AXLE: A Cloud Infrastructure for LEAN 4 Theorem Proving Utilities

Jimmy Xin [1] **Alex Schneidman** [2 †] **Chris Cummins** [3 †] **Karun Ram** [1] **Srihari Ganesh** [1] **Jannis Limperg** [1]

Abstract

We present AXLE (Axiom Lean Engine), a cloud service for LEAN 4 proof manipulation, extraction, and verification. Recent progress in AI for mathematics—reinforcement learning pipelines, agentic proving workflows, dataset curation—demands LEAN 4 tooling that scales to millions of requests while remaining correct and robust; existing infrastructure offers parallel compilation but not scalable proof verification, higher-level proof manipulation, multi-version support, or per-request isolation at the throughput modern AI workflows require. AXLE provides 14 LEAN 4 metaprogramming tools spanning strict proof verification, declaration metadata extraction, semantic source manipulation, lemma extraction, and deterministic proof repair and simplification. The service runs as a multi-tenant cloud deployment with per-request isolation and concurrent support for multiple LEAN 4 and MATHLIB versions, accessible via a Python SDK, command-line interface, web UI, MCP server, and HTTP API. AXLE is publicly available and free to use at `https://axle.axiommath.ai` and via the `axiom-axle` PyPI package, with no local LEAN 4 installation required. It has served over 500 million requests to date and is the underlying infrastructure for Axiom Math's proving efforts, including its 12/12 score on the 2025 Putnam competition.

1. Introduction

LEAN 4 (de Moura & Ullrich, 2021) is a programming language and proof assistant in which users write programs and proofs together, with correctness guaranteed by a dependent type system. Its community-driven mathematics library, MATHLIB (The mathlib Community, 2020), contains over 250,000 formal theorems and has become a central resource for formalization. LEAN 4 has also emerged as the language of choice for AI-driven mathematics: AlphaProof (Hubert et al., 2025), DeepSeek-Prover (Ren et al., 2025), Kimina-Prover (Wang et al., 2025), Seed-Prover (Chen et al., 2025), Aristotle (The Harmonic Team, 2025), Leanstral (Mistral AI, 2026), and more all use LEAN 4 as their formal backend. These efforts—spanning model training, reinforcement-learning pipelines, agentic proving workflows, and dataset curation—share a common need for LEAN 4 tooling that is not only fast and correct, but that supports the high concurrency these workloads require.

Several prior systems provide infrastructure for interacting with LEAN 4 at scale (§6). However, these efforts have various limitations in scope. First, compilation is not verification: A passing compile accepts proofs containing `sorry`, unsound axioms, or incorrectly restated theorems—all common failure modes in AI-generated proofs. Strict verification techniques exist (Lean FRO, 2023; 2025; GasStation-Manager, 2025), but no prior system makes them available as a fast, scalable cloud service. Second, prior work lacks additional higher-level manipulation primitives and metadata extraction that agentic and dataset-curation workloads depend on, all of which require custom functionality implemented in LEAN's metaprogramming framework. Third, shared-process architectures risk state leakage and crash propagation across requests. Fourth, most infrastructure is pinned to a single LEAN 4 or MATHLIB version, whereas AI systems frequently target multiple versions for training data from different eras or for cross-version compatibility. Finally, existing systems do not offer elastic scalability under a simple interface: automatically scaling capacity up during training runs and down during idle periods, with load balanced across workers.

AXLE (Axiom Lean Engine) was built to address these gaps. It provides 14 LEAN 4 metaprogramming tools, served from a hosted multi-tenant cloud deployment with per-request isolation and concurrent support for multiple LEAN 4 versions. AXLE has been deployed in production, including as the underlying infrastructure for Axiom Math's 12/12 score on the 2025 Putnam competition and autoformalizations of Fel's conjecture on numerical semigroups (Chen et al., 2026b) and the partial-regularity-of-primes result related to Vandiver's conjecture (Chen et al., 2026a), among others.

[†]Work done while at Axiom Math. [1]Axiom Math [2]Thinking Machines Lab [3]Recursive Superintelligence, Inc. Correspondence to: Jimmy Xin <axle@axiommath.ai>.

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

The service is free and publicly accessible; Appendix D collects the relevant resource URLs, installation commands, and support channels.

Our contributions are:

1. **Complete suite of proof-manipulation tools.** AXLE provides 14 metaprogramming tools, each implemented as a LEAN 4 metaprogram: strict proof verification, declaration metadata extraction, semantic source manipulation, lemma extraction, and deterministic proof repair and simplification.

2. **Strict proof verification at scale.** AXLE's `verify_proof` rejects candidates containing `sorry`, non-whitelisted axioms, or signature mismatches—failure modes that are common in AI-generated proofs and that plain compilation accepts. AXLE makes strict LEAN 4 proof verification available at production throughput rather than as a single-process command-line tool.

3. **Multi-environment support.** A single AXLE endpoint serves multiple LEAN 4 and MATHLIB versions concurrently, enabling cross-version compatibility checks and seamless support for models targeting different LEAN 4 versions.

4. **Free public cloud service.** AXLE is deployed as a free, publicly accessible cloud service with per-request isolation and automatic retries; users access it through a web UI at `https://axle.axiommath.ai`, a Python SDK and CLI via the `axiom-axle` PyPI package, an MCP server, and an HTTP API, none of which require a local LEAN 4 installation.

2. Motivating Applications

AXLE serves the proof-engineering needs of AI-driven LEAN 4 mathematics. Although the tools and infrastructure are general, the principal workloads fall into three categories: reinforcement learning and model training, agentic proving, and dataset curation. Each places distinct demands on AXLE's tools, latency profile, and scalability; subsequent sections describe the tools and infrastructure that meet them.

2.1. Reinforcement Learning and Model Training

Reinforcement learning over formal proofs has become a central paradigm for training LEAN 4 proving models (Hubert et al., 2025; Ren et al., 2025; Wang et al., 2025; Chen et al., 2025; The Harmonic Team, 2025). A typical RL step samples thousands of candidate proofs from a policy, evaluates each, and uses the outcome as a reward signal. This workload bursts at each rollout boundary into a flood of concurrent verification requests, and demands strict correctness: A verifier that accepts `sorry`, unsound axioms, or restated theorems would teach the policy to game the reward (see Appendix C). Per-request isolation is equally important, since a malformed candidate must not corrupt subsequent verifications on the same worker. AXLE's strict-verification tool `verify_proof` (§4) and its per-request isolation directly target this regime, and the service handles the concurrent verification load these rollouts produce. Raw speed matters too: A typical verification completes well under a second, keeping step-level rollout times bounded even with thousands of parallel candidates. Importantly, `verify_proof` adds little overhead beyond plain compilation, so the strictness checks do not penalize throughput.

2.2. Agentic Proving

A second class of workloads is *agentic* pipelines (Thakur et al., 2024; Ospanov et al., 2025; Varambally et al., 2025; Math Inc., 2025; Liu et al., 2026), in which an LLM iteratively proposes, inspects, and refines proof candidates. A single attempt may involve dozens of turns: Propose a proof, verify it, inspect errors or remaining goals, extract those goals as standalone lemmas, attempt them recursively, repair localized failures, and merge the resulting fragments into a complete proof. These workflows demand a richer toolkit than verification alone—lemma extraction from `sorry` positions and elaboration errors, declaration metadata and dependency analysis for retrieving relevant context, merging of partial proofs from parallel attempts, and deterministic repair of common errors in LLM-generated drafts. Sitting inside the agent's inner loop, the tools must be fast, robust, and stable; they must be invoked dozens of times per attempt without bottlenecking the agent or producing spurious errors that are uninterpretable to an LLM. Appendix B demonstrates one such decompose-solve-merge loop end-to-end on a non-trivial LEAN 4 lemma.

2.3. Dataset Curation and Filtering

A third class of workloads is the construction and filtering of training datasets from existing LEAN 4 corpora such as MATHLIB. Common operations include extracting theorem-level metadata in bulk, filtering by structural properties (dependency complexity, proof length, tactic usage), deduplicating across multiple sources, and normalizing or simplifying statements for downstream consumption. These workloads are throughput-bound rather than latency-bound—a single build may issue millions of requests over hours or days—and can span multiple LEAN 4 and MATHLIB versions, since training corpora are often pinned to specific releases or built across version boundaries. AXLE's high request throughput, concurrent multi-version support, and metadata- and dependency-extraction tools are aimed at this workload.

3. Service Overview

AXLE is a hosted, multi-tenant cloud service. A client issues a tool request—a tool name, an input payload, and a target LEAN 4 environment—to AXLE's HTTPS endpoint, and receives a JSON response with the tool's structured output. Authentication is by API key. No local LEAN 4 installation is required.

3.1. Client Surfaces

Clients reach AXLE through any of five interchangeable methods for submitting JSON-encoded tool calls: a Python SDK and a command-line interface, both distributed via the `axiom-axle` PyPI package; an MCP server distributed via `axiom-axle-mcp` (or hosted at https://mcp.axiommath.ai) for agent integrations; a hosted web UI at https://axle.axiommath.ai; and the underlying HTTPS API.

3.2. Multi-Version Environments

Each request carries an `environment` field that selects a particular LEAN 4 version paired with a MATHLIB snapshot and any project-specific pre-built dependencies. A single AXLE deployment serves multiple environments concurrently behind the same endpoint, which AI workloads use for cross-version compatibility checks, evaluation against legacy proofs, and serving models trained against different MATHLIB eras. The available environments are exposed by the API; by default, the public tier serves a range of recent LEAN 4–MATHLIB release snapshots, with additional custom environments available on request.

3.3. Per-Request Isolation

Each request runs in its own sandboxed process. State a request mutates—loaded definitions, set options, registered attributes, allocated memory—does not persist into any later request, and a crash or runaway elaboration in one request does not affect concurrent or subsequent requests. The sandbox additionally blocks network access and prevents the candidate from writing to the filesystem—useful for AI-generated input that may contain unbounded loops or unintended side effects.

3.4. Scaling and Reliability

AXLE scales in response to sustained load and uses queueing to absorb short request bursts from existing capacity. Per-request wall-clock timeouts are configurable by the caller, up to a service-wide cap; transient infrastructure failures (worker crashes, network errors) trigger automatic retries without surfacing to the client. Per-API-key fair-share queueing ensures one heavy caller cannot starve other tenants on the deployment.

4. Tools

All of AXLE's tools are implemented as LEAN metaprograms operating on parsed and elaborated LEAN syntax. This section walks through the principal tools; a complete summary is given in Table 1, and per-tool reference documentation is available online.[1]

4.1. Compilation Checking and Proof Verification

The simplest tool is `check`, which compiles a LEAN file and reports any errors, warnings, and linter messages. Its strict counterpart, `verify_proof`, takes a candidate file and a separate *formal statement* file defining the theorems to be proved, and accepts the candidate only if its declarations match the formal statements and constitute a valid proof.

Compilation alone is frequently unsuitable for AI-generated proof workloads. LEAN 4 permits user-defined macros, custom tactics, metaprograms, and new axioms; the same features that make LEAN expressive also create many ways for a file to compile without proving the intended statement. The `verify_proof` tool rejects any declarations that:

- contain `sorry`;
- use any axiom outside the small whitelist of known-consistent axioms shipped in the LEAN standard library (`propext`, `Quot.sound`, `Classical.choice`);
- have a type which does not match the formal statement (e.g., the candidate proves a weakened or restated version of the theorem);
- are marked `unsafe`, and can therefore use kernel-bypassing primitives such as `unsafeCast`.

AXLE intentionally does not defend against certain rare classes of adversarial inputs, prioritizing performance over maximal strictness. Several existing tools provide stronger forms of strict proof verification, including `lean4checker` (Lean FRO, 2023), Comparator (Lean FRO, 2025), and SafeVerify (GasStationManager, 2025). For scalability, `verify_proof` assumes that every declaration in the loaded LEAN environment was added through LEAN's normal kernel-checked elaboration path: it does not re-verify the environment from scratch and hence does not defend against inputs that use LEAN metaprogramming to install unchecked declarations directly into the environment and make invalid proofs appear valid. For the typical case of AI-generated proofs this is rarely an issue (§5.2), and `verify_proof` is substantially faster and more scalable than the alternatives, as Section 5 discusses. Appendix C gives concrete examples for this discussion.

[1] https://axle.axiommath.ai/v1/docs/

4.2. Declaration Extraction

`extract_decls` extracts structured metadata for every declaration in a file—theorems, definitions, structures, instances, and so on. Its principal output for each declaration is a self-contained LEAN snippet that includes the declaration together with all its transitive dependencies within the file, ready to compile in isolation. Alongside this, `extract_decls` returns per-declaration metadata: name, kind, signature, body, source-line range, a stable type hash, whether the declaration contains `sorry`, lightweight statistics such as proof length and tactic counts, any LEAN-emitted messages, and dependency information at three levels: *type* (constants appearing in the stated type), *value* (constants in the elaborated body, including intermediate constants used by the elaborator during proof construction), and *syntactic* (constants the source code names literally). Dataset pipelines use `extract_decls` to turn a multi-declaration file into one standalone snippet per declaration; agents use it to isolate the chunk of a larger file they need to work on.

4.3. Semantic Merge

`merge` is the inverse of `extract_decls`: It takes a collection of LEAN source fragments and combines them into a single compilable file. Naïve concatenation does not work in general—declarations must appear in dependency order, duplicate declarations must be unified, and name conflicts must be resolved.

`merge` handles each. Duplicate declarations are detected by α-equivalence of types, optionally using definitional equality. Among duplicates, `merge` keeps a single compiling representative—that also has only compiling dependencies—if one exists; otherwise it keeps one invalid representative. Surviving declarations are then topologically sorted by their dependencies. When two declarations share a name but their types do not match, the conflict is resolved by renaming one of them and updating references using AXLE's `rename` tool.

A typical application is combining proof fragments from multiple AI candidates into a single compiling file—a routine operation in agentic workflows.

4.4. File- and Theorem-Level Transformations

AXLE exposes a family of source-level transformations. Among these are `rename`, which renames a declaration and propagates the change to every reference site—handling namespace resolution, dot notation, constructor projections, and named structure fields—and `normalize`, which expands abbreviated names, unfolds notation, and removes `section` and `namespace` blocks while preserving compile semantics like maintaining `noncomputable` modifiers from a removed section. Other transformations include `theorem2lemma`, which swaps the `theorem` keyword for `lemma` (and vice versa), and `theorem2sorry`, which replaces proofs with `sorry` stubs while leaving statements intact. Several of these (for example, `theorem2sorry` and `normalize`) are routine building blocks in dataset-preparation pipelines: `theorem2sorry` turns finished proofs into sorry-stubbed problem statements, and `normalize` standardizes surface syntax across corpora before training.

4.5. Lemma Extraction

`have2lemma` and `sorry2lemma` lift proof obligations from inside a partial proof into standalone top-level lemmas. The mechanism is built on MATHLIB's `extract_goal` tactic: At the target location—an unresolved `have` for `have2lemma`, an explicit `sorry` or the position of an elaboration error for `sorry2lemma`—the tool captures the proof state, generalizes over its local context, and emits a fresh lemma whose statement equals the captured goal.

The extracted lemma is callsite-reconstructable: AXLE can additionally rewrite the original proof to invoke the new lemma in place of the original obligation, with the correct arguments supplied. A separate transformation tool, `have2sorry`, replaces `have` steps with `sorry` stubs without extracting a lemma; it is useful for sketching proof outlines that subsequent calls fill in. An agent stuck on a proof step uses `have2lemma` or `sorry2lemma` to isolate the obligation, solves the resulting lemma independently, and optionally reassembles the pieces with `merge`.

We note that due to bugs and limitations in existing utilities like `extract_goal`, this tool, and several others, may fail on certain inputs. More details on the failure modes can be found in the documentation.

4.6. Proof Optimization

`simplify_theorems` and `repair_proofs` transform existing proofs through pipelines of small, deterministic passes—`simplify_theorems` to tidy up a compiling proof, `repair_proofs` to coax a failing one into compiling.

`simplify_theorems` prunes structure the proof does not actually use, like tactics flagged as unused or unreachable by LEAN's linters, and `have` steps whose hypothesis is never referenced. The pipeline iterates to a fixed point, with each pass surfacing new opportunities for the next; if any pass introduces an error, the iteration is rolled back, so the tool never breaks a working proof.

`repair_proofs` targets common failure modes in AI-generated drafts by attempting to prove residual unproven goals, truncating proofs when there are no

Tool	Output
`check`	errors, warnings, linter messages
`verify_proof`	verification pass/fail, failed declarations
`extract_decls`	per-declaration metadata, dependencies, standalone snippets
`merge`	merged compilable file with conflicts resolved
`normalize`	normalized source preserving semantics
`rename`	file with renames applied and references updated
`repair_proofs`	repaired file with unfixable goals left as-is
`simplify_theorems`	file with unused tactics and hypotheses pruned
`disprove`	counterexample or proof of negation
`theorem2sorry`	file with proofs replaced by `sorry`
`theorem2lemma`	file with `theorem` swapped for `lemma`
`have2lemma`	top-level lemmas extracted from `have` positions
`have2sorry`	file with `have` steps replaced by `sorry`
`sorry2lemma`	top-level lemmas extracted from `sorry` positions

Table 1. Complete list of AXLE's tools and their primary outputs.

goals remaining, attempting to replace unsafe constructs that `verify_proof` would reject (like the `native_decide` tactic), and cleaning up cross-version porting issues such as `set_option` commands naming options that no longer exist.

5. Evaluation

We evaluate two aspects of AXLE's design: per-request cost and aggregate throughput against alternative ways to run LEAN at scale (§5.1), and strict-proof-verification quality and speed against other published strict checkers (§5.2). The throughput workload is drawn from the public `banach1729/goedel-workbook-lean427` dataset (banach1729, 2026), a set of Lean Workbook competition-math proofs pinned to LEAN 4.27.0. The strict-checker comparison uses an internal paired `verify_proof` corpus drawn from production traffic. To keep the candidate set identical across systems on the throughput workload, every request's import block is normalized to a single `import Mathlib`; candidates with non-Mathlib dependencies are dropped (discussed as a limitation in §7). Additional experiments are in Appendix A, with setup details in §A.1.

5.1. Latency and Throughput on a Single Machine

Table 2 compares AXLE to two alternatives on a randomly-sampled 5,000-proof `check` workload from the Goedel workbook dataset: Kimina Lean Server (Dos Santos et al., 2025)—a long-running LEAN REPL pool—and a baseline that spawns a fresh LEAN process per request. All three run at concurrency 8 on a single `r7a.2xlarge` AWS instance (8 vCPU, 64 GiB), with a 150 s per-request budget.

All three systems return the same verdict on every one of the 5,000 proofs (Table 2). On throughput, raw LEAN is about 2× slower than the other two: it pays the full cold-start cost on each request of reloading MATHLIB modules, which dominates when the median proof needs less than a second of actual elaboration. AXLE and Kimina both keep MATHLIB preloaded and avoid paying the start-up cost. Kimina is marginally faster (0.75 s vs. 1.05 s latency, 2.13 req/s vs. 2.09 req/s throughput). This gap comes from AXLE's per-request isolation (§3): every request runs in a sandboxed worker process, which adds overhead, while Kimina dispatches into a warm REPL from a long-running pool. Appendix A.2 reports an AXLE-only concurrency scaling experiment on the same workload.

5.2. `verify_proof` vs. Strict Proof Checkers

`verify_proof` is one of several published tools that aim to perform strict proof checking in LEAN 4: Given a candidate and a formal statement, decide whether the candidate proves the statement. We compare it against Comparator (Lean FRO, 2025) and SafeVerify (GasStationManager, 2025) on a set of 1,000 production `verify_proof` requests, running each alternative on the same `r7a.4xlarge` (16 vCPU, 128 GiB) at concurrency 4. The per-request budget is 600 s.

Speed. Table 3 reports the per-request median latency, aggregate throughput, total wall time, and success rate. `verify_proof`'s median is sub-second, roughly 10× faster than SafeVerify and 99× faster than Comparator at the median. The throughput gap is dominated by each alternative's per-request defense work. In addition to reloading imports on every request, both Comparator and SafeVerify run an *environment replay* step that re-typechecks each declaration to verify its soundness, and they include additional safeguards such as sandboxing, rebuilding expressions, and detecting suspicious use of `Nat` literals. `verify_proof` trusts that constants were added through the standard kernel-checked elaboration path (§4, Appendix C) and so does not re-check the environment. `verify_proof` finishes the workload at 0.43 req/s, while SafeVerify runs at 0.107 req/s and Comparator at 0.026 req/s.

Agreement. AXLE and Comparator agree on every one of the 938 conclusive pairs (Table 4). The remaining 62 are

system	median latency	p90 latency	throughput	total wall	verdict counts
AXLE	1.05 s	5.96 s	2.09 req/s	2389 s	4708 / 287 / 5
Kimina Lean Server (Dos Santos et al., 2025)	0.75 s	6.38 s	**2.13 req/s**	2351 s	4708 / 287 / 5
LEAN invoked directly (no shared MATHLIB)	5.14 s	9.81 s	1.03 req/s	4873 s	4708 / 287 / 5

Table 2. Per-request latency and aggregate throughput on a 5,000-proof `check` workload from the Goedel workbook dataset, at concurrency 8 on a single `r7a.2xlarge`. "Verdict counts" is pass / fail / timeout at the 150 s budget. All three systems return the same verdict on every one of the 5,000 proofs.

system	median latency	throughput	total wall	success
AXLE `verify_proof`	0.97 s	**0.43 req/s**	9,205 s	**100.0%**
SafeVerify (GasStationManager, 2025)	10.1 s	0.107 req/s	37,258 s	99.9%
Comparator (Lean FRO, 2025)	95.7 s	0.026 req/s	151,267 s	93.8%

Table 3. Per-request median latency and aggregate throughput on 1,000 paired production `verify_proof` requests; all three systems use the same instance at the same concurrency (Appendix A.1). "Success" is the share of requests that produce an accept-or-reject verdict within the 600 s per-request budget; the remaining are heavier candidates that exhaust the budget.

Comparator-side timeouts at the 600 s budget: Comparator's median latency is 95.7 s and its slowest candidates cross 10 minutes. AXLE and SafeVerify agree on 992 of 999 conclusive pairs. The 7 disagreements all arise from a single mechanism: When a candidate places `private` helpers inside a namespace, LEAN stores each private declaration under a mangled symbol that includes the surrounding module name, and the same source compiled into two different module names ends up with two distinct sets of mangled symbols. SafeVerify performs a whole-environment comparison between the target and submission `.olean` files and flags every such mangled symbol as missing. However, the user-named theorem itself matches in both files, and so `verify_proof` and Comparator both accept the submission. Appendix C.2 shows the pattern with an example.

None of the disagreements reflect a soundness issue on `verify_proof`'s part. We did not encounter naturally-occurring requests that incorrectly pass `verify_proof` while failing Comparator or SafeVerify, although such examples can be deliberately crafted (§C.3).

6. Related Work

6.1. LEAN 4 Services and Tooling

The Lean REPL (Lean community, 2023) provides JSON-over-stdio access to a long-running LEAN process. LeanInteract (Poiroux et al., 2025) wraps it as a Python library for parallel processing; leanclient (Dressler, 2025) and lean-lsp-mcp[2] take a different route and wrap LEAN's LSP, the latter packaged as an MCP server for agentic proof sessions. The Kimina Lean Server (Dos Santos et al., 2025) pools many REPL instances behind a REST API with header-level LRU caching, aimed at high-throughput verification in RL training. It matches AXLE in throughput. However, none of these provide a scalable hosted service, isolation between requests, multi-version support, or anything beyond compilation and tactic-state access.

LeanDojo (Yang et al., 2023) pairs a Python interface for proof-state interaction with an extracted dataset and a retrieval-augmented prover; Pantograph (Aniva et al., 2025) exposes LEAN 4's proof states for tactic-level interaction and tree-search procedures; ProofWala (Thakur et al., 2025) provides a multilingual Coq/Lean framework for proof-step data collection and parallel proof search. These operate at the tactic-state level, requiring a long-running proof-state object AXLE deliberately does not maintain; AXLE works at the declaration level (file in, transformed source or per-declaration metadata out) and is request-scoped and stateless, which is what enables the tool suite (§4) and per-request isolation. Jixia (FrenzyMath, 2024) is a command-line static-analysis tool that extracts source-level metadata—declarations, symbols, elaboration info, and the parsed AST—from a compiled LEAN 4 file through a plugin architecture; it is closest in scope to a single AXLE tool (`extract_decls`).

6.2. Strict Proof Verification

lean4checker (Lean FRO, 2023), Comparator (Lean FRO, 2025), and SafeVerify (GasStationManager, 2025) run candidates in more scrutinous settings than `verify_proof`, catching the environment-manipulation attacks of Appendix C at the cost of speed and concurrency. AXLE's deliberate trade-off is throughput on cooperating-client populations; the alternatives suit verification of untrusted external code. Note that lean4checker inspects a candidate environment but does not verify proofs against an external *formal statement*, so it is not included in the comparison in §5.2.

[2] https://github.com/oOo0oOo/lean-lsp-mcp

comparison	both accept	both reject	vp stricter	alt. stricter	inconclusive	agreement
`verify_proof` vs. Comparator	58	880	0	0	62	**100.00%**
`verify_proof` vs. SafeVerify	51	941	0	7	1	**99.30%**

Table 4. Verdict agreement on 1,000 paired production `verify_proof` requests. "vp stricter" indicates that `verify_proof` rejected and the alternative accepted; "alt. stricter" indicates the reverse. "Inconclusive" counts requests where one side produced no clean verdict (excluded from the agreement percentage).

6.3. Agentic Proving Pipelines

APOLLO (Ospanov et al., 2025) and HILBERT (Varambally et al., 2025) are agentic systems that combine LLM reasoning with formal verification through decompose-repair-recombine loops. They exercise exactly the operations AXLE exposes (sub-lemma extraction, source repair, semantic merging), but implement them as heuristic string- or model-based orchestrations around the Lean REPL rather than as LEAN metaprograms behind a shared service.

7. Conclusion

AXLE is a cloud service for LEAN 4 proof manipulation, built on the premise that these tools belong in consolidated infrastructure for AI4math rather than remaining afterthoughts in each downstream system. Its 14 LEAN 4 metaprograms—spanning capabilities in verification, source manipulation, dependency and lemma extraction, simplification, and repair—serve all three principal AI4math workloads (reinforcement learning of prover models, agentic proving pipelines, and scalable dataset curation) from a single multi-tenant service with concurrent multi-version support.

7.1. Limitations

AXLE has two limitations worth noting.

Multi-file projects. Custom LEAN projects can be added as additional environments (Section 3), but onboarding them is higher-friction than selecting a new LEAN version: The project must be built and registered ahead of time, with no way to drop in arbitrary user-supplied dependencies at request time. As a result, the public service does not offer generalized multi-file support. Each environment exposes a fixed import header, typically just `import Mathlib`, and external imports beyond what that header provides are not accepted.

Proof search and interactive sessions. AXLE is built around request-scoped, stateless tool invocations and does not support proof-tree search, interactive proof sessions, or tactic-state-level manipulations—all of which require a long-running proof-state object AXLE does not maintain. Users who need that style of interaction should look at tools like Pantograph (Aniva et al., 2025) and lean-lsp-mcp.

7.2. Future Work

Concrete directions include broader multi-file support beyond fixed-header environments, detailed proof state extraction, and additional capabilities for tools like `simplify_theorems` and `repair_proofs`. We welcome community suggestions for additional tools and features.

Acknowledgements

We thank the Lean FRO for their help and feedback, the rest of the Axiom Math team as AXLE's initial internal users, and our early public users for their feedback during the initial release.

Impact Statement

AXLE is freely available infrastructure for AI-driven LEAN 4 formal mathematics. We see no specific societal risks beyond those already attendant on advances in machine learning for theorem proving.

AI Disclosure AI tools were used for prose drafting and editing throughout. All design decisions and quantitative results are the authors'.

References

Aniva, L., Sun, C., Miranda, B., Barrett, C., and Koyejo, S. Pantograph: A machine-to-machine interface for advanced theorem proving, high-level reasoning, and data extraction in Lean 4. In Gurfinkel, A. and Heule, M. (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 104–123. Springer Nature Switzerland, 2025. doi: 10.1007/978-3-031-90643-5_6.

banach1729. Goedel-workbook-lean-4.27: Lean workbook competition-math proofs from DeepSeek-Prover-V1.5, migrated to Lean 4.27.0 / Mathlib v4.27.0. https://huggingface.co/datasets/banach1729/goedel-workbook-lean427, 2026.

Chen, E., Cummins, C., Eltschig, B., et al. Almost all primes are partially regular. *arXiv preprint arXiv:2602.05090*, 2026a.

Chen, E., Cummins, C., Grubisic, D., et al. Fel's conjecture on syzygies of numerical semigroups. *arXiv preprint arXiv:2602.03716*, 2026b.

Chen, L. et al. Seed-Prover: Deep and broad reasoning for automated theorem proving. *arXiv preprint arXiv:2507.23726*, 2025.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In *Automated Deduction (CADE 28)*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Dos Santos, M., de Saxé, H., Wang, H., Wang, R., Bakšys, M., Ünsal, M., Liu, J., Liu, Z., and Li, J. Kimina Lean Server: A high-performance Lean server for large-scale verification. In *NeurIPS 2025 Workshop on AI for Math*, 2025. arXiv preprint arXiv:2504.21230.

Dressler, O. leanclient: A Python client for interacting with the Lean 4 language server. https://github.com/oOo0oOo/leanclient, 2025.

FrenzyMath. jixia: A static analysis tool for Lean 4. https://github.com/frenzymath/jixia, 2024.

GasStationManager. Safeverify: A Lean 4 script for robustly verifying submitted proofs of theorems and implementations of functions. https://github.com/GasStationManager/SafeVerify, 2025.

Hubert, T. et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 2025. doi: 10.1038/s41586-025-09833-y.

Lean community. A read–eval–print loop for Lean 4. https://github.com/leanprover-community/repl, 2023.

Lean FRO. lean4checker: Replay the Environment for a given Lean module. https://github.com/leanprover/lean4checker, 2023. Integrated into Lean 4 v4.28.0 as `leanchecker`; repository archived 2026.

Lean FRO. Comparator: A trustworthy judge for Lean proofs. https://github.com/leanprover/comparator, 2025.

Liu, J., Zhou, Z., Zhu, Z., Dos Santos, M., He, W., Liu, J., Wang, R., Xie, Y., Zhao, J., Wang, Q., Zhi, L., Li, J., and Li, W. Numina-Lean-Agent: An open and general agentic reasoning system for formal mathematics. *arXiv preprint arXiv:2601.14027*, 2026.

Math Inc. Gauss: An agent for autoformalization. https://www.math.inc/gauss, 2025.

Mistral AI. Leanstral: An open-source code agent for Lean 4. https://mistral.ai/news/leanstral, 2026.

Ospanov, A., Farnia, F., and Yousefzadeh, R. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. *arXiv preprint arXiv:2505.05758*, 2025.

Poiroux, A., Kunčak, V., and Bosselut, A. LeanInteract: A Python interface for Lean 4. https://github.com/augustepoiroux/LeanInteract, 2025.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., et al. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. In *Conference on Language Modeling (COLM)*, 2024. arXiv preprint arXiv:2310.04353.

Thakur, A., Tsoukalas, G., Durrett, G., and Chaudhuri, S. ProofWala: Multilingual proof data synthesis and theorem-proving. *arXiv preprint arXiv:2502.04671*, 2025.

The Harmonic Team. Aristotle: IMO-level automated theorem proving. *arXiv preprint arXiv:2510.01346*, 2025.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

Varambally, S., Voice, T., Sun, Y., Chen, Z., Yu, R., and Ye, K. Hilbert: Recursively building formal proofs with informal reasoning. *arXiv preprint arXiv:2509.22819*, 2025.

Wang, H. et al. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Yang, K., Swope, A., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R., and Anandkumar, A. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

in-flight	median latency	p90 latency	throughput	scaling
1	0.96 s	5.73 s	0.29 req/s	1.00×
2	0.97 s	5.80 s	0.57 req/s	1.98×
4	0.98 s	5.79 s	1.11 req/s	3.87×
8	**1.05 s**	**5.99 s**	**2.09 req/s**	**7.27×**
16	4.68 s	11.4 s	2.09 req/s	7.27×

Table 5. AXLE's per-request median latency, p90 latency, and aggregate throughput at varying in-flight concurrency on a `r7a.2xlarge` (8 vCPU), on the 5,000-proof Goedel-Lean4.27 `check` workload. "Scaling" is throughput relative to c=1. Throughput scales linearly to the box's vCPU count; pushing beyond 8 concurrent requests oversubscribes the physical cores, inflating latency without adding throughput.

A. Further Evaluation

A.1. Experimental Setup

Hardware and isolation. Every measurement runs on a single AWS EC2 instance, with the specific instance varying across experiments. All systems are pre-warmed before measurement with a single `import Mathlib` request. The per-request budget is 150 s for the `check` workload (§5.1) and 600 s for the `verify_proof` workload (§5.2). For AXLE, the LEAN executor workflow is isolated from the client-interaction logic (rate-limiting, queueing, load balancing, etc.). For a fair comparison with raw LEAN invocations and Kimina, measurements only include per-request computation. Public users should expect additional (minimal) server-side and network overhead.

Workloads. The latency/throughput comparison (§5.1) uses 5,000 random proofs sampled with a fixed seed from the public `banach1729/goedel-workbook-lean427` HuggingFace dataset (banach1729, 2026)—DeepSeek-Prover-V1.5 outputs on the Lean Workbook competition-math corpus, published as compilable LEAN files migrated to LEAN 4.27.0. Every system and concurrency setting sees the same 5,000 in the same order. Imports are normalised to a single `import Mathlib` and candidates with non-Mathlib dependencies are dropped (discussed in §7). The dataset's published claim is that 94.1% of its proofs compile cleanly on the target snapshot; our sample lands at 94.16% (4708 / 5000) under all three systems, which is consistent with the reported figure. The strict-checker comparison (§5.2) uses a separate 1,000-request paired `verify_proof` sample drawn randomly from production traffic; each row pairs a target formal statement with a candidate proof from the same production request.

Latency/throughput setup (§5.1). AXLE, Kimina Lean Server, and the direct-invocation baseline all run on a single `r7a.2xlarge` (8 vCPU, 64 GiB), each at concurrency 8. Production AXLE's `check` does some extra bookkeeping Kimina's `/verify` does not—custom per-declaration logging and sorry-state extraction—so for the comparison we disable that bookkeeping in AXLE, leaving the same parse-and-elaborate work Kimina does. Kimina is built from source against LEAN 4.27.0 and runs with its default settings. The direct-invocation baseline runs separate LEAN 4.27.0 subprocesses, one per request.

Strict-checker setup (§5.2). AXLE, Comparator, and SafeVerify all run on a single `r7a.4xlarge` (16 vCPU, 128 GiB) at concurrency 4. Each alternative is driven through a thin Python wrapper that compiles target and submission to oleans (for SafeVerify) or stages them in a shared Lake project (for Comparator), then runs the upstream binary. The per-request budget is 600 s for every system; Comparator's median replay is 95.7 s but several take up to 10 minutes, and 62 requests exceed the time budget. AXLE is configured with `use_def_eq=False` on the per-target signature check, matching what Comparator and SafeVerify do (neither reduces expressions before comparing).

A.2. Concurrency Scaling for AXLE on the Same Instance

To show how AXLE scales by number of in-flight requests, we sweep its concurrency on the same 5,000-proof Goedel-Lean4.27 workload and the same `r7a.2xlarge` as §5.1.

As expected, throughput scales near-linearly from c=1 through c=8. Median and p90 latency remain flat in the same range, confirming that adding workers does not slow individual requests; it only adds capacity. Past c=8 the workers oversubscribe the 8 physical cores: throughput plateaus at 2.09 req/s while median latency jumps roughly 4.5× (to 4.68 s) and p90 nearly doubles. The verdict split is identical at every concurrency (4708 / 287 / 5); concurrency changes the speed, not the verdict.

B. Tool Demo

This appendix walks through using AXLE to prove a non-trivial LEAN 4 lemma. Taken from the public `starting_demo` notebook in the AXLE examples repository,[3] it illustrates the *subproblem decomposition* pattern: Starting from an LLM-generated proof attempt that does not compile end-to-end, we use AXLE to break it into independent sub-obligations, solve each, and reassemble the result.

The target statement

We aim to prove a divisibility lemma on greatest common divisors of consecutive odd numbers.

```
import Mathlib

lemma gcd_dvd_two_mul_diff (m n : ℕ) :
    (Nat.gcd (2 * m + 1) (2 * n + 1) : ℤ) ∣ 2 * ((m : ℤ) - (n : ℤ)) := by sorry
```

The initial LLM attempt

The agent produces the following proof attempt—a reasonable two-step strategy, but with tactics that do not all elaborate cleanly:

```
lemma gcd_dvd_two_mul_diff (m n : ℕ) :
    (Nat.gcd (2 * m + 1) (2 * n + 1) : ℤ) ∣ 2 * ((m : ℤ) - (n : ℤ)) := by
  have h1 : Nat.gcd (2 * m + 1) (2 * n + 1) ∣ (2 * m + 1) - (2 * n + 1) := by
    apply Nat.gcd_dvd_sub
  have h2 : (2 * m + 1) - (2 * n + 1) = 2 * (m - n) := by ring
  rw [h2] at h1
  rcases le_or_gt n m with hn | hn
  · rw [← Nat.cast_sub hn]; exact_mod_cast h1
  · rw [show (2 : ℤ) * (↑m - ↑n) = -(↑(2 * n + 1) - ↑(2 * m + 1)) from by push_cast; ring
    ]
    exact dvd_neg.mpr (dvd_sub (by exact_mod_cast Nat.gcd_dvd_right _ _)
      (by exact_mod_cast Nat.gcd_dvd_left _ _))
```

Calling `check` on the attempt confirms it fails. The relevant errors:

```
error: Unknown constant `Nat.gcd_dvd_sub`
error: unsolved goals
  m n : ℕ
  h1 : (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1)
  ⊢ 1 + m * 2 - (1 + n * 2) = (m - n) * 2
```

Rather than discarding the attempt, we use AXLE to isolate the failing pieces and fix each independently.

Step 1: Decompose with `have2lemma`

`have2lemma` lifts each `have` step into a standalone lemma and rewrites the main proof to invoke it at the call site, preserving the overall structure:

```
result = await client.have2lemma(
    content=initial_attempt,
    reconstruct_callsite=True,
    environment=ENVIRONMENT,
)
decomposed = result.content
```

The decomposed file:

[3] https://colab.research.google.com/github/AxiomMath/axiom-lean-engine/blob/main/examples/starting_demo.ipynb

```
import Mathlib

lemma gcd_dvd_two_mul_diff.h1 (m n : ℕ) :
    (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1) := sorry

lemma gcd_dvd_two_mul_diff.h2 (m n : ℕ)
    (h1 : (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1)) :
    2 * m + 1 - (2 * n + 1) = 2 * (m - n) := sorry

lemma gcd_dvd_two_mul_diff (m n : ℕ) :
    (Nat.gcd (2 * m + 1) (2 * n + 1) : ℤ) ∣ 2 * ((m : ℤ) - (n : ℤ)) := by
  have h1 : Nat.gcd (2 * m + 1) (2 * n + 1) ∣ (2 * m + 1) - (2 * n + 1) :=
    gcd_dvd_two_mul_diff.h1 m n
  have h2 : (2 * m + 1) - (2 * n + 1) = 2 * (m - n) :=
    gcd_dvd_two_mul_diff.h2 m n h1
  rw [h2] at h1
  rcases le_or_gt n m with hn ∣ hn
  · rw [← Nat.cast_sub hn]; exact_mod_cast h1
  · rw [show (2 : ℤ) * (↑m - ↑n) = -(↑(2 * n + 1) - ↑(2 * m + 1)) from by push_cast; ring
    ]
    exact dvd_neg.mpr (dvd_sub (by exact_mod_cast Nat.gcd_dvd_right _ _)
      (by exact_mod_cast Nat.gcd_dvd_left _ _))
```

Step 2: Extract sub-problems with `extract_decls`

We then run `extract_decls` to obtain each sub-problem as an independent, self-contained LEAN snippet. The returned document map contains three entries: `gcd_dvd_two_mul_diff.h1`, `gcd_dvd_two_mul_diff.h2`, and the rewritten main lemma—each with the dependencies it needs to compile in isolation.

Step 3: Solve each sub-problem independently

The first sub-problem is a straightforward divisibility fact that the agent solves on the first attempt:

```
lemma gcd_dvd_two_mul_diff.h1 (m n : ℕ) :
    (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1) := by
  exact Nat.dvd_sub (Nat.gcd_dvd_left _ _) (Nat.gcd_dvd_right _ _)
```

The second is a simple arithmetic identity, so we save the LLM call and hand it directly to `repair_proofs`:

```
repaired = await client.repair_proofs(
    content=h2_sorry,
    names=["gcd_dvd_two_mul_diff.h2"],
    environment=ENVIRONMENT,
)
```

The tool closes the goal with a single terminal tactic:

```
lemma gcd_dvd_two_mul_diff.h2 (m n : ℕ)
    (h1 : (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1)) :
    2 * m + 1 - (2 * n + 1) = 2 * (m - n) := by grind
```

Step 4: Reassemble with `merge`

We `merge` the two solved sub-problems with the main decomposed proof. Declarations with matching types are reconciled by preferring the compiling version, and the surviving declarations are topologically ordered:

```
result = await client.merge(
    documents=[h1_proof, repaired.content, decomposed],
    environment=ENVIRONMENT,
)
final_proof = result.content
```

The merged output:

```
import Mathlib

lemma gcd_dvd_two_mul_diff.h1 (m n : ℕ) :
    (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1) := by
  exact Nat.dvd_sub (Nat.gcd_dvd_left _ _) (Nat.gcd_dvd_right _ _)

lemma gcd_dvd_two_mul_diff.h2 (m n : ℕ)
    (h1 : (2 * m + 1).gcd (2 * n + 1) ∣ 2 * m + 1 - (2 * n + 1)) :
    2 * m + 1 - (2 * n + 1) = 2 * (m - n) := by grind

lemma gcd_dvd_two_mul_diff (m n : ℕ) :
    (Nat.gcd (2 * m + 1) (2 * n + 1) : ℤ) ∣ 2 * ((m : ℤ) - (n : ℤ)) := by
  have h1 : Nat.gcd (2 * m + 1) (2 * n + 1) ∣ (2 * m + 1) - (2 * n + 1) :=
    gcd_dvd_two_mul_diff.h1 m n
  have h2 : (2 * m + 1) - (2 * n + 1) = 2 * (m - n) :=
    gcd_dvd_two_mul_diff.h2 m n h1
  rw [h2] at h1
  rcases le_or_gt n m with hn ∣ hn
  · rw [← Nat.cast_sub hn]; exact_mod_cast h1
  · rw [show (2 : ℤ) * (↑m - ↑n) = -(↑(2 * n + 1) - ↑(2 * m + 1)) from by push_cast; ring
    ]
    exact dvd_neg.mpr (dvd_sub (by exact_mod_cast Nat.gcd_dvd_right _ _)
      (by exact_mod_cast Nat.gcd_dvd_left _ _))
```

Step 5: Final verification

Finally, `verify_proof` confirms that the assembled file actually proves the original formal statement:

```
result = await client.verify_proof(
    formal_statement=formal_statement,
    content=final_proof,
    environment=ENVIRONMENT,
)
# result.okay == True
```

The complete proof is now verified end-to-end. The pipeline used two generative-model calls (the initial attempt and the proof of `h1`); every other step was a mechanical invocation of AXLE's tools. Decompose with `have2lemma` and `extract_decls`, solve sub-problems independently, reassemble with `merge`, and check with `verify_proof`—this is the basic shape of many agentic proving workflows AXLE is designed to support.

C. Strict Verification: Case Studies

This appendix grounds the verification claims in concrete cases, organized into three groups:

1. Two snippets that compile cleanly under `check` but are rejected by `verify_proof`'s rule checks (§C.1).
2. The residual verdict disagreements between `verify_proof` and Comparator/SafeVerify on the 1,000 paired requests of §5.2—inter-tool differences that do not affect soundness on the user-named theorem (§C.2).
3. One adversarial kernel-bypass that `verify_proof` accepts but a kernel-replay checker rejects (§C.3). This is the soundness gap for adversarial-input populations; we have not observed it in production traffic.

C.1. Cases `verify_proof` Catches

Both snippets below compile under plain `check`; `verify_proof` rejects each on a different rule.

Hidden axiom via a custom macro. The snippet below defines a macro `totally_safe_decl` that, despite its name, expands to an `axiom` declaration—introducing `hello_world : False`. A custom `fake_simp` tactic then uses this axiom to "prove" `1 = 2`.

```
syntax (name := totally_safe_decl) (priority := default + 1) declModifiers
  group("totally_safe_decl " declId ppIndent(declSig)) : command

@[macro «totally_safe_decl»] def expandTotallySafeDecl : Macro := fun stx =>
  let stx := stx.modifyArg 1 fun stx =>
    let stx := stx.modifyArg 0 (mkAtomFrom · "axiom" (canonical := true))
    stx.setKind ``Parser.Command.axiom
  pure <| stx.setKind ``Parser.Command.declaration

totally_safe_decl hello_world : False

elab "fake_simp" : tactic => do
  evalTactic (← `(tactic| exact hello_world))

theorem main : 1 = 2 := by exfalso; fake_simp
```

`verify_proof` rejects `main` on the axiom-whitelist check, with error message:

> `In 'main': Axiom 'hello_world' is not in the allowed set of standard axioms`

This is not an entirely contrived pattern. Math Inc's FormalQualBench encountered it in the wild: both Codex and OpenCode submitted solutions that pass a plain `lake build` but reach the goal through an elaborator-level workaround.[4] One `BanachStoneTheorem` submission built an axiom via concatenation (`"ax" ++ "iom"`) and elaborated it at command time, so the keyword never appears literally in the source; the injected axiom then discharged the goal. `verify_proof` properly rejects this pattern.

Namespace shadowing the intended definition. Another case redefines `sqrt` inside a local namespace, then opens that namespace before stating a theorem whose name and surface syntax match the intended MATHLIB statement.

```
namespace hack
def sqrt (r : ℝ) := r
end hack

section
open hack

theorem impossible : sqrt 4 = 4 := by rfl

end
```

The file compiles—`rfl` closes the goal because `hack.sqrt 4` unfolds to `4`. `verify_proof` rejects it on the signature-match check, with error message:

> `Theorem 'impossible' does not match expected signature: expected type` $\sqrt{4} = 4$`, got hack.sqrt 4 = 4`

C.2. Inter-Tool Differences Without Soundness Implications

On the 1,000-request paired sample (§5.2), the only residual verdict disagreement is the 7 SafeVerify-stricter cases of Table 4 (`verify_proof` and Comparator accept; SafeVerify rejects). All 7 follow the same pattern. The candidate places `private` helpers inside a `namespace` and uses them in the proof of a public theorem. LEAN mangles each `private` declaration's internal name with a module path—`_private.`*`Module`*`.0.`*`Namespace.name`*—so the same source compiled into two different modules ends up with two different mangled names. SafeVerify performs a whole-environment comparison between the target's `.olean` and the submission's `.olean`, and reports each mangled private helper present in one olean but not the other as "declaration not found in submission". The public theorem itself (the one named by the

[4] https://www.math.inc/formalqualbench

formal statement) is identical in both oleans and would type-check, but SafeVerify's checks the entire environment and rejects the whole submission.

A minimal example:

```
import Mathlib

namespace Paper

private def castForm (x : ℝ) : ℝ := x + 1

private lemma castForm_add (x y : ℝ) :
    castForm (x + y) = castForm x + castForm y - 1 := by
  unfold castForm; ring

-- the user-named theorem the formal statement asks us to prove
theorem main (x : ℝ) : castForm x = x + 1 := rfl

end Paper
```

In the target olean (compiled from the formal statement), `castForm` and `castForm_add` are stored as `_private.Target.0.Paper.castForm` and `_private.Target.0.Paper.castForm_add`. In the submission olean (compiled from the candidate), the same source produces `_private.Submission.0.Paper.castForm` and `_private.Submission.0.Paper.castForm_add`. SafeVerify walks every declaration in the target olean and looks it up by name in the submission olean; the `_private.Target.*` names have no counterpart on the submission side, and SafeVerify reports them as missing:

```
Found a problem in Submission.olean with declaration
  _private.Target.0.Paper.castForm:
  declaration not found in submission
Found a problem in Submission.olean with declaration
  _private.Target.0.Paper.castForm_add:
  declaration not found in submission
```

`verify_proof` and Comparator only check for matching signatures on the named theorem (`Paper.main`) and accept. None of these cases reflect a soundness gap on the user-named theorem.

C.3. Cases `verify_proof` Does Not Catch

`verify_proof` assumes every declaration in the environment was added through the standard, kernel-checked path, and so does not catch inputs that smuggle declarations into the environment by other means. The canonical example, from lean4checker's adversarial test suite, installs a declaration directly through `Environment.addDeclCore` with kernel checking disabled:

```
import Lean.Elab.Term

open Lean in
run_elab
  modifyEnv fun env => Id.run do
    let decl := .thmDecl { name := `false, levelParams := [],
                           type := .const ``False [], value := .const ``False [] }
    let .ok env := env.addDeclCore (doCheck := false) 0 decl none |
      let _ : Inhabited Environment := ⟨env⟩
      unreachable!
    env
```

The result is an environment with a declaration `false : False`, never typechecked. `verify_proof` accepts `false : False` as-is; a kernel-replay checker like lean4checker (Lean FRO, 2023) re-runs the environment through the kernel from a clean state, detects the ill-formed declaration, and rejects.

Related kernel-bypass patterns work similarly: They evade the kernel's normal type-checking step at elaboration time and then use the unchecked declaration in the proof. The lean4checker, Comparator (Lean FRO, 2025), and SafeVerify (GasSta-

tionManager, 2025) test suites collect cases of this form.[5] None of these patterns arise in the cooperating-client population `verify_proof` targets; workloads verifying adversarial or external code should layer a strict checker on top. Readers interested in the broader landscape of LEAN 4 proof verification can consult the LEAN reference.[6]

D. Resources

Resource	URL or command
Service	https://axle.axiommath.ai
Documentation	https://axle.axiommath.ai/v1/docs/
Python SDK and CLI	`pip install axiom-axle`
MCP server	`pip install axiom-axle-mcp`, or hosted at https://mcp.axiommath.ai
Example notebook	https://colab.research.google.com/github/AxiomMath/axiom-lean-engine/blob/main/examples/starting_demo.ipynb
Release announcement	https://axiommath.ai/territory/releasing-axle
Discussion	Axiom Lean Engine topic on the Lean Zulip
Email support	axle@axiommath.ai

[5] https://github.com/leanprover/lean4checker/tree/master/Lean4CheckerTests, https://github.com/leanprover/comparator/tree/master/tests/projects, https://github.com/GasStationManager/SafeVerify/tree/main/SafeVerifyTest.

[6] https://lean-lang.org/doc/reference/latest/ValidatingProofs/