: Figure 3 shows the performance of Leanstral 1.0 and 1.5 against models that are 3-10 times larger: GLM5-744B-A40B, Kimi-K2.5-1T-32B, and Qwen3.5-397B-A17B. Qwen3.5 being the strongest open-source competitor, still requires pass@4 to reach 25.4%. In contrast, Leanstral 1.0 achieves 26.3% at pass@2. Leanstral 1.5 takes a step further, reaching 28.9% with a single pass and climbs to 13.5% higher than its strongest open-source counterpart at pass@4.

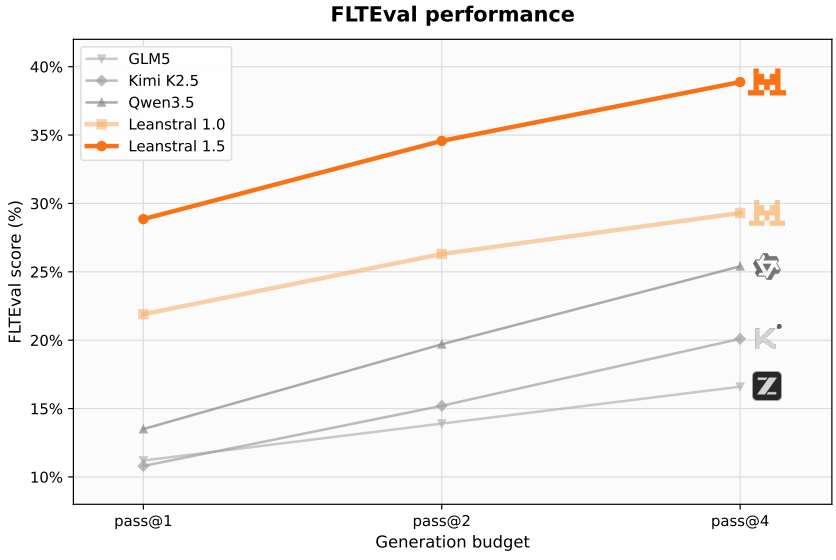

Figure 3: FLTEval score vs. number of passes for Leanstral and open-source models.

4 Case Study: Code Verification

Formal verification has shown a path to provably correct code, but has historically been bottlenecked by the difficulty of writing proofs by hand. Verification agents like Leanstral have the potential to unlock scalable code verification by automating proof generation. We investigated the ability of Leanstral to verify code correctness properties. Crucially, for the first time to our knowledge, we demonstrated a pipeline that *uses LLMs to apply deductive verification scalably, without a human in the loop, to catch bugs in real-world code*.

To explore Leanstral's ability to verify code, we conducted two experiments:

1. a hard verification case study that involved proving properties of an AVL tree, and
2. a practical usage study on bug catching in open-source repositories. Leanstral autoformalized the desired correctness properties of code, and Leanstral verified them, flagging provable violations as potential bugs.

Leanstral completed both tasks, demonstrating generalization beyond the domain of mathematics. Upon running on 57 translated repositories, Leanstral identified 5 previously unknown bugs.

4.1 AVL Trees

AVL trees are self-balancing binary search trees that maintain logarithmic height by rebalancing on insertion and deletion [Adelson-Velsky and Landis, 1962]. As a consequence, lookups, insertions, and deletions stay logarithmic. Leanstral, given a concrete Lean implementation, proved that the implementation delivered on this guarantee, specifically proving three properties: 1. height remains logarithmic in the size of the tree (i.e., the tree remains balanced), 2. insertion into the tree is $O(\log n)$, and 3. deletion of a node is $O(\log n)$. The given specification defines AVL trees recursively. A node contains subtrees, which contain subtrees, so a natural way to prove properties about them is by structural induction, mirroring the structure of the code. This surfaces two challenges: finding the right inductive hypothesis (the property assumed for subtrees is often a stronger statement than what we ultimately want to prove), and unfolding the code at each recursive case to show the inductive step over all execution paths. The latter is made harder by the tree implementation being monadic, interleaving time tracking with control flow (discussed further below). These proof challenges are a common pattern in code verification. Over more than 2.7 million output tokens, Leanstral was able to surmount these challenges and construct the proofs.

Figure 4 illustrates the process of proving the time complexity of insertion. The given AVL tree code includes the insert function `insertM`. It reads like standard insertion for trees representing sets, with the usual traversal depending on whether the inserted value is smaller or greater than the root value, but with an additional rebalancing step (implementation elided for brevity)[2]. Additionally, time is tracked via a monadic `tick` function, implemented using the `cslib` time monad `TimeM` [Barrett et al., 2026]. The insertion function registers one unit of time cost for pattern matching and additional units for node-value comparisons. Each helper function additionally registers its own time costs. The property to prove is that the time cost of `insertM`, as measured by the `time` field of the result which has accumulated the time costs during execution, is bounded by a logarithmic function in the size of the tree. The proof is generated by Leanstral, which identifies a concrete logarithmic function that bounds the time cost as a function of height, provable by induction (helper lemma `insert_time_le_48_height_plus_48`). Height is in turn bounded by a logarithmic function in the size of the tree (helper lemma `height_le_two_log2_size`), allowing Leanstral to complete the proof. The helper lemmas, while not listed in Figure 4, were all generated by Leanstral.

The complexity of the problem required Leanstral, running within Mistral Vibe, to repeatedly self-correct over a proof attempt lasting more than eight hours and spanning 22 compactions. The code's time tracking is woven directly into its control flow via the `TimeM` monad, and the agent's early attempts to separate cost from computation repeatedly stalled: unfolding one monadic layer often left the next bind, return value, or conditional still opaque. Through failed proof states, Leanstral eventually found the right combination of tactics to fully unfold these interleaved computations.

[2]The semantics of the tree here are that it represents a set of values, so duplicate insertions are ignored. Alternatively, one could define the tree to represent a multiset, in which case the insertion function would always insert the new value. The proofs of time complexity would be similar in either case.

Given Specification

Code

```
def insertM [LinearOrder α]
    (x : α) (t : AVLTree α) :
    CostM (AVLTree α) := do
  tick
  match t with
  | .leaf => mkNodeM x .leaf .leaf
  | .node k _ l r =>
      let cmp ← compareM x k
      tick
      if cmp < 0 then
        let l' ← insertM x l
        let t ← mkNodeM k l' r
        rebalanceM t
      else
        tick
        if cmp > 0 then
          let r' ← insertM x r
          let t ← mkNodeM k l r'
          rebalanceM t
        else mkNodeM k l r
```

Property

```
theorem insert_time_le_log_size [LinearOrder α] :
    ∃ C K : Nat, ∀ (x : α) (t : AVLTree α),
      IsAVL t → (insertM x t).time
        ≤ C * Nat.log2 (size t + 1) + K
```

Leanstral Generates

```
by refine ⟨96, 48, fun x t h => ?_⟩
   have h_time : (insertM x t).time
       ≤ 48 * (heightM t).ret + 48 :=
     insert_time_le_48_height_plus_48 x t h
   have h_height : (heightM t).ret
       ≤ 2 * Nat.log2 (size t + 1) :=
     height_le_two_log2_size t h
   have h_total : 48 * (heightM t).ret + 48
      ≤ 96 * Nat.log2 (size t + 1) + 48 := by omega
   exact le_trans h_time h_total
```

Figure 4: Example theorem: insertion is $O(\log n)$. The **given specification** consists of **code** and a theorem expressing its time complexity **property**. Leanstral **generates** a proof (including elided helper lemmas `insert_time_le_48_height_plus_48` and `height_le_two_log2_size`).

For the insertion proof, a second wall arose at the rebalancing step, which triggers complex tree transformations with several execution paths. Leanstral handled these by exhaustive case analysis to carefully account for time costs across all scenarios and established a cost ceiling of 38 steps. With that ceiling in hand, it proved the structural cost recurrence with a bound that happened to be almost tight[3]:

$$(\texttt{insertM}\ x\ t).\texttt{time} \leq 48 \cdot (\texttt{heightM}\ t).\texttt{ret} + 48.$$

Finally, it connected tree height to tree size by showing that balanced trees have logarithmic height. Composing the operational recurrence with this height bound yielded the final logarithmic time theorem. This establishes conclusively that in the implementation being analyzed, insertion is $O(\log n)$ for every possible tree and every possible input.

4.2 Leanstral for Bug Catching

Beyond single-system verification, we investigated the question: can Leanstral find bugs in real code, automatically and at scale? To answer this, we built a pipeline that used Leanstral's code verification capabilities to search for bugs in open-source repositories. Figure 5 illustrates the pipeline. The pipeline targets Rust repositories on GitHub. The automated tool Aeneas [Ho and Protzenko, 2022, Ho et al., 2024] translates the code into Lean. Leanstral then looks at the code to infer the intentions of the programmer. It comes up with properties that the code should satisfy, and Leanstral, running within Mistral Vibe, attempts to prove each property. When proof search fails after 4 attempts with 30 turns, Leanstral instead tries to prove that the property *does not hold* for the code. If successful, it flags the property as pointing to a potential bug. To ensure scalability, the pipeline is fully automated: humans are *not* involved in writing specifications or proofs. In our experiment, 57 repositories were successfully translated by Aeneas, and the pipeline flagged 47 properties as provably not satisfied by the code, of which 11 genuinely pointed to bugs. The remaining flags concerned properties that were too strong (for instance failing because of a state that was unreachable from the public interface) or malformed. Since some properties failed because of the same root cause, this corresponded to 5 bugs, which were all previously unreported on GitHub.

We discuss a concrete example, drawn from the repository `datrs/varinteger`. It implements the function `sign`, shown below, which is part of *zigzag decoding*. Zigzag encoding represents signed

[3]The tight bound would have been 47 ticks.

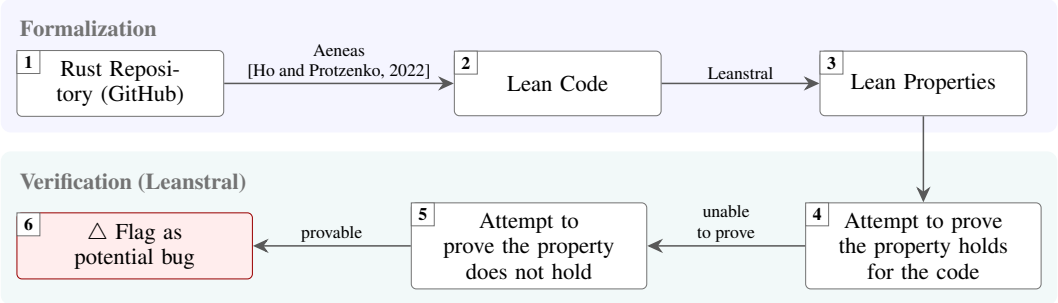

Figure 5: The bug-catching pipeline. Rust code from a GitHub repository is translated into Lean code by Aeneas [Ho and Protzenko, 2022]; Leanstral then, based on the code, autoformalizes expected correctness properties. Leanstral attempts to prove each property (pass@4); on failure it attempts to prove that the code does not satisfy the property (pass@4). Upon completing such a proof, it flags the property as pointing to a potential bug.

integers as unsigned ones, and `sign` is meant to reverse the encoding. It maps even unsigned integers to non-negative signed integers and odd unsigned integers to negative signed integers.

```
fn sign(value: u64) -> i64 {
    if value & 1 != 0 {
        -(((value + 1) / 2) as i64)
    } else {
        (value / 2) as i64
    }
}
```

We would expect this function never to panic, satisfying the following property:

```
def sign_never_panics (value : Std.U64) : Prop := ∃ i, sign value = ok i
```

Therefore, we would expect the following theorem to be provable:

```
theorem sign_never_panics_thm : ∀ (value : Std.U64), sign_never_panics
  value
```

However, on the input `Std.U64.MAX`, the computation `(value + 1)` on the third line overflows. The function crashes in debug mode and silently returns 0 in release mode instead of `Std.i64.MIN`, corrupting the result. Testing and fuzzing would miss this issue unless they specifically tried the input `Std.U64.MAX`. In our pipeline, by contrast, formal verification ensures reasoning about *every single possible input*. Leanstral cannot complete the proof for the theorem `sign_never_panics_thm`; instead, it proves that the property *does not* hold, and flags this as evidence of a potential bug.

```
theorem sign_never_panics_neg :
      ¬ (∀ (value : Std.U64), sign_never_panics value) := by
  intro h
  have hmax := h core.num.U64.MAX
  unfold sign_never_panics at hmax
  rcases hmax with ⟨i, hi⟩
  have hsign : sign core.num.U64.MAX = fail integerOverflow := by rfl
  rw [hsign] at hi
  have : (fail integerOverflow : Result Std.I64) = ok i := hi
  injection this
```

A natural question is what systematic advantages or disadvantages this approach presents in practice relative to other ways to catch bugs, like testing, fuzzing, or simply asking an LLM to identify bugs in a codebase[4]. Obtaining a rigorous answer to this question is a direction for future work.

5 Conclusion

We presented Leanstral, a family of Lean 4 code-agent models that operate directly in the Mistral Vibe scaffold. The central result is that a native code-agent approach can be highly efficient: Leanstral uses a general interactive coding interface, yet reaches strong results on formal mathematics benchmarks and transfers to code verification tasks that were not the primary target of training. In small-scale real-world use, Leanstral has also helped with Lean development in live repositories; for example, it contributed to an accepted PR formalizing the Black–Scholes higher-order Greek "charm" ($\partial\Delta/\partial\tau$).

The development of Leanstral 1.5 also exposed several limitations of long-horizon proof agents. Earlier checkpoints sometimes declared success without compiling the project, refused difficult but feasible proofs, or drifted away from the original objective after repeated context compactions. We addressed these issues by filtering SFT data for false-success and premature-refusal behaviors, and by preserving the original task prompt alongside compacted trajectory summaries. The monotonic test-time scaling in Figure 2 suggests that these changes made extended interaction more useful: additional token budget leads to additional solved problems rather than merely longer failed attempts.

Two directions are especially important for future work. First, we want to build more general code-agent test-time scaffolding for multi-agent collaboration: one agent can decompose a theorem or repository task, others can attack independent subgoals, and a final agent can integrate verified artifacts back into the repository. Multiple agents working in parallel in the verifiable context give great promises for leveraging high throughput to lower end-to-end latency. Second, we need to improve persistence and calibration on research-level problems. In LeanEval-style tasks, Leanstral sometimes inferred that a proof required missing Mathlib dependencies and stopped early; in at least one case, a theorem it described as requiring unavailable infrastructure was solved by another attempt in 587 lines of Lean code. Better retrieval, stronger informal mathematical reasoning, and compaction mechanisms that preserve the original research objective should help close this gap.

Core contributors

Aditi Kabra, Albert Q. Jiang, Andrew Zhao, Dhia Garbaya, Indraneel Mukherjee, Jason Rute, Mert Unsal, Roman Soletskyi, and Simon Sorg.

References

G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

C. W. Barrett, S. Chaudhuri, F. Montesi, J. Grundy, P. Kohli, L. de Moura, A. Rademaker, and S. Yingchareonthawornchai. CSLib: The Lean computer science library. *CoRR*, abs/2602.04846, 2026. doi: 10.48550/ARXIV.2602.04846. URL `https://doi.org/10.48550/arXiv.2602.04846`.

K. Buzzard. Fermat's Last Theorem: Ongoing lean formalisation of the proof of Fermat's Last Theorem, 2025. URL `https://github.com/ImperialCollegeLondon/FLT`.

J. Chen, W. Chen, J. Du, J. Hu, Z. Jiang, A. Jie, X. Jin, X. Jin, C. Li, W. Shi, Z. Wang, M. Wang, C. Wei, S. Wei, H. Xin, F. Yang, W. Gao, Z. Yuan, T. Zhan, Z. Zheng, T. Zhou, and T. H. Zhu. Seed-prover 1.5: Mastering undergraduate-level theorem proving via learning from experience, 2025. URL `https://doi.org/10.48550/arXiv.2512.17260`.

J. Dekoninck, N. Jovanovic, T. Gehrunger, K. Rögnvaldsson, I. Petrov, C. Sun, and M. T. Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with LLMs. *CoRR*,

[4]For example, Claude when given the `datrs/varinteger` repository and asked to identify potential bugs was able to catch the issue with the `sign` function highlighted in this section.

abs/2605.00674, 2026. doi: 10.48550/ARXIV.2605.00674. URL https://doi.org/10.48550/arXiv.2605.00674.

GasStationManager. Safeverify: A Lean4 script for robustly verifying submitted proofs of theorems and implementations of functions, 2024. URL https://github.com/GasStationManager/SafeVerify.

S. Ho and J. Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547647. URL https://doi.org/10.1145/3547647.

S. Ho, G. Boisseau, L. Franceschino, Y. Prak, A. Fromherz, and J. Protzenko. Charon: An analysis framework for Rust, 2024. URL https://arxiv.org/abs/2410.18042.

J. Jiang, W. He, Y. Wang, G. Gao, Y. Hu, J. Wang, N. Guan, P. Wu, C. Dai, L. Xiao, and B. Dong. FATE: A formal benchmark series for frontier algebra of multiple difficulty levels, 2025. URL https://doi.org/10.48550/arXiv.2511.02872.

P. Massot. Lean blueprints: plasTeX plugin to build formalization blueprints, 2023. URL https://github.com/PatrickMassot/leanblueprint.

MiniMax. Minimax-m1: Scaling test-time compute efficiently with lightning attention, 2025. URL https://doi.org/10.48550/arXiv.2506.13585.

Mistral AI. Mistral Vibe: Minimal CLI coding agent by Mistral, 2025. URL https://github.com/mistralai/mistral-vibe.

Mistral AI. Leanstral: Open-source foundation for trustworthy vibe-coding, 2026a. URL https://mistral.ai/news/leanstral/.

Mistral AI. Mistral Small 4, 2026b. URL https://docs.mistral.ai/models/model-cards/mistral-small-4-0-26-03.

A. Poiroux, V. Kuncak, and A. Bosselut. Leaninteract: A python interface for lean 4, 2025. URL https://github.com/augustepoiroux/LeanInteract.

B. R. Pozo, A. Letson, K. Nowakowski, I. B. Ferreiro, and L. Sarra. A minimal agent for automated theorem proving, 2026. URL https://doi.org/10.48550/arXiv.2602.24273.

G. Tsoukalas, J. Lee, J. Jennings, J. Xin, M. Ding, M. Jennings, A. Thakur, and S. Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/1582eaf9e0cf349e1e5a6ee453100aa1-Abstract-Datasets_and_Benchmarks_Track.html.

K. Zheng, J. M. Han, and S. Polu. miniF2F: a cross-system benchmark for formal Olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.