UlamAI Prover: An Open-Source Lean 4 Theorem Prover and Formalizer

Przemyslaw Chojecki

ulam.ai

February 16, 2026

Abstract

This technical report analyzes the *UlamAI Prover* codebase as a proof-search and autoformalization scaffold targeting Lean 4. The system follows a "truth-first" design: candidate steps are proposed by an LLM but accepted only if verified by Lean (via LeanDojo/Pantograph or the Lean CLI). We provide a code-level architecture map; describe the implemented proof-search loops (best-first and scripted), retrieval interfaces (token overlap and embedding cosine similarity), and the iterative TeX→Lean formalization engine (typecheck–repair, optional statement equivalence checks, and lemma-first expansion guided by trace logs). We also identify engineering risks visible in the current snapshot (notably a CLI flag collision around `-lean`) and place UlamAI in the broader Lean automation landscape (LeanDojo, Lean Copilot, LeanAide, Aristotle SDK) and autoformalization research (ProofNet, type-check filtering, process-driven supervision). All claims in this report are based on direct static inspection of the UlamAI repository snapshot. [1]

Scope, artifact, and method

This report is *only* about the UlamAI prover implementation: we analyze the repository structure, command-line interface, Python modules, and in-repo documentation (`readme.md`, `docs/pipeline.md`) from the snapshot referenced in [1]. We do *not* claim new benchmark results or performance measurements here; the contribution is a precise, implementation-grounded description of how UlamAI works today, where the trust boundaries sit, and what research directions follow naturally from the existing design.

Background: Lean as a verification kernel

Lean is a dependently typed programming language and interactive theorem prover whose trusted kernel checks proof terms, enabling a small "soundness core" even when automation and metaprogramming are extensive. [3, 2] Large-scale formalization in Lean is practically enabled by `mathlib`, a community-maintained library spanning algebra, analysis, topology, measure theory, and more. [4] Lean projects typically rely on Lake (build tool) and Elan (toolchain manager) to pin toolchains and manage dependencies. [5, 6]

UlamAI builds on this ecosystem by treating Lean as an oracle for correctness and organizing LLM interaction as a proposal mechanism whose outputs are filtered through Lean. [1]

System overview

UlamAI exposes a single entrypoint `ulam` (Python console script) with subcommands including `prove`, `formalize`, `bench`, `replay`, `auth`, and `lean-setup`, plus an interactive menu when invoked without arguments. [1] At a high level, the architecture is:

Proving. `ProofState` → (retrieve premises) → (LLM proposes tactics) → (Lean executes) → (search updates frontier) → proof or failure, with every attempt logged to JSONL traces. [1]

Formalization. TeX is segmented into theorem-like units; an LLM drafts Lean with `sorry`; Lean typechecks; errors trigger repair prompts; optional equivalence checks compare TeX vs Lean statements; proof search is optionally run to replace `sorry`; failures can trigger lemma planning and lemma expansion driven by trace summaries. [1]

Proof search in UlamAI

State representation and trust boundary

UlamAI represents proof progress with `ProofState(key, pretty)`. The `pretty` field is a string serialization of the Lean goal state (as returned by LeanDojo/Pantograph or mocks). The trusted boundary is clear: a tactic proposal is *not* accepted by the system unless Lean returns `ok=True` (and ultimately `is_solved=True`). [1] This "LLM-as-policy, Lean-as-checker" pattern aligns with the retrieval-augmented theorem proving approach popularized in LeanDojo and related systems. [7]

Best-first search with beam cap

The default search engine maintains a priority queue of nodes scored by proof length (shorter is better). It records a transposition map `best_seen : state_key → best score` and a step cache keyed by `(state_key, tactic)` to avoid repeated Lean calls. [1] This is a pragmatic, reproducible baseline: it provides determinism under a fixed LLM seed/model and allows post-hoc analysis via traces, but does not implement a learned value function or heuristic beyond proof length.

Table 1: Code-level map of core UlamAI components (paths are within the repository snapshot). [1]

Module / file	Primary role in the system
`ulam/cli.py`	CLI entrypoint; argument parsing; chooses runner/LLM/retriever; dispatches to proving, formalization, benchmark, replay, auth, lean setup; implements axiom guardrail checks and proof insertion into files.
`ulam/search/best_first.py`	Two proof-search strategies: best-first with beam cap and transposition (`best_seen`); and a sequential "scripted" loop; includes repair attempts and optional autop fallback tactics.
`ulam/lean/dojo.py`	LeanDojo/Pantograph runner: loads `sorry` goals, exposes `start/apply`, and wraps Lean goal states as `ProofState`.
`ulam/lean/cli_check.py`	Lean CLI typecheck using `lake env lean` when available; used as an alternate backend for formalization typechecking.
`ulam/llm/*`	LLM clients (OpenAI-compatible HTTP, Anthropic HTTP, Ollama HTTP, Codex/Claude CLI wrappers, mocks) and prompt/tactic parsing utilities.
`ulam/retrieve/*`	Retrievers: token-overlap (`SimpleRetriever`) and embedding cosine similarity (`EmbeddingRetriever`) over a premises file; embedding client speaks OpenAI-compatible `/v1/embeddings`.
`ulam/trace.py`, `ulam/types.py`	JSONL trace logging of each attempted tactic (`ProofStep`); core dataclasses for states and results.
`ulam/formalize/*`	Formalization engine: TeX segmentation, LLM drafting/repair, typecheck loop, optional statement equivalence checks, proof-search integration, lemma-first planning/expansion.
`docs/pipeline.md`	In-repo design narrative for the intended end-to-end pipeline; partially aspirational relative to the current implementation.

Scripted search and repair

A second strategy, "scripted search," requests a short sequence of tactics and executes them in order, with an optional repair mode that triggers when a line fails. [1] Both best-first and scripted modes include a repair loop: on failure, the LLM is asked to propose repaired tactics conditioned on the failure message.

Autop fallback tactics

When enabled, UlamAI augments LLM suggestions with a small fixed set of fallback tactics: `aesop`, `simp`, `ring_nf`, `linarith`, `nlinarith`. [1] This mirrors a common engineering pattern in Lean automation: leverage strong built-in and library tactics as cheap "local solvers" and use the LLM primarily for choosing when and how to apply them. [4]

Lean backends: LeanDojo/Pantograph vs Lean CLI

LeanDojo runner (Pantograph server)

The primary proving backend is `LeanDojoRunner`, which depends on `pantograph.Server` and uses `load_sorry` to obtain goal states at each `sorry` in a file, then applies tactics using `goal_tactic`. [1, 9, 8] This design is consistent with LeanDojo's aim of programmatic interaction with Lean environments for ML-driven proving, including premise selection and tactic generation. [7]

Lean CLI typechecking for formalization

For formalization, UlamAI can typecheck with the Lean CLI (prefer `lake env lean`) as an alternative to Pantograph-based checking. This matters because typechecking is a central feedback signal in modern autoformalization: candidate translations can be filtered by compiler/kernel acceptance. [1, 5] The broader autoformalization literature explicitly exploits such typecheck feedback at scale (e.g., type-check filtering with self-consistency and related verifier training). [12]

Retrieval interfaces implemented

UlamAI offers two retrieval implementations over a user-supplied premises file:

Token-overlap retrieval

`SimpleRetriever` tokenizes the pretty-printed proof state and each premise using whitespace splitting and ranks premises by intersection size. [1] This is lightweight and deterministic but does not handle morphology, symbols, or semantic similarity.

Embedding-based retrieval

`EmbeddingRetriever` computes embeddings for premises (with caching keyed by a SHA-256 hash of the joined premise list) and ranks by cosine similarity to an embedding of the current state. The provided embedding client targets an OpenAI-compatible `/v1/embeddings` endpoint. [1] In principle this aligns with premise selection practices in retrieval-augmented provers (e.g., LeanDojo/ReProver). [7]

LLM prompting and tactic parsing

UlamAI constructs prompts that include: (i) the current proof state, (ii) retrieved premises, (iii) optional user instruction text, and (iv) optional context file contents. The system message is hard-coded as "You are a Lean 4 theorem prover. Only output tactic lines. No explanations."

[1] Outputs are parsed into single-line tactics with filtering against multi-line constructs, `case` blocks, and similar patterns. [1] This is an explicit attempt to match the operational constraints of LeanDojo-style tactic execution.

Autoformalization engine: implemented loops vs aspirational plan

UlamAI includes a working TeX→Lean formalization pipeline in `ulam/formalize`. It is important to distinguish what is *implemented* from what is *described as a goal* in `docs/pipeline.md`. [1]

Implemented: segmentation, drafting, and typecheck–repair

The segmenter is regex-based and recognizes LaTeX environments `definition`, `lemma`, `theorem`, `proposition`, `corollary`, `example`, `proof`. Proof environments can be attached to the preceding theorem-like segment. [1] The engine drafts Lean code via an LLM (or a fallback wrapper that embeds the original TeX in a block comment) and then iterates:

$$\text{Draft} \to \text{Typecheck} \to \text{Repair prompt if failing} \to \cdots$$

until typechecking succeeds or a repair budget is exhausted. [1] This loop is conceptually aligned with typecheck-filtered decoding methods that treat the compiler/kernel as a high-precision validator. [12]

Implemented: statement equivalence checks

Optionally, UlamAI runs LLM-based equivalence checks between TeX statements and extracted Lean declaration headers. Results are recorded and "mismatches" can trigger targeted statement repair prompts. [1] This resembles the broader trend toward adding *semantic validation* layers beyond mere typechecking—a key challenge highlighted in autoformalization benchmarks such as ProofNet. [11]

Implemented: proof filling, lemma-first planning, and lemma expansion

For proofs, UlamAI can attempt to replace `sorry` by calling the prover against the generated Lean file using LeanDojoRunner and scripted search (with autop enabled). [1] In "lemma-first" mode, the system can ask the LLM to generate auxiliary lemmas (inserted directly into the Lean file) and then iteratively expand lemmas when proof attempts fail. Expansion is guided by trace logs: the engine reads recent JSONL steps, summarizes the last goal and failure messages, and conditions the next lemma-generation prompt on these summaries. [1] This is a concrete, code-level instance of a *process-guided* approach: intermediate feedback (goals, failures) becomes supervision for subsequent generation, echoing process-driven autoformalization paradigms in the literature. [13]

Aspirational: richer document understanding

`docs/pipeline.md` describes additional phases (named-entity extraction, dependency graphs, "gap detection" beyond current heuristics). These are not fully present in the current codebase and should be treated as roadmap rather than shipped functionality. [1]

Reproducibility and logging

UlamAI emphasizes reproducible debugging via JSONL traces. Each step includes the pre-state, attempted tactic, success flag, error (if any), next-state key, and whether the step was cached. [1] For formalization, the engine creates an artifact directory (`runs/formalize_YYYYMMDD_HHMMSS/`) containing input TeX, context, segments, config snapshot, per-round Lean files, diffs, and error logs. [1] These choices are consistent with "build-and-replay" scientific norms: the goal is not merely to obtain a proof once, but to make failures inspectable and iteration systematic.

Implementation risks and sharp edges visible in the snapshot

CLI flag collision (`-lean`)

UlamAI's `cli.py` contains a pre-parse check that triggers "Lean setup" mode if `-lean`, `-lean`, or `-lean-setup` appears *anywhere* in `argv`. This conflicts with the `prove` and `bench` subcommands, which also define an option named `-lean` for selecting the Lean backend. In standard `argparse` usage (`-lean dojo`), the token `-lean` appears and can unintentionally route execution into setup mode. [1] This is not a conceptual flaw in the prover, but it is a practical paper-cut that affects reproducibility of experiments unless mitigated (e.g., using `-lean=dojo` or invoking the explicit `lean-setup` subcommand).

Version-string inconsistency

Within the snapshot, version identifiers differ across `readme.md`, `pyproject.toml`, and `PKG-INFO`. This is common in early-stage repos, but it complicates external reporting unless the snapshot hash is used as the primary identifier. [1]

Reliance on `sorry`-anchored goals

The LeanDojo runner locates goals by loading all `sorry` units and selecting the target theorem's corresponding `sorry`. This makes proving workflows explicit and reproducible but imposes a constraint on the input file structure: target theorems must contain a `sorry` placeholder. [1]

Positioning in the Lean automation and autoformalization landscape

UlamAI is best read as an orchestration layer over (i) Lean verification and (ii) LLM proposal mechanisms. It overlaps with, but is distinct from:

- **LeanDojo and ReProver:** toolkits, data, and baselines for interacting with Lean and training retrieval-augmented provers. [7]
- **Pantograph:** a Python server interface used by LeanDojo-v2 and adopted as a backend runner in UlamAI. [9, 8]
- **Lean Copilot:** an in-editor proof assistant for Lean with interactive suggestions. [10]
- **LeanAide:** Lean-side tools emphasizing autoformalization and interactive assistance. [15]
- **Aristotle SDK:** an API-style automated theorem proving interface for Lean that represents an alternate deployment model (service/API vs local CLI). [16]

For evaluation, UlamAI's stated targets (and common community practice) naturally connect to established benchmarks such as miniF2F (formal theorem proving) and ProofNet/LeanEuclid (autoformalization and statement translation). [17, 11, 14]

Research directions suggested by the current design

The current codebase already contains the right *interfaces* to explore several research directions, without requiring a redesign:

Typecheck-filtered decoding at the statement level

UlamAI already loops on typecheck failure for TeX$\to$Lean drafting. A next research step is to turn this into systematic *candidate generation + typecheck filtering* with calibrated selection (e.g., self-consistency over typechecking candidates), as studied for Lean 4 autoformalization. [12]

Process-supervised proof/lemma generation

Lemma expansion currently uses trace summarization as a textual conditioning signal. A natural extension is to record richer process traces (goals, local contexts, failing subgoals) and treat them as supervised data for training or fine-tuning, consistent with process-driven autoformalization approaches. [13]

Better search heuristics beyond proof length

Best-first currently scores by proof length only. UlamAI's trace logs and cached results could support learning a value heuristic that predicts "distance to solved" from `state.pretty`, retrieved premises, and tactic history, in the style of modern RL/IL theorem proving pipelines (while retaining Lean as the verifier). [7]

Benchmark discipline and artifact-based reporting

Because UlamAI writes detailed traces and formalization artifacts, it is well positioned for benchmark reporting that is fully replayable. A concrete next step is to define stable evaluation suites over miniF2F and formalization datasets, where each run is pinned by (model, prompt version, toolchain) and yields auditable artifacts. [17, 5]

Conclusion

UlamAI Prover, as implemented in the analyzed snapshot, is a compact but coherent scaffold for "LLM-propose / Lean-verify" theorem proving and iterative autoformalization in Lean 4. Its strongest current contributions are engineering choices that make the workflow inspectable and reproducible (trace logs, artifact directories, caching) and a concrete implementation of iterative formalization with repair, equivalence checks, and lemma-driven expansion. The codebase also makes clear what remains research-grade and open: principled document understanding, learned search heuristics, robust statement semantics, and systematic evaluation. [1]

References

[1] UlamAI Prover (`ulamai/ulamai`) repository snapshot. GitHub: `https://github.com/ulamai/ulamai`. v.0.1.5

[2] L. de Moura and S. Ullrich, *The Lean 4 Theorem Prover and Programming Language*, in *Automated Deduction – CADE 28*, LNCS 12699, Springer (2021). DOI: `https://doi.org/10.1007/978-3-030-79876-5_37`.

[3] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, *The Lean Theorem Prover (System Description)*, in *Automated Deduction – CADE-25*, LNCS 9195, Springer (2015). DOI: `https://doi.org/10.1007/978-3-319-21401-6_26`.

[4] The mathlib Community, *The Lean Mathematical Library*, *Proc. ACM Program. Lang.* (CPP) (2020). DOI: `https://doi.org/10.1145/3372885.3373824`.

[5] Lean Language Reference, *Build Tools and Distribution* (Lake, Elan). `https://lean-lang.org/doc/reference/latest/Build-Tools-and-Distribution/`. Accessed 2026-02-15.

[6] leanprover community, *Elan: The Lean version manager*. `https://github.com/leanprover/elan`. Accessed 2026-02-15.

[7] K. Yang et al., *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*, NeurIPS (Datasets and Benchmarks Track) (2023). arXiv: `https://arxiv.org/abs/2306.15626`.

[8] LeanDojo project, *LeanDojo-v2* documentation and repository. `https://leandojo.org/`. Accessed 2026-02-15.

[9] Jakob Schwab and contributors, *Pantograph: The Lean4 interaction library* (PyPantograph). `https://github.com/lean-dojo/Pantograph`. Accessed 2026-02-15.

[10] P. Song et al., *Lean Copilot: Large Language Models as Copilots for Theorem Proving in Lean*, NeurIPS (2025). Project page: `https://leancopilot.github.io/`. Accessed 2026-02-15.

[11] Z. Azerbayev et al., *ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics*, arXiv (2023). arXiv: `https://arxiv.org/abs/2302.12433`.

[12] A. Poiroux, G. Weiss, V. Kunčak, and A. Bosselut, *Improving Autoformalization using Type Checking*, arXiv (2024). arXiv: `https://arxiv.org/abs/2406.07222`.

[13] J. Lu et al., *Process-Driven Autoformalization in Lean 4*, arXiv (2024). arXiv: `https://arxiv.org/abs/2406.01940`.

[14] L. Murphy et al., *Autoformalizing Euclidean Geometry*, ICML (2024). arXiv: `https://arxiv.org/abs/2405.17216`.

[15] S. Gadgil, *LeanAide: Tools based on AI for helping with Lean 4*. `https://github.com/siddhartha-gadgil/LeanAide`. Accessed 2026-02-15.

[16] *aristotlelib* (Aristotle SDK), PyPI package page. `https://pypi.org/project/aristotlelib/`. Accessed 2026-02-15.

[17] K. Zheng, J. M. Han, and S. Polu, *MiniF2F: a cross-system benchmark for formal Olympiad-level mathematics*, arXiv (2021). arXiv: `https://arxiv.org/abs/2109.00110`. Repository: `https://github.com/openai/miniF2F`. Accessed 2026-02-15.