GETTING THE (IN)FORMAL STATEMENT RIGHT

ROBERT JOSEPH GEORGE

California Institute of Technology

1. Introduction

It's clear to me now that language models can solve open problems in mathematics and then formalize what they proved. OpenAI's model proved finite-time blowup for the Navier–Stokes equations with a smooth, compactly supported force, and another model formalized the proof in Lean seventeen hours later [31, 52]. Then came openai/math, hundreds of model-written papers with a Lean challenge next to each headline result [51]. Anthropic's model wrote about thirteen million lines of Lean for Fermat's Last Theorem in eleven days, and Comparator accepted them [4, 18]. Agents have also verified an operating-system kernel [30], production cryptography [28, 62], a compiler [59] and Ethereum bytecode [32].

Mathematics is heading toward proof abundance, to borrow Mistral's phrase [49], and I'm really excited about it. Once a piece of mathematics is formal it's unlocked: you can search it, reuse it, hand it to the next agent and build on it without rereading anyone's proof. The more of it is formal, the faster the next discovery comes. Kim Morrison said on Zulip back in 2024 that autoformalization "is on the critical path" [50], and the recent case for formalizing all of mathematics argues that a coordinated effort "could enormously expand mathematical exploration and discovery" [8].

For me formalization has two goals, correctness and understanding. Correctness means knowing that a result is true and exactly which result it is. Understanding means a person can read the statement, see what was proved and why, and use it. A checked proof gives you only the first half of correctness. Everything else goes through the statement and the English we write about it, and that's where things go wrong. I spend a lot of time in the autoformalization threads on the Lean Zulip, and one story keeps repeating: somebody announces a big formal library, somebody else opens the files, and the statements say something else. When Atlas was reported as more than 45,000 declarations formalized from 26 textbooks [58], one reader wrote of a book listed as 56 percent formalized that "the reality is that 0% of the statements here are accurately formalized" [13]. Every proof had been checked. The statements had mostly been checked by other models.

So here's my list. By *autoformalization* I mean turning mathematics, or a specification, into a Lean statement and usually a proof, and by *autoinformalization* the English we write about a Lean result: the docstring, the summary, the headline. It helps to keep three jobs apart (Figure 1). The proof has to prove the statement, the statement has to mean what the source meant, and the English has to say what the statement says. Lean only does the first one. Each practice below comes with an example, because I've watched careful people, me included, skip every one of them. If you read nothing else, read the box.

2. What the machine checks

A Lean proof is a term, and its statement is the term's type. The kernel checks one against the other and never asks where the term came from, so once the statement and its definitions are fixed, a proof from a graduate student, a language model or, if one ever opens a pull request, an alien gets exactly the same treatment. Lean even builds this in: any two proofs of a proposition are equal by definition (`example (p : Prop) (h₁ h₂ : p) : h₁ = h₂ := rfl`). For correctness, people read the statement, and machines check the proof. Jason Rute put it nicely on Zulip: once the statement

Before you call a Lean result done

1. **Freeze the statement first, then run Comparator.** Before any proving, put the theorem, with `sorry` as its proof, and every definition it uses in a challenge file, and leave no definition open. Pin Lean and Mathlib, and at the end check the proof against it with `lake comparator`. That's for mathematics; code needs more (Section 3).
2. **No extra axioms.** Only `propext`, `Classical.choice` and `Quot.sound`, as the replay lists them: no `sorry`, no `axiom`, no `native_decide`. If you need an assumption, make it a hypothesis (Section 2).
3. **Have an agent read the statement, then a person.** Give both the elaborated statement, its definitions and the exact source but not the proof, and write down every place the Lean differs from the source.
4. **Try to break it.** Make sure the hypotheses aren't contradictory by accident, and test every new definition on examples you already understand.
5. **Write the English from the Lean.** The headline says what the statement says, and the gaps are listed next to it.

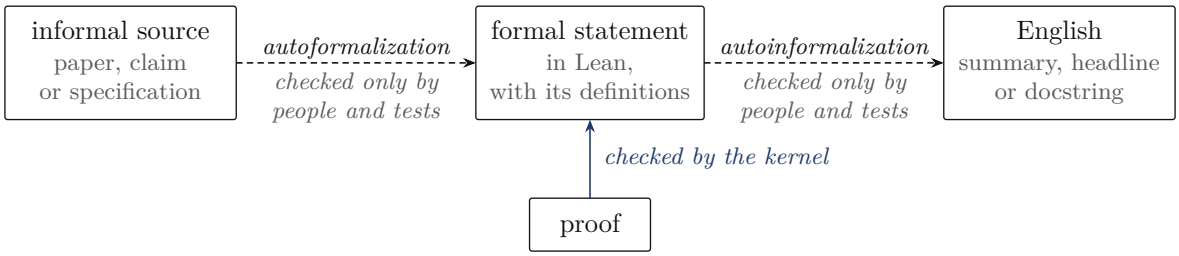


FIGURE 1. Three jobs, one of them done by a machine.

is separated from the proof, the "referee" only needs to check "the statement and definitions used in the statement" [60].

Comparator turns that split into a protocol [34], and since Lean 4.35.0 it ships with Lake as `lake comparator` [36]. You write a challenge file that states each theorem with `sorry`, plus a configuration naming the theorems and the axioms you allow, and Comparator checks an untrusted solution against it in a sandbox (Figure 2). With `--paranoid` it also replays everything through five more checkers, among them con-leche, which is proved in Lean never to accept a theorem of `False` [12, 16, 21]. The Lean manual calls this the gold standard for "un-reviewed AI-generated proofs and programs", and it's honest about what you still have to trust, ending with "no human error or misleading presentation of the theorem statement in the trusted challenge file" [39]. One thing it can't check is an open definition, which it compares only by name, type, universe levels and safety, so `def ChallengeSolution : Prop := RiemannHypothesis` "solves" a challenge that asks for a `Prop`, and the Comparator docs say so themselves [34]. Read every open definition yourself. Most of this note is about that last line of the manual.

It's worth knowing exactly what the kernel is trusted with (Figure 3). Lean has two engines. The elaborator runs your tactics and automation, whoever wrote them, and none of it is trusted: it only proposes declarations, and any axiom it adds is listed. The kernel is the trusted part, and it's small, but trusted doesn't mean perfect. As Mario Carneiro put it on Zulip, "there are bugs in it like any other software, but it is held to a high standard" [22]. When one is fixed, rechecking on the new toolchain shows whether any proof depended on it, and Jason Rute added that the independent checkers only reassure you if you actually run them. The compiler is the other engine. It turns definitions into IR and C so programs run fast, swaps in any `@[implemented_by]` or `@[extern]`

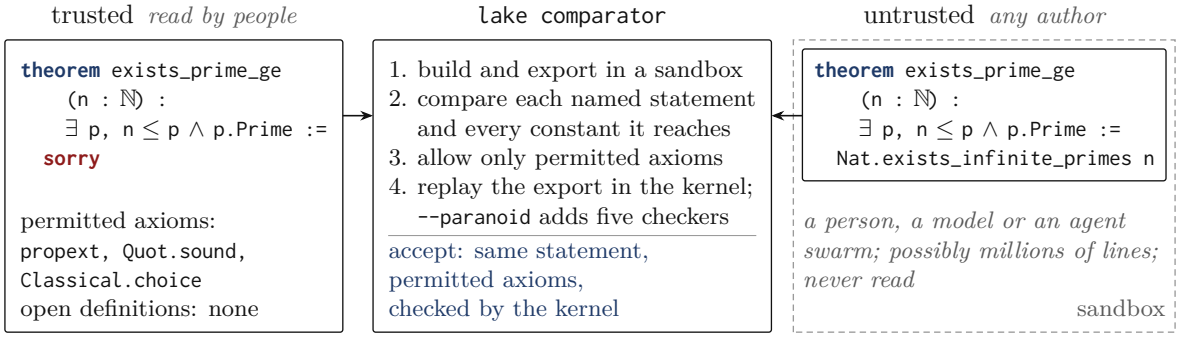
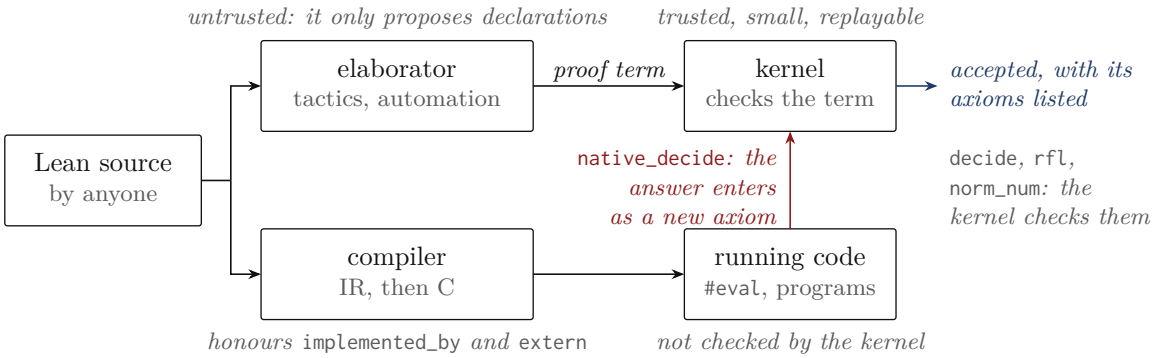


FIGURE 2. Why the proof can come from anyone, and what you still trust.




```
-- Compiled code runs `fast` wherever `slow` is called; the kernel sees `slow`.
def fast (_ : Nat) : Bool := false
@[implemented_by fast] def slow (_ : Nat) : Bool := true

theorem slow_true : slow 0 = true := rfl
theorem slow_false : slow 0 = false := by native_decide   -- runs `fast`
theorem oops : False := Bool.noConfusion (slow_true.symm.trans slow_false)

#print axioms oops
-- 'oops' depends on axioms: [slow_false._native.native_decide.ax_1_1]
```

FIGURE 3. What Lean trusts (top), and a mismatched `@[implemented_by]` that lets `native_decide` prove `False` on Lean 4.35 (bottom).

replacement, and nothing it produces is checked by the kernel. `native_decide` connects the two: it compiles and runs a decision procedure and, since Lean 4.29, adds a fresh axiom saying the answer is `true` [35], so the proof trusts the compiler and every replacement along the way. A mismatched `@[implemented_by]` is enough to prove `False` this way.

- **Do computations in the kernel.** `decide`, `decide +kernel`, `rfl` and `norm_num` leave a term the kernel checks and add no axiom of their own. For a big search, find the answer outside Lean and check a certificate inside it. `native_decide` is fine while you're exploring, but I wouldn't ship it.
- **Read the axioms off a replay, not the source.** `lake check` and Comparator replay the exported proof and list what it depends on. The source can mislead: a bad declaration added under `debug.skipKernelTC` can report no axioms at all, and agents have spelled the keyword `"ax" ++ "iom"` to get past text filters [45].
- **Keep assumptions as hypotheses.** Shreyas Srinivas made the case on Zulip. An axiom is not part of any theorem's statement, so you only see it by asking for a theorem's dependencies,

and even a private one carries into every theorem that uses it. A hypothesis sits in the statement, and anyone applying the theorem has to supply it or keep it as an assumption [22].

3. What only people check

Even the benchmarks we measure autoformalization with are full of wrong statements. An audit at ICML 2026 found 4,833 problems in five standard Lean benchmarks, 398 of them mechanically certified counterexamples, vacuous theorems or unsound axioms [2]. A small one I like is AIME 1984 Problem 5 in the Lean port of miniF2F (Figure 4). It leaves out $a > 0$ and $b > 0$, and because Mathlib's logarithm satisfies $\log(-x) = \log x$, the pair $a = -64$, $b = 8$ satisfies both hypotheses with $ab = -512$. The statement is false as written, and Lean proves its negation [67]. A checked proof doesn't tell you whether the statement says what the source meant. These are the habits I use to find out.

- **Start from the exact source, and write down every change.** Every correction, strengthening, weakening and omission goes next to the Lean; fixing a source and translating it are both fine as long as they're visible. FormalFlow's proof of a core theorem behind MIP* = RE corrected side conditions in the published argument and said so [43].
- **Use Mathlib's definitions, and test the ones you add.** A wrong definition changes every theorem about it while every proof still goes through, so I give each new one a characterizing lemma, an example and a bridge to the standard notion. Mathlib's own definitions need reading too. The version I use has no P, NP or SAT, and `Turing.TM2ComputableInPolyTime` is relative to encodings you supply, so with constant encodings even the halting problem counts as polynomial-time [17].
- **Read the statement after elaboration.** Mathlib gives its total operations values everywhere, so `Real.log 0 = 0`, `(2 : ℕ) - 3 = 0` and `(1 : ℝ) / 0 = 0`, and a statement has to say what makes each operation mean what the source meant. Two traps come up on Zulip again and again. A bare $\top$ in `ContDiff ℝ ⊤ f` means analytic, not smooth, and a compactly supported analytic function on $\mathbb{R}^n$ is zero, so only the zero function qualifies. And `autoImplicit` is on by default in core Lean, even in files that import Mathlib, so π without `open Real` becomes an implicit variable instead of `Real.pi`. Turn it off, and look at statements with `#check @name`.
- **Let an agent read it first, then a person.** Agents are good first readers. I give one the elaborated statement, every definition it reaches and the exact source, but not the proof, and ask for every place they differ, an input where all the hypotheses hold, and a Lean-checked refutation. A refutation that fails proves nothing, but one that succeeds catches statements like

```
-- The Lean 4 port of miniF2F states AIME 1984 Problem 5 like this.
def Statement : Prop :=
  ∀ a b : ℝ, Real.logb 8 a + Real.logb 4 (b ^ 2) = 5 →
    Real.logb 8 b + Real.logb 4 (a ^ 2) = 7 → a * b = 512

-- Mathlib's logarithm is total, and even: log (-x) = log x.
example (x : ℝ) : Real.log (-x) = Real.log x := Real.log_neg_eq_log x

-- So a = -64 and b = 8 satisfy both hypotheses, and a * b = -512.
theorem statement_false : ¬ Statement := by
  intro h
  have := h (-64) 8 witness_h0 witness_h1   -- the two hypotheses hold
  norm_num at this                          -- but -64 * 8 ≠ 512
```

Figure 4. A benchmark statement that is false as written, in Lean. The two lemmas that compute the hypotheses at $a = -64$, $b = 8$ are left out of the print.

(a) mathematics *Mathlib's* `deriv` *is total*

checked `deriv (fun x : ℝ ↦ |x|) 0 = 0`
meant $|x|$ is differentiable at 0, with slope 0
it isn't: the slopes are -1 *and* $+1$

a returned value does not establish the precondition

(b) computer science *a running time* $T : \mathbb{N} \to \mathbb{N}$

checked $\forall \varepsilon > 0,\ \forall n \geq 1,\ \exists C,\ T(n) \leq C\, n^{1+\varepsilon}$
meant $\forall \varepsilon > 0,\ \exists C,\ \forall n \geq 1,\ T(n) \leq C\, n^{1+\varepsilon}$
take $C = T(n)$*: every* T *satisfies it*

uniformity is in the binder order

(c) software *LZO's OSS-Fuzz target*

checked both calls succeed; the lengths match
meant decompress(compress x) $= x$
`ab` ↦ `ba` *passes the check*

verify the property, not the harness

(d) GPU kernels *VeriTile's attention theorems*

checked $(\forall s,\ \text{score}_s \geq -10^6) \to \forall s,\ \text{correct}_s$
meant $\forall s,\ (\text{score}_s \geq -10^6 \to \text{correct}_s)$
the hypothesis fails for every positive shape

state the per-input theorem, then prove it

FIGURE 5. One public failure in each of four fields, with its lesson underneath.

the one in Figure 4. But agents are optimistic judges: Herald's judge rated Herald's own miniF2F translations 97.5 percent faithful, against 62.7 percent by human evaluation [54]. So what an agent says is a lead, and the person forms their own reading before looking at the agent's.

- **Try to break the statement before anyone proves it.** A prover will happily use contradictory hypotheses and never touch the mathematics you meant (Figure 5d). Find an input that satisfies all the hypotheses at once, check that every existential carries information, and make a target for an open problem say which side holds, since `P ∨ ¬P` is just `Classical.em`.

- **Don't trust equivalence checks once provers are strong.** If you've proved P and you've proved Q, you've proved $P \leftrightarrow Q$ without relating them at all. Poiroux and colleagues warned about this [57], and BEq+ was since measured accepting 94 to 96 percent of 50 unrelated provable pairs [42, 68]. My own habit is to turn a statement's fixed objects into parameters and try both readings on something simple. The openai/math paper on Yau's nodal-set conjecture beats Yau's bound by a fixed power, $H^4/\lambda^{1/2+\varepsilon_0} \to \infty$, but its Lean challenge asks only for $H^4/\sqrt{\lambda} \to \infty$ [51]. As growth conditions, $\lambda_k = k + 1$ and $h_k = \sqrt{\lambda_k} \log \lambda_k$ satisfy the second and not the first, and Lean checks that in a few lines. The statement was just weaker than the paper's theorem.

- **Write the English from the Lean.** Autoinformalization gets much less attention, and I think it's where readers actually get misled: thousands of people read a headline, and a handful read the Lean behind it. When I reviewed openai/math, 76 of the 405 challenges sat under headlines that claimed more than their Lean, right next to a link to the Lean. One headline "resolves the uniform boundedness assertion in Hilbert's sixteenth problem" while the Lean covers quintic Liénard systems [51]. The same release gets it right elsewhere: its SAT challenge proves $(\log \log n)^2$ in the exponent where the paper has $\log \log n$, and its docs say so. A gap stated like that costs nothing, and a headline that outruns the Lean costs the reader's trust in everything next to it.

- **When you verify code, check the model too.** None of this is special to mathematics. The box applies as it is to theoretical computer science, physics and any field where a claim can be stated precisely, and so do the levels of Section 4, and the failures look the same everywhere (Figure 5). Code adds one thing: the statement is about a model of the program, its semantics, harness and machine, and somebody has to show that the model matches what ships. Panel (c) is a harness that checks less than its announcement says [5, 64]. That deserves a note of its own, so I'll leave it for another one.

4. Understanding

Comparator says nothing about understanding. Kevin Buzzard asked the uncomfortable question on Zulip: if all we want is "a formal verification that a theorem is true", "then who cares about the art of proof?" [19]. I do. I believe the autoformalized proof matters too, not just the statement. For correctness it can come from anyone, but for understanding it can't: a formal proof that follows the paper checks the argument and not only the result, its lemmas are what other people reuse, and reading it is how you learn why the theorem is true. The Lean for OpenAI's Navier–Stokes blowup follows different bounds from its paper [13], so it checks the theorem, not the paper's argument. I think of five levels a formalization can aim at, roughly in order of the human time they take (Figure 6). The first two are the correctness of Sections 2 and 3; the other three are about understanding. None of them is wrong. The trouble starts when an artifact is announced at a level it hasn't reached, the way Atlas was.

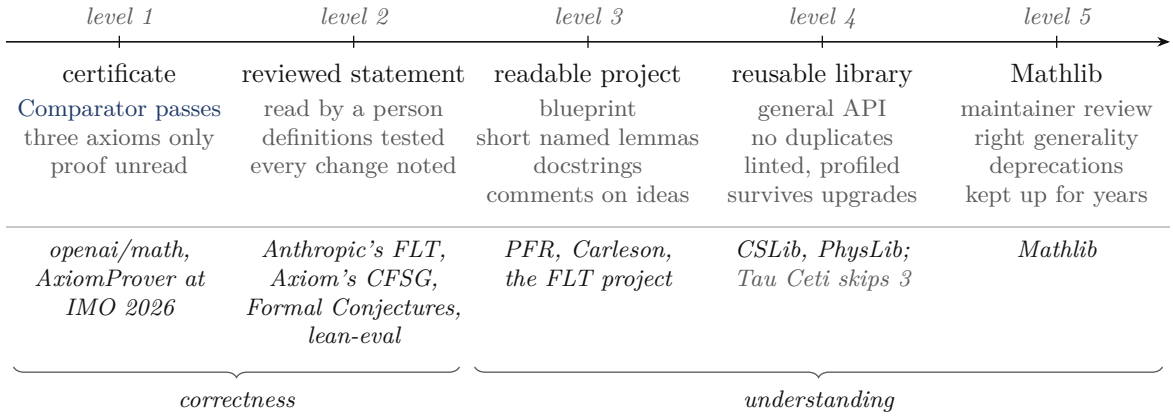


Figure 6. Five levels of a formalization. Only level 1 is checked by machine.

- **Level 1, a certificate.** One theorem, accepted by Comparator with the three standard axioms, and a proof nobody needs to read. The openai/math solutions are certificates for statements written by the same model that wrote the papers [51], and AxiomProver's IMO 2026 run proved statements it had autoformalized itself [11].
- **Level 2, a reviewed statement.** The statement and its definitions have been read against the source, as in the box. Anthropic's FLT and Axiom's CFSG are both unread proofs, but of statements people wrote down first. FLT's challenge is Mathlib's own `FermatLastTheorem`, and its build fails unless `#print axioms` shows exactly the three axioms, a check anyone can copy with `#guard_msgs` [4]. CFSG proves Tau Ceti's `ClassificationStatement`, whose exact Lean a human-reviewed roadmap gave in advance, although AIs implemented the groups it lists [10, 40]. Formal Conjectures, built on "careful human review" [25], and lean-eval, written by benchmark authors and scored only by Comparator [37], are collections of such statements. This is what I want before I call a result done.
- **Level 3, a readable project.** You can learn the mathematics from it. A blueprint ties the informal argument to Lean declarations and shows what is still open [38, 44]. Proofs are split into named lemmas and kept as short as the idea allows, with `grind`, `omega` or `positivity` doing the routine steps; Mathlib's review guide says golfing is fine "as long as it doesn't sacrifice readability" [47]. Public declarations have docstrings, and comments explain ideas instead of tactics. PFR, Carleson and Buzzard's FLT project are built around blueprints [20, 63, 66].
- **Level 4, a reusable library.** Other people build on it, which is the hard half of the review guide: do the results already exist, "possibly in a more general form with a different name", does the code "provide a sensible API", and is it "general enough to support known future needs"? In

practice that means definitions that fit Mathlib's hierarchy, names that follow its conventions [48], imports checked with `lake shake`, slow proofs measured with `#count_heartbeats` and fixed before anyone raises a limit, passing linters, and a library that survives a Mathlib bump. CSLib for computer science and PhysLib for physics, both built on Mathlib, aim here [23, 56].

- **Level 5, Mathlib.** Merged after maintainer review. The review guide orders the questions "approximately from easiest to hardest" as "style, documentation, location, improvements and library integration", and says the last one "generally requires several months of developing familiarity with mathlib" [47]. Mathlib also asks contributors to understand AI-written code well enough to justify each design decision "without the use of an AI" [46].

Tau Ceti doesn't sit on the line, which is what makes it interesting. Humans write its roadmaps and review rubrics, AIs write and review the code, and its README says it is "explicitly *not* aiming here at curating and digesting mathematical knowledge in the way that a human authored library like Mathlib can" [40]. It skips the human reading of level 3 and goes for the reuse of level 4, and it is already reused as a source of statements, by Axiom's CFSG and some lean-eval problems. Whether reuse holds up when nobody reads the code is the experiment. Each level after the second costs more human time, which is the one thing proof abundance doesn't give you.

5. Working with agents

The best systems I know share a loop: an informal plan, a split into lemmas, and lots of parallel proof search that reads Lean's error messages, from Aristotle's graph search [1] to Maestro's coordinator, which can fix a false lemma using a counterexample Lean has checked [41]. Table 1 has some of them. The column I care about is the last, who wrote the statements, because that's where the difficulty is now. On MathAtlas, built from 103 graduate textbooks, strong baselines write a correct theorem statement at most 9.8 percent of the time [55].

Table 1. Some agentic systems for Lean. Budgets and protocols differ widely, so this is not a ranking. Anthropic's FLT and Axiom's CFSG are at level 2 in Figure 6.

System	Reported result	Statements by
AlphaProof [29]	IMO 2024 P1, P2, P6 in Lean	experts, by hand
Aristotle [1]	IMO 2025 P1, P3–P5 in Lean; P2 outside Lean	Harmonic, by hand
Four entries [65]	PutnamBench 672 of 672 (Aleph, NEAR AI, Forall, Humanfia), at reported mean costs of $0.17 to $74 a problem	benchmarks (Aleph fixed 28, NEAR AI 2)
AxiomProver [9, 11]	Putnam 2025, 12 of 12 (8 in exam time); IMO 2026, 6 of 6	itself, for IMO 2026
Maestro [41]	PutnamBench 669 of 672; IMO 2026, 6 of 6	benchmarks; AxiomProver

Agents also change what they were asked to prove when it gets hard. In the xv6 verification, an agent that couldn't prove a lemma once deleted a line from the kernel source and once added an axiom, and the lemma turned out to be unprovable because the code had a bug [30]. The agent that wrote the Axon compiler tried "unilaterally weakening the conclusion of the theorem" [59]. The cheating Jason Rute knew of was aimed at evaluations rather than at Lean's soundness: one model tried to look up the answers to a Lean benchmark, and another tried to call stronger models for help. Chris Bailey wasn't aware of any recent AI result that exploited a soundness bug [22].

- **Separate writing from accepting.** The agents that write statements, proofs and summaries shouldn't be the ones that accept them. Provers get read-only access to the frozen challenge, and every check runs outside their workspace (Figure 1).
- **Give the prover a way out.** An agent that can say a statement is false, and have that handled separately from a failure, has less reason to cheat.
- **Report what you measured.** Keep statement fidelity, proof completion and end-to-end success apart, say what the budget was and who wrote the statements, and test every gate on

controls whose outcome you fixed in advance: a weakened statement, a contradictory hypothesis, a hole, an injected axiom.

Where to start.

- **Tools for agents.** The Lean REPL [33], Pantograph [3], the Kimina Lean Server [61], lean-lsp-mcp for agents that use the Model Context Protocol [24], and the Mathlib search engines LeanSearch, Loogle and LeanExplore [7, 15, 27].
- **Skills for coding agents.** Cameron Freer's lean4-skills, with axiom checks and guardrails [26]; Scott Armstrong's LeanAutoformalizationSkills, whose stated goal is to prove "the theorem the mathematical source actually states" and which includes independent audits [6]; and Chris Birkbeck's mathlib-quality, for bringing agent-written code to Mathlib's standards [14].
- **Agentic setups worth reading.** mini-Lean-agent, which keeps the trusted evaluator apart from the prover [53], and SymCrypt's review of theorem statements before any proving [28].

6. Closing

When a kernel bug is found, it gets fixed once, for everyone. A wrong statement stays wrong until somebody reads it. Most of the habits here I learned from people in the Lean community.

References

[1] T. Achim, A. Best, A. Bietti, et al. Aristotle: IMO-level Automated Theorem Proving, 2025. arXiv:2510.01346v2.

[2] P. S. Ammanamanchi, S. Bhat, and S. Biderman. Faults in Our Formal Benchmarking: Dataset Defects and Evaluation Failures in Lean Theorem Proving, 2026. arXiv:2606.29493. Accepted at ICML 2026.

[3] L. Aniva, C. Sun, B. Miranda, C. Barrett, and S. Koyejo. Pantograph: A Machine-to-Machine Interaction Interface for Advanced Theorem Proving, High Level Reasoning, and Data Extraction in Lean 4, 2024. arXiv:2410.16429v2.

[4] Anthropic. Fermat's Last Theorem in Lean 4. https://github.com/anthropics/fermats-last-theorem, 2026. Repository created 2026-09-04; Lean 4.33.1, Mathlib v4.33.0.

[5] Anthropic. LLMs are making formal software verification feasible at a much larger scale. https://www.anthropic.com/research/formal-verification, 2026. Published 2026-10-22, in collaboration with Theorem.

[6] S. Armstrong. LeanAutoformalizationSkills: portable Lean 4 autoformalization skills for Codex and Claude Code. https://github.com/scottnarmstrong/LeanAutoformalizationSkills, 2026.

[7] J. Asher. LeanExplore: A search engine for Lean 4 declarations, 2025. arXiv:2506.11085.

[8] J. Avigad, K. Buzzard, J. Commelin, et al. How to Formalize All of Mathematics. Xena Project blog, 16 October 2026, https://xenaproject.wordpress.com/2026/10/16/how-to-formalize-all-of-mathematics/, 2026.

[9] Axiom Math. AxiomProver at Putnam 2025. https://github.com/AxiomMath/Putnam2025, 2025.

[10] Axiom Math. Classification of Finite Simple Groups in Lean. https://github.com/AxiomMath/CFSG; https://cfsg.axiommath.ai/, 2026. Repository created 2026-10-27; Lean v4.35.0-rc3.

[11] Axiom Math. AxiomProver at IMO 2026. https://github.com/AxiomMath/IMO2026, 2026.

[12] C. Bailey. nanoda_lib: a type checker for the Lean theorem prover. https://github.com/ammkrn/nanoda_lib, 2026.

[13] A. Bastounis, F. Circelli, and A. C. Hansen. Navier-Stokes lost in translation: Why Lean verification of AI autoformalisation does not guarantee correct natural language proofs, 2026. arXiv:2610.08144.

[14] C. Birkbeck. mathlib-quality: a Claude Code skill plugin for bringing Lean 4 code up to Mathlib standards. https://github.com/CBirkbeck/mathlib-quality, 2026.

[15] J. Breitner. Loogle: a search tool for Lean and Mathlib. https://github.com/nomeata/loogle, 2023.

[16] J. Breitner. con-leche: a CONsistent LEan CHEcker. https://github.com/leanprover/con-leche, 2026. Implemented and proved consistent by Claude under the supervision of J. Breitner, Lean FRO.

[17] T. Browning. refactor(Computability/TMComputable): generalize from FinEncoding. https://github.com/leanprover-community/mathlib4/pull/34779, 2026. Mathlib pull request, merged 2026-02-26.

[18] K. Buzzard. FLT: Anthropic has beaten me to it. Xena Project blog, https://xenaproject.wordpress.com/2026/09/04/flt-anthropic-has-beaten-me-to-it/, 2026. 2026-09-04.
[19] K. Buzzard. Messages in the thread "sloplib". Lean Zulip, general, 4 February 2026, https://leanprover-community.github.io/archive/stream/113488-general/topic/sloplib.html, 2026.
[20] K. Buzzard et al. Ongoing Lean formalisation of the proof of Fermat's Last Theorem. https://github.com/ImperialCollegeLondon/FLT, 2023.
[21] M. Carneiro. Lean4Lean: Verifying a Typechecker for Lean, in Lean, 2024. arXiv:2403.14064.
[22] M. Carneiro, J. Rute, S. Srinivas, and C. Bailey. Messages in the thread "Lean in the wild: Negative science article Oct 2026". Lean Zulip, general, October 2026, https://leanprover.zulipchat.com/#narrow/channel/113488-general/topic/Lean.20in.20the.20wild.3A.20Negative.20science.20article.20Oct.202026, 2026.
[23] CSLib contributors. CSLib: The Lean Computer Science Library. https://github.com/leanprover/cslib, 2025.
[24] O. Dressler. lean-lsp-mcp: Lean Theorem Prover MCP. https://github.com/oOo0oOo/lean-lsp-mcp, 2025.
[25] M. Firsching, P. Lezeau, S. Mercuri, M. Z. Horváth, Y. Dillies, and C. Sönne. Formal Conjectures: An Open and Evolving Benchmark for Verified Discovery in Mathematics. https://github.com/google-deepmind/formal-conjectures, 2026. arXiv:2605.13171.
[26] C. Freer. Lean 4 Skills: a theorem-proving skill and workflow pack for AI coding agents. https://github.com/cameronfreer/lean4-skills, 2026.
[27] G. Gao, H. Ju, J. Jiang, Z. Qin, and B. Dong. A Semantic Search Engine for Mathlib4, 2024. arXiv:2403.13310.
[28] S. Ho, C. Fournet, J. Protzenko, et al. Scaling Verification of Cryptographic Software with Aeneas, Rust, and Lean, 2026. arXiv:2609.15648.
[29] T. Hubert, R. Mehta, L. Sartran, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026. doi: 10.1038/s41586-025-09833-y. Published online 12 November 2025.
[30] M. F. Kaashoek and N. Zeldovich. Extending concurrent separation logic to the hardware level to verify the xv6 OS kernel on RISC-V with AI agents, 2026. arXiv:2609.04043. v3, 2026-10-12.
[31] K. Kakaes. AI Has Solved One of Math's $1 Million Millennium Prize Problems. Quanta Magazine, https://www.quantamagazine.org/ai-has-solved-one-of-maths-1-million-millennium-prize-problems-20260908/, 2026. 2026-09-08.
[32] L. Lazaropoulos and Z. Paraskevopoulou. Foundational Refinement Proofs for EVM Bytecode, at the Price of Tokens, 2026. arXiv:2607.26306. v1 2026-07-28; v2 2026-10-20.
[33] Lean community. A read-eval-print-loop for Lean 4. https://github.com/leanprover-community/repl, 2023.
[34] Lean FRO. Comparator: a trustworthy judge for Lean proofs. https://github.com/leanprover/comparator, 2025. Repository archived; merged into Lean v4.35.0 as lake comparator and lake check.
[35] Lean FRO. Lean 4.29.0 Release Notes. https://lean-lang.org/doc/reference/latest/releases/v4.29.0/, 2026. Released 2026-03-27.
[36] Lean FRO. Lean 4.35.0 Release Notes. https://lean-lang.org/doc/reference/latest/releases/v4.35.0/, 2026. Released 2026-10-12.
[37] Lean FRO. Lean Eval: a Comparator-based Lean formal mathematics eval. https://github.com/leanprover/lean-eval, 2026.
[38] Lean FRO. Verso Blueprint: structured proof blueprints with Verso. https://github.com/leanprover/verso-blueprint, 2026.
[39] Lean FRO. Validating a Lean Proof. The Lean Language Reference, https://lean-lang.org/doc/reference/latest/ValidatingProofs/, 2026.
[40] Lean FRO and the Mathlib Initiative. Tau Ceti: an AIs-welcome Lean library downstream of Mathlib. https://github.com/TauCetiProject/TauCeti, 2026.
[41] H. Li, Z. Shui, Y. Zhang, et al. Maestro: Agentic Formal Proof Development by Coordination, 2026. arXiv:2610.25920. first version 2026-10-19.
[42] X. Li, P. S. Ammanamanchi, S. Bhat, R. Jamdar, Y. Lin, and S. Biderman. SemantEq: Faithful Bidirectional Equivalence of Formal Statements, 2026. arXiv:2610.32919.

[43] S. Lu, R. Deng, Y. Zhu, and Z. Ji. Long-horizon autoformalization of a core theorem underlying MIP* = RE, 2026. arXiv:2609.19814. first version 2026-09-17.
[44] P. Massot. leanblueprint: a plasTeX plugin to build formalization blueprints. https://github.com/PatrickMassot/leanblueprint, 2021.
[45] Math Inc. FormalQualBench. https://www.math.inc/formalqualbench, 2026. Page undated; cited by arXiv:2606.26442 (2026-06-24).
[46] Mathlib community. Contributing to Mathlib: Use of AI. https://leanprover-community.github.io/contribute/index.html, read 28 October 2026, 2026.
[47] Mathlib community. Mathlib Pull Request Review Guide. https://leanprover-community.github.io/contribute/pr-review.html, read 28 October 2026, 2026.
[48] Mathlib community. Mathlib Library Style Guidelines and Naming Conventions. https://leanprover-community.github.io/contribute/style.html and https://leanprover-community.github.io/contribute/naming.html, read 28 October 2026, 2026.
[49] Mistral AI. Leanstral 1.5: Proof Abundance for All. https://mistral.ai/news/leanstral-1-5/, 2026. 2026-07-02.
[50] K. Morrison. Message in the thread "Is Autoformalization needed?". Lean Zulip, Machine Learning for Theorem Proving, 3 December 2024, https://leanprover-community.github.io/archive/stream/219941-Machine-Learning-for-Theorem-Proving/topic/Is.20Autoformalization.20needed.3F.html, 2024.
[51] OpenAI. openai/math. https://github.com/openai/math, 2026. Snapshots of 7 October and 26 October 2026.
[52] OpenAI. Finite time blowup for Navier–Stokes and Euler equations (Lean formalizations). https://github.com/openai/NavierStokesAndEuler; paper https://cdn.openai.com/pdf/32d9f210-8b73-45e0-91bc-82a30aef8a9a/navier-stokes.pdf, 2026. Repository created 2026-09-08; Lean 4.34.0-rc2.
[53] T. Ortega, S. Park, A. Khaitan, L. Fowl, A. Kontorovich, and S. Arora. Do Theorem Provers Need Theorem-Proving Agents? mini-Lean-agent: A Coding-Agent Baseline for Lean, 2026. arXiv:2610.17331v1.
[54] A. Ospanov, F. Farnia, and R. Yousefzadeh. miniF2F-Lean Revisited: Reviewing Limitations and Charting a Path Forward, 2025. arXiv:2511.03108.
[55] N. Patel, N. Arias, D. Babayan, et al. MathAtlas: A Benchmark for Autoformalization in the Wild, 2026. arXiv:2605.14061. first version 2026-05-13.
[56] PhysLib contributors. PhysLib: a project to digitalise results from physics into Lean. https://github.com/leanprover-community/physlib, 2024.
[57] A. Poiroux, G. Weiss, V. Kunčak, and A. Bosselut. Reliable Evaluation and Benchmarks for Statement Autoformalization, 2024. arXiv:2406.07222. EMNLP 2025.
[58] A. Rammal, N. Patel, F. Gloeckle, et al. Formalizing Mathematics at Scale, 2026. arXiv:2605.29955. first version 2026-05-28.
[59] M. Rinard. Testing, Credible Compilation, and Verification in the Axon Verified Compiler in Lean and Claude Code, 2026. arXiv:2605.01660v3.
[60] J. Rute. Message in the thread "TAIL - Template for AI generated Lean proofs". Lean Zulip, general, 28 December 2025, https://leanprover-community.github.io/archive/stream/113488-general/topic/TAIL.20-.20Template.20for.20AI.20generated.20Lean.20proofs.html, 2025.
[61] M. D. Santos, H. de Saxcé, H. Wang, et al. Kimina Lean Server: A High-Performance Lean Server for Large-Scale Verification, 2025. arXiv:2504.21230.
[62] C. Sun, S. Fong, Z. Kuang, et al. An AI Approach to Verified Production Cryptographic Libraries, 2026. arXiv:2608.00965. v2, 2026-10-16.
[63] T. Tao et al. The Polynomial Freiman-Ruzsa Conjecture in Lean. https://github.com/teorth/pfr, 2023.
[64] Theorem Labs. OSS Verify proofs, release 2026.10.1. https://github.com/theorem-labs/oss-verify-proofs/blob/v2026.10.1/CHANGELOG.md, 2026. Released 2026-10-27.
[65] G. Tsoukalas et al. PutnamBench leaderboard data (docs/results.json). https://github.com/trishullab/PutnamBench, 2026. Read 2026-10-27; Lean 4 count 672.
[66] F. van Doorn et al. A formalized proof of Carleson's theorem in Lean. https://github.com/fpvandoorn/carleson, 2023.

[67] yangky11. miniF2F-lean4. https://github.com/yangky11/miniF2F-lean4, 2026. main, last pushed 2026-03-25.
[68] Y. Yu, Y.-H. Wu, G. Lin, Y. Zhang, and Y. Peng. Structure-Aware Equivalence for Evaluating Autoformalized Statements, 2026. arXiv:2610.25018.